Measurable Scenario Description Language Reference

Open Source Version 0.9

Last generated: September 19, 2019

Table of Contents

M-SDL Reference

1. Introducing M-SDL

To verify the safety of an autonomous vehicle (AV) or an advanced driver assistance system (ADAS), you need to observe its behavior in various situations, or scenarios. Using M-SDL, you can create scenarios that describe the behavior of the AV as well as other actors in the environment, such as other vehicles, pedestrians, weather, road conditions and so on.

Because M-SDL scenarios are high-level, abstract descriptions, you can create many concrete variants of a scenario by varying the scenario parameters, such as speed, vehicle type, weather conditions and so on. M-SDL tools can generate these variants automatically, within the constraints that you specify. These tools then collect and aggregate parameter data from successful tests, thus enabling you to measure the safety of your AV.

The Measurable Scenario Description Language (M-SDL) is a mostly declarative programming language. The only scenario that executes automatically is the top-level scenario. You control the execution flow of the program by adding scenarios to the top-level scenario.

M-SDL is an aspect-oriented programming language. This means you can modify the behavior or aspects of some or all instances of an object to suit the purposes of a particular verification test, without disturbing the original description of the object.

2. Using M-SDL

Summary: This topic shows how to create and reuse M-SDL scenarios.

M-SDL is a small, domain-specific language designed for describing scenarios where actors (sometimes called *agents*), such as cars and pedestrians, move through an environment. These scenarios have parameters that let you control and constrain the actors, the movements and the environment.

M-SDL is designed to facilitate the composition of scenarios and tests, making it possible to define complex behaviors using your own methodology. A minimal, extensible set of actors and scenarios comprise the fundamental building blocks. Some built-in scenarios perform tasks common to all scenarios, such as implementing parallel execution. Others describe relatively complex behavior, such as the **car.drive** scenario. By calling these scenarios, you can describe even more complex behavior, such as a vehicle approaching a yield sign. For further complexity, multiple scenarios can be mixed. For example, a weather scenario can be mixed with a car scenario.

It is easy to create new actors and new scenarios as the need arises, either from scratch, or using what you have defined so far. For example, the scenario **cut_in**, presented below, is defined using the scenario **car.drive**.

There will eventually be a standard scenario library, possibly containing both the **drive** and **cut_in** scenarios, but organizations will be able to add or customize scenarios as needed.

2.1. M-SDL building blocks

The building blocks of M-SDL are data structures:

- Simple structs – a basic entity containing attributes, constraints and so on.
- Actors – like structs, but also have associated scenarios.
- Scenarios – describe the behavior of actors.

These structures have attributes that hold scalar values, lists, and other structures. Attribute values can be described as expressions or calculated by external method definitions. You can control attribute values with **keep()** constraints, for example:

```
keep(speed < 50kph)
```

You can control attribute values in scenarios either with **keep()** constraints or with scenario modifiers such as **speed()**.

```
speed(20kph, faster_than: car1)
```

Structures also define events, for example:

```
event too_close is (distance_between(car1, car2) < 10m)
```

You can describe scenario behavior by calling the built-in scenarios. You can call the operator scenarios **serial**, **parallel**, or **mix**, to implement your scenario in a serial or parallel execution mode or to mix it with another scenario. Other built-in scenarios implement time-related actions, such as emit, wait, or error reporting.

2.2. Example scenarios

Now let's look at some examples.

Example 1 shows how to define and extend an actor. The actor **car_group** is initially defined with two attributes. Then it is extended in a different file to add another attribute.

Example 1

```
# Define an actor
actor car_group:
    average_distance: distance
    number_of_cars: uint

# Extend an actor in a separate file
import car_group.sdl

extend car_group:
    average_speed: speed
```

Example 2 shows how to define a new scenario called **two_phases**. It defines a single attribute, **car1**, which is a **green truck**. It uses the **serial** and **parallel** operators to activate the **car1.drive** scenario, and it applies the **speed()** modifier.

two_phases works as follows:

- During the first phase, **car1** accelerates from 0 kph to 10 kph.
- During the second phase, **car1** keeps a speed of 10 to 15 kph.

Note: **two_phases** is very concrete because the value for each parameter is defined explicitly. We'll see how to define more abstract scenarios later.

Example 2

```
# A two-phase scenario
scenario traffic.two_phases:   # Scenario name
    # Define the cars with specific attributes
    car1: car with:
        keep(it.color == green)
        keep(it.category == truck)

    path: path # a route from the map; specify map in the test

    # Define the behavior
    do serial:
        phase1: car1.drive(path) with:
            spd1: speed(0kph, at: start)
            spd2: speed(10kph, at: end)
        phase2: car1.drive(path) with:
            speed([10..15]kph)
```

Example 3 shows how to define the test to be run:

1. Import the proper configuration. In this case we want to run this test with the SUMO simulator.

2. Import the **two_phases** scenario we defined before.

3. Extend the predefined, initially empty **top.main** scenario to invoke the imported **two_phases** scenario.

Example 3

```
import sumo_config.sdl
import two_phases.sdl

extend top.main:
    set_map("/maps/hooder.xodr") # specify map to use in this test
    do two_phases()
```

Example 4 shows how to define the **cut in** scenario. In it, **car1** cuts in front of the **dut.car**, either from the left or from the right. **dut.car**, also called the **ego** car, is predefined.

Note: This scenario is more abstract than **two_phases.** We'll see later how we can make it more concrete if needed.

It has three parameters:

- The car doing the cut_in (**car1**).
- The side of the cut_in (left or right).
- The path (road) used by the two cars, constrained to have at least two lanes.

Then we define the behavior:

- In the first phase, **car1** gets ahead of the **ego**.
- In the second phase, **car1** cuts in front of the **ego.**

The scenario modifiers **speed()**, **position()** and **lane()** are used here. Each can be specified either in absolute terms or in relationship to another car in the same phase. Each can be specified for the whole phase, or just for the start or end points of the phase.

Example 4

```
# The cut-in scenario

scenario dut.cut_in():
    car1: car        # The other car
    side: av_side    # A side: left or right
    path: path

    path_min_driving_lanes(path, 2) # Needs at least two lanes

    do serial():
        get_ahead: parallel(duration: [1..5]s): # get_ahead is a label
            dut.car.drive(path) with:
                speed([30..70]kph)
            car1.drive(path, adjust: true) with:
                position(distance: [5..100]m,
                     behind: dut.car, at: start)
                position(distance: [5..15]m,
                    ahead_of: dut.car, at: end)
        change_lane: parallel(duration: [2..5]s): # change_lane is a label
            dut.car.drive(path)
            car1.drive(path) with:
                lane(side_of: dut.car, side: side, at: start)
                lane(same_as: dut.car, at: end)
```

Example 5 shows how to define the **two_cut_in** scenario using the **cut_in** scenario. It executes a cut_in from the left followed by a cut_in from the right. Furthermore, the colors of the two cars involved are constrained to be different.

Example 5

```
# Do two cut-ins serially
import cut_in.sdl

scenario dut.two_cut_ins():
    do serial():
        c1: cut_in() with:        # c1 is a label
            keep(it.side == left)
        c2: cut_in() with:        # c2 is a label
            keep(it.side == right)
            keep(c1.car1.color != c2.car1.color)
```

Example 6 shows how to run **cut_in** with concrete values. The original **cut_in** specified ranges, so by default, each run would choose a random value within that range. However, you can make the test as concrete as you want using constraints.

Example 6

```
# Run cut_in with concrete values
import cut_in.sdl

extend top.main:
    do dut.cut_in() with:
        keep(it.get_ahead.duration == 3s)
        keep(it.change_lane.duration == 4s)
```

Example 7 shows how to mix multiple scenarios: the **cut_in** scenario, another scenario called **interceptor_at_yield**, and a **set_weather** scenario. The **mix_dangers** scenario has a single attribute of type **weather_kind**, which is constrained to be not **nice**, because we want a dangerous situation. This attribute is passed to **set_weather**.

Example 7

```
# Mixing multiple scenarios
import_my_weather.sdl
import interceptor.sdl
import interceptor_at_yield.sdl
import cut_in.sdl

scenario dut.mix_dangers:
    weather: weather_kind
    keep(weather != nice)

    do mix():
        cut_in()
        interceptor_at_yield()
        set_weather(weather)
```

Example 8 runs **mix_dangers**. In this case we chose to specify a concrete weather (**rain**) rather than letting it be a random, non-nice weather.

Example 8:

```
# Activating mix_dangers

import mix_dangers_top.sdl

extend top.main:
    do mix_dangers() with:
        keep(it.weather == rain)
```

3. M-SDL Basics

Summary: This topic describes basic features of the M-SDL language.

3.1. Lexical conventions

M-SDL is similar to Python. An M-SDL program is composed of statements that declare or extend types such as structs, actors, scenarios, or import other files composed of statements. Each statement includes an optional list of members indented one unit (a consistent number of spaces) from the statement itself. Each member in the block, depending on its type, may have its own member block, indented one unit from the member itself. Thus, the hierarchy of an M-SDL program and the place of each member in that hierarchy is indicated strictly by indentation. (In C++, the beginning and end of a block is marked by curly braces {}.)

In the following figure, the code blocks are indicated with a blue box.

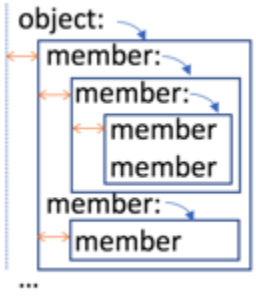

Figure 1 Code blocks

Common indentation indicates members at the same level of hierarchy. It is recommended to use multiples of four spaces (blanks) to indicate successive hierarchical levels, but multiples of other units (two, three and so forth) are allowed, as long as usage is consistent. Inconsistent indentation within a block is an error. If you use tabs, you must set the editor to translate tabs to spaces.

Members (other than strings) that are too long to fit in a single physical line can be continued to the next line after placing a backslash character (\) before the newline character.

```
object:
    member ... \
        next line of same member \
        end of member
    member
```

However, a line with an open parenthesis **(** or open square bracket **[** flows across newlines with no need for a backslash character.

You can continue strings onto multiple lines with the + character:

```
"a string" + \
" with continuation"
```

Inline comments are preceded by a hashtag character (#) and end at the end of the line. Block comments are allowed. Each line in the block must begin with the /* characters and end with */. Nested block comments are allowed. Newlines within comments and indentation of comments does not affect code nesting.

Example

```
/*
This is the first line of a block comment.
This is the second line of the block comment.
*/

extend top.main:
    do dut.cut_in() with: # This is an inline comment
        keep(it.get_ahead.duration == 3s)
        keep(it.change_lane.duration == 4s)
```

3.2. Document conventions

This document uses the following conventions to display syntax:

- Required elements and keywords are shown in **bold.**
- Optional elements are shown within square brackets as [*optional*].
- Values that you must specify are shown in *italics*.
- Values that you must choose between are shown with a bar: *choice1* | *choice2*
- Values that you can specify in a list are indicated by *type**, meaning zero or more items of that type in the list, or by *type*+, meaning one or more items in the list.

For example, given the syntax:

```
[!]field-name:type [with:

    member+]
```

- The do-not-generate operator (!) is optional.
- *field-name* and *type* are required. All other items are optional.
- **with**, if specified, must have at least one member

For example, the following field declarations are valid:

```
!current_speed: speed
start_speed: speed with:
    keep(it < 100kph)
cars: list of car with:
    keep(soft cars.size() <= 10)
```

Notes about these examples:

- The do-not-generate operator specifies that no value should be generated for this field before the run executes. Instead, a value is generated or assigned during the run.
- In the second example, the *member* list consists of one constraint on the value of the field. **it** is an implicit variable that refers to the current item, in this case, the **start_speed** field.
- In the third example, the list is constrained to have no more than 10 items.

3.3. User-defined names, constants and keywords

User-defined names in M-SDL code consist of a case-sensitive combination of any length, containing the characters A–Z, a-z, 0-9, and underscore (_). They must begin with a letter.

M-SDL has three predefined constants: **true**, **false**, and **null**.

Entities whose name start with "_" are considered internal or implementation entities. This is just a convention; they can be accessed like any other entity.

The following are keywords, and cannot be used as names:

actor	**emit**	**in**	**list of**	**scenario**	**when**
any	**empty**	**is**	**of**	**soft**	**with**

call	**event**	**is also**	**outer**	**struct**
cover	**extend**	**is first**	**match**	**true**
default	**external**	**is only**	**multi_match**	**try**
do	**false**	**it**	**modifier**	**type**
def	**if**	**keep**	**not**	**undefined**
else	**import**	**like**	**on**	**wait**

3.4. Overview of M-SDL constructs

M-SDL syntax requires that all constructs appear in a valid context. The table below shows the proper context for each construct. Below the table is a brief description of each construct.

Statements	Statements are top-level constructs that are outside of any other construct in the program. Statements include: • Enumerated type declarations • Struct declarations • Actor declarations • Scenario declarations • Scenario modifier declarations • Extensions to those declarations • Import statements

Struct, actor or scenario members	The following constructs can appear only inside a struct, actor or scenario declaration or extension. These constructs include: • Coverage definitions • Events • Field declarations • Field constraints (**keep()**) • External method declarations • **when** subtypes **Note**: Scenarios are associated with specific actors but are declared as top-level statements.
Scenario members	Scenario members appear within a scenario declaration or extension. Scenario members include: • Members allowed in structs or actors • Scenario modifier invocations • **do** (behavior definition)
Scenario invocations	You can invoke the following types of scenarios: • Operator scenarios • Event-related scenarios • Zero-time scenarios • User-defined scenarios
Expressions	Expressions can be used within statements or members and evaluate to values of the specified type.

3.4.1. Statements

Statements are top-level constructs that define or extend a type or import a file composed of statements.

Enumerated type declarations define a set of explicitly named values. For example, an enumerated type **driving_style** might define a set of two values: **normal** and **aggressive**.

Struct declarations define compound data structures that store various types of related data. For example, a struct called **car_collision** might store data about the vehicles involved in a collision.

Actor declarations model entities like cars, pedestrians, environment objects like traffic lights etc. They are compound data structures that store information about these entities. In contrast to structs, they are also associated with scenario declarations or extensions. Thus, an actor is a collection of both related data and declared activity.

Scenario declarations define compound data structures that describe the behavior or activity of one or more actors. You control the behavior of scenarios and collect data about their execution by declaring data fields and other members in the scenario itself or in its related actor or structs. Example scenarios are **car.drive**, **dut.cut_in**, **dut.cut_in_with_person_running**, and so on.

Scenario modifier declarations modify, but do not define, scenario behavior, by constraining attributes such as speed, location and so on. Scenario modifier declarations can include previously defined modifiers.

Extensions to an existing type or subtype of an enumerated type, a struct, an actor or a scenario add to the original declaration without modifying it. This capability allows you to extend a type for the purposes of a particular test or set of tests.

3.4.2. Struct, actor or scenario members

The constructs described in this section can appear only within a struct, actor or scenario declaration or extension.

Cover definitions let you sample key parameters related to scenario execution. Collecting this data over multiple executions of a scenario helps you evaluate the safety of the AV. For example, if a **car** actor has a field **speed**, you probably want to collect the value of this field at key points during a scenario. Cover definitions appear in scenarios in order to have access to the scenario's parameters and to vary coverage definitions according to the scenario.

Field declarations define a named data field of any scalar, struct or actor type or a list of any of these types. The data type of the field must be specified. For example, this field declaration defines a field named **legal_speed** of type **speed**.

```
legal_speed: speed
```

Field constraints defined with **keep()** restrict the values that can be assigned to or generated for data fields. For example, because of the following **keep()**constraint, randomized values for **legal_speed** are held below 120 kph.

```
keep(legal_speed < 120kph)
```

This constraint can also be written as follows, with the implicit variable **it** referring to the **legal_speed** field:

```
legal_speed: speed with:
    keep(it < 120kph)
```

Events define a particular point in time. An event is raised by an explicit **emit** action in a scenario or by the occurrence of another event to which it is bound. Scenarios and scenario phases have three predefined events: **start**, **end** and **fail**.

when subtypes extend an object when a specified condition is **true**. For example, if an actor **my_car** has a field of type **driving_style**, the actor's attributes or behaviors can be different when the value of **driving style** is **aggressive** from when it is **normal**.

External method declarations identify imperative code written in other programming languages, such as C++, Python, and the ***e*** verification language, that you want to call from an M-SDL program. For example, you might want to call an external method to calculate and return a value based on a scenario's parameters.

3.4.3. Scenario members

Scenarios have two members that are not allowed in structs or actors.

Scenario modifiers are scenarios that constrain various attributes of a scenario's behavior. They do not define a scenario's primary behavior. There are both relative and absolute modifiers. In the example below, the **speed()** modifier sets the speed of the affected car to be 1 to 5 kph faster relative to car1:

```
speed([1..5]kph, faster_than: car1)
```

The **do** scenario member defines the behavior of the scenario when it is invoked.

3.4.4. Scenario invocations

Scenario invocations extend the execution of an M-SDL program. A built-in **top.main** scenario is automatically invoked. **top.main** needs to be extended to invoke other scenarios, defining the behavior of the whole SDL program.

3.4.5. Expressions

Expressions are allowed in constructs as specified. The expression must evaluate to the specified type. Expressions can include calls to external value-returning methods.

3.5. M-SDL file structure

M-SDL is designed to facilitate the composition of scenarios. To that end, it lets you separate the following components into separate files:

- Higher-level scenarios describing complex behavior.
- Lower-level scenarios that you invoke from multiple, higher level scenarios.
- Coverage definitions.
- Extensions of any of the above for the purposes of a particular test.
- The definition of a test.
- The configuration of an execution platform.

The test file defines the test by:

- Importing the components of the test.
- Extending the built-in, top-level scenario to
    - Specify the map.
    - Invoke a high-level user scenario.

Here is a simple example of a test file. The **my_scenario_top.sdl** file imports the top-level user scenario, any lower-level scenarios, and the coverage definitions. The scenario invocation constrains the **weather** attribute of **my_scenario** to be **rain**, illustrating how you can constrain the attributes or behavior of a scenario for the purposes of a particular test.

```
import /config/simulator_config.sdl
import my_scenario_top.sdl

extend top.main:
    set_map("/maps/hooder.xodr")
    do a: my_scenario with:
        keep(.weather == rain)
```

3.6. Actor hierarchy and name resolution

MSDL defines a global scope that includes a number of pre-defined actors, in particular the actor **top**. **top** and any of its fields are members of the global scope.

To facilitate readability, it is recommended to use the type's name as the field name for the global field. For example, if **top** is the top-level actor, and **env** is a global actor, it is recommended to declare **env** as follows:

```
extend top:
    env: env
```

It is recommended to represent scenario libraries as new global actors. For example, simulator-specific scenarios for the **my_sim** simulator sit inside the global actor **my_sim.**

When the program is executed, the top-level actor's main scenario is invoked. Extending the top-level scenario to invoke a scenario creates an instance of that scenario in the invoking scenario, as well as instances of all the scenario's members. The hierarchy of the tree expands level by level as each scenario instance calls other scenarios or scenario modifiers.

Objects in the program tree such as scenario instances or fields within scenario instances can be accessed by absolute or relative path expressions, for example **top_actor.scenario1.field1**.

Scenario invocations can have labels, and these labels allow access to them. For example **scenario1_slowdown_drive** accesses a **drive** sub-scenario within the **slowdown** phase of **scenario1**. This access allows the behavior or attributes of a specific scenario or scenario invocation to be controlled from outside the scenario.

3.6.1. Implicit labels

It is recommended to add at least one explicit label for the top-level scenario. In the following example, there is an explicit label for the serial invocation in **top.main**. The remaining invocations are labelled implicitly. If there is ambiguity (more than one object with the same name) a count is added at the end of the label in the form !*n*.

```
extend top.main:
    do a: serial():                 # explicit label a
        car1.drive(path) with:      # implicit label a!car1
            speed(0kph, at: start)  # implicit label a!car1!speed
            speed(10kph, at: end)   # implicit label a!car1!speed!2
```

If there is no explicit label at the top level, the implicit label is the scenario name. Without the explicit label in the example above, the implicit label for **car1.drive** is **serial!car1.**

3.6.2. Scenario name resolution

Scenarios reside in the namespace of their actor, so there can be a **car.turn** and a **person.turn**. When scenario **car1.slow_down** invokes a lower-level scenario **turn**, these rules determine which scenario **turn** is called:

- If you specify the actor instance for **turn** explicitly, for example **car2.turn**, the actor is **car2.**
- Else if **car1**'s actor type (**car**) has a scenario **turn**, then **car1** is used.
- Else if a global actor has a scenario **turn**, then that actor is used.
- Else this is an error.

Notes:

- Invoking the generic form of the scenario (*actor.scenario*) is not allowed, unless the actor is a global actor. A scenario invocation must be associated with an actor instance.
- Scenario modifiers are searched by the same rules as scenarios.

3.6.3. Name resolution for other objects

Here are the scoping rules when referring to an object other than a scenario, such as a field, method or event **x** inside a struct, actor, or scenario **y**.

If you refer to **x** via a path (**car1.x**) then that path is used.

Else if you refer to **x** using the implicit variable **it.x**, the path of the object referred to by **it** is used. (**it** is available only in certain contexts.)

Else if **y** has an object **x**, its path is used.

Else if **y** is a scenario of actor **z**, and **z** has an object **x**, that object's path is used.

Else if else if **x** is in the global scope, **x** is used.

Else this is an error.

3.7. Data types

M-SDL defines the following data types:

- Scalar types hold one value at a time: numeric, Boolean and enumerated.
- List types hold multiple values of one type.
- String types hold a sequence of ASCII characters enclosed in double quotes.
- Resource types hold a list of map junctions and segments.
- Real types hold floating-point values.
- Compound types hold multiple values of multiple types.

3.7.1. Generic numeric types

The generic numeric types are scalar types. You do not need to specify a unit of measurement for generic numeric types. Generic numeric types include signed and unsigned integers of 32 or 64 bits in size.

You can specify integers in decimal or hexadecimal format (prefixed with **0x**). Commas are not allowed, but you can add an underscore for readability, for example, 100_000.

Name	Type
int	32 bit integer
uint	32 bit unsigned integer
int64	64 bit integer
uint64	64 bit unsigned integer

3.7.2. Physical types

Physical types are used to characterize physical movement in space, including speed, distance, angle and so on. When you specify a value for one of these types in an expression, or when you define coverage for it, you must use a unit. As shown in the table below, you have a choice of units for the mostly commonly used types.

Physical constants have implied types. For example, **12.5km** has an implied type of **distance**.

Examples:

```
2meter
1.5s
[30..50\]kph
```

Name	Units
acceleration	kphps (= kph per second), mpsps (= meters per second per second)
angle	deg, degree, rad, radian
angular_speed	degree_per_second, radan_per_second
distance	mm, millimeter, cm, centimeter, in, inch, feet, m, meter, km, kilometer, mile
speed	kph, kilometer_per_hour, mph, mile_per_hour
temperature	c, celsius, f, fahrenheit
time	ms, millisecond, s, sec, second, min, minute, hr, hour
weight	kg, kilogram, ton

3.7.3. Boolean types

The Boolean type is called **bool** and represents truth (logical) values, **true** or **false**.

```
true_value: bool with:
    keep(it == true)
```

3.7.4. Enumerated types

Enumerated types represent a set of explicitly named values. In the following example, the enumerate type **driving_style** has two values, **aggressive** and **normal**.

```
type driving_style: [aggressive, normal]
```

3.7.5. List types

A list is a way to describe a container of similar values in M-SDL. A list can contain any number of elements from a data type. For example, you can declare a convoy to contain a list of car actors, or a shape as a list of points.

List literals are defined as a comma-separated list of items, for example:

```
[point1, point2]
```

The [*n..n*] notation is not allowed for lists; it is reserved for ranges.

Example lists

```
convoy: list of car
shape: list of point
```

Example list assignment

```
shape: list of point with:
    keep(it == [map.explicit_point("-15",0,20m,1),
      map.explicit_point("-15",0,130m,1)]
```

Example list constraint

```
distances: list of distance with:
    keep(it == [12km, 13.5km, 70km])
```

3.7.6. String types

The M-SDL predefined type **string** is a sequence of ASCII characters enclosed in double quotes (“”). Single quotes are not allowed.

In the following example, the implicit variable **it** refers to the field **text**.

```
struct data:
    text: string with:
        keep(it == "Absolute speed of ego at start (in km/h)")
```

The default value of a field of type string is **null**. This is equivalent to an empty string,“”.

Literal strings allow expression interpolation. The **$(***exp***)** converts an expression to a string and inserts it in place. For example:

```
info("There are $(cars.size()) cars.")
```

You can concatenate multiple strings with the + character, for example:

```
"a string" + " with continuation"
```

You can continue strings onto multiple lines with the + character and \ character:

```
"a string" + \
" with continuation"
```

Within a string, the backslash is an escape character, for example:

```
"My name is: \tJoe"
```

3.7.7. Resource types

Resource types include **junction** and **segment**. They hold a global list of locations on the current map.

3.7.8. Real type

The real type holds floating-point values. It is equivalent to C++ **double**.

3.7.9. Compound types

M-SDL defines three built-in compound types:

- **Scenarios** define behavior, such as a car approaching a yield sign, a pedestrian crossing the street, and so on. Scenarios define behavior by activating other scenarios. M-SDL provides a library of built-in scenarios describing basic behavior, such as moving, accelerating, turning and so on.
- **Actors** typically represent physical entities in the environment and allow scenarios to define their behavior. M-SDL provides a set of built-in actors, including **car**, **traffic**, **env**, and so on. If you create an instance of the actor **car** in a program with the name **my_car**, its built-in scenario **drive** can be invoked as **my_car.drive.**
- **Structs** define sets of related data fields and store the values assigned or generated by the program for those fields. For example, a struct might store a car's location, speed, and distance from other cars at a particular time.

Compound types can be extended to include new data or behavior, and their definitions can be passed to new compound types by **like** inheritance. For example, you can extend the predefined actor **car** to include new scenarios, or you can create a new actor **my_car** that inherits the scenarios of **car** and then add new scenarios.

Compound types can also be extended conditionally using **when** inheritance. With this feature, a type is extended only when a specified condition is **true**. For example, if an actor **my_car** has a field of type **driving_style**, the actor's attributes or behaviors can be different when the value of **driving style** is **aggressive** from when it is **normal**.

3.7.10. Predefined AV types

There are predefined actors and types that you can extend or constrain to facilitate the verification of your AV.

Predefined actors

An M-SDL environment contains several predefined actors.

The actor **top** contains instances of the following actors:

- **builtin** represents M-SDL's built-in scenarios.
- **av_sim_adapter** represents M-SDL's simulator interface.
- **map** represents the sets of paths traveled by actors.
- **traffic** represents cars, pedestrians and so on.
- **env** represents environmental systems and has scenarios such as weather and time of day.
- **dut** represents the AV system or device under test.

Under **traffic**, there is a list called **cars** of **car** actors.

Under **dut** there is

- A **dut.car** of type **car** represents the actual dut car (also called the ego)
- Possibly other actors, corresponding to various supervisory functions and so on.

Note: Because **map, env, traffic** and **dut** are instantiated as fields in **top,** you can access them directly as global actors, without reference to their place in the hierarchy, for example:

```
keep(x == env.time_of_day)
keep(car_in_position == map.reach(car1, point1))
```

You can extend any actor in this hierarchy to add actors. M-SDL can create actors before or during a run. Upon creation, an actor's fields are randomized according to the constraints you have specified and its built-in **start** scenario starts running in active mode. A **start** scenario can execute other scenarios, and these also run in active mode.

The scenario **top.main()** is called indirectly from **top.start()**. This scenario is initially empty and defines what the test does. Thus, to make your test run the **cut_in_and_slow** scenario, you can extend **top.main**:

```
extend top.main:
    do c: cut_in_and_slow()
```

Predefined env actor

The **env** actor is a global actor, and contains all environment-related activity. It has scenarios which change the environment like **weather** and **time_of_day,** for example:

```
weather(kind: rain, temperature: 12.5c, duration: 3min)
time_of_day(part: evening, start_time: 18hr)
```

The type **part** is morning, noon, evening, or night and **kind** is rain, snow, sunshine.

Example

```
scenario dut.cut_in_and_rain:
    do mix():
        cut_in()
        weather(rain)
        time_of_day(afternoon)
```

Predefined car actor fields

You can extend or constrain these fields of the **car** actor to match the allowed types of your simulator. You can also sample these fields and use them in coverage definitions.

avoid_collisions: bool	The default is **false**. When constrained to **true**, a car actor instance tries to avoid collisions.
category: car_category	The car_category type is defined as sedan, truck, bus, van, semi_trailer, trailer, four_wheel_drive.
color: car_color	The car_color type is defined as white, black, red, green, blue, yellow, brown, pink, grey.
driving_style	Relevant only for non DUT cars, the style type is defined as **aggressive, normal**.
length: distance	Car length.
max_speed: speed	Maximum speed. The default is 120kph.
model: string	Initially an empty string.
speed, **acceleration** and **road_position**	These fields hold the current speed, acceleration and road position of a car actor instance. **Note:** These fields are not generatable and must be assigned values explicitly during the run.
width: distance	Car width.

The **car** actor also has a scenario called **drive**. **drive** is a movement scenario, and it has a path parameter that represents the path to drive on. You can specify scenario modifiers inside a car's **drive** scenario.

Predefined enumerated types

Type name	Values
av_car_side	front, front_left, left, back_left, back, back_right, right, front_right, other
av_side	right, left
car_category	sedan, truck, bus, van, semi_trailer, trailer, four_wheel_drive
car_color	white, black, red, green, blue, yellow, brown, pink, grey
curvature	other, straightish, # [-1e-6..1e-6] soft_left, # (1e-6..1e-12] hard_left, # (1e-12..1e-18] soft_right, # [-1e-12..-1e-6) hard_right] # [-1e-18..-1e-12)
direction	other, straight, # (-20..20] degrees rightish, # (20..70] degrees right, # (70..110] degrees back_right, # (110..160] degrees backwards, # (160..200] degrees back_left, # (200..250] degrees left, # (250..290] degrees leftish] # (290..340] degrees
lane_use	none, car, pedestrian, cyclist
lane_type	none, driving, stop, shoulder, biking, sidewalk, border, restricted, parking, median, road_works, tram, entry, exit, offRamp, onRamp, rail, bidirectional
road_condition	paved, gravel, dirt

sign_type	speed_limit, stop_sign, yield, roundabout
time_of_day	midnight, sunrise, morning, noon, afternoon, sunset, evening
weather_kind	clear, cloudy, wet, wet_cloudy, soft_rain, mid_rain, hard_rain

3.8. M-SDL operators and special characters

M-SDL supports the use of the following operators in expressions.

Boolean comparison	==, !=, <=, >=	compares two expressions and returns a Boolean
Boolean compound	and, or, =>	join two simple Boolean expressions
Boolean negation	!, not	negate a Boolean expression
List indexing	[*n*]	reference an item in a list
Range	[*range*]	reference a range of values, any of the following or combination of the following: [i..] [i..j] [..j]
If then else	x ? y : z	select an expression
Arithmetic	+ - * / %	perform arithmetic operations
Qualified event yield operator	=>	create a field to facilitate coverage definition

These expression operators are distinct from special characters used in other contexts:

- The do-not-generate character ! is used only in field declarations to prevent pre-run generation of random values for the field. (A value is generated or assigned during the run instead.)
- The character combination => is used in a qualified event to create a field, thus facilitating coverage definition.
- The backslash character \ is used to continue a member over multiple lines or, within a string, to escape a character, for example “\t”.

- The plus character + is used with the line continuation character \ to concatenate a string continued over multiple lines.
- The dollar character $ is used for interpolation within string literals and is otherwise reserved for internal use. When it appears in this document it indicates either string interpolation, a shell environment variable or an internal M-SDL resource.

3.9. User task flow

The verification task flow that M-SDL supports is as follows:

1. Plan the verification project.

- Identify the top-level *scenario categories* that represent risk dimensions such as urban driving, highway driving, weather, sensor malfunction and so on.
- Identify the *scenario subcategories.* For example, lane-changes may be a subcategory of highway driving.
- Identify the *behaviors* in each scenario subcategory. For example, cutting-in-and-slowing-down would be a behavior in the lane changes subcategory.
- Identify the *coverage collection points* you can use to determine how thoroughly each scenario has been *covered* (exercised successfully). For example, the cutting-in-and-slowing-down behavior might have coverage points including road conditions, distance, and speed.
- identify the checking criteria (grading) used to judge how well the dut performed in various scenarios.
- Identify coverage goals for those behaviors and scenarios.

2. Create the verification environment.

- Describe the scenarios, behaviors and coverage points in M-SDL, basing them on lower-level built-in scenarios or available in a library.
- Identify the DUT and the execution platform.
- Identify any other additional tools you will use.

3. Automate test runs.

- Write tests, possibly mixing scenarios from different subcategories, such as cutting-in-and-slowing-down with conflicting-lane-changes.
- Launch multiple runs with different values for the scenario’s variables, such as road conditions, speed and visibility.

4. Analyze failures.

- Identify the cause of any checking error, such as collision or near collision.

- Fix the DUT or apply a temporary patch so that tests can continue.
- Rerun all failed runs automatically.

5. Track progress.

- Analyze the coverage data correlated with each goal specified in the verification plan to determine which scenarios have not been adequately tested.
- Write new tests to reach those corner cases.

3.10. Terminology

This section defines the terms used to describe M-SDL, M-SDL entities, and M-SDL tools.

Term	Definition
abstract scenario	A scenario whose constrainable fields are defined with a range of possible values.
bucket	A subrange of a range of possible values for a constrainable field defined to facilitate coverage analysis.
built-in	Data types, including enumerated types, actors, scenarios and structs that are predefined in an M-SDL tool.
concrete scenario	The result of solving a scenario while obeying all the constraints and randomizing where needed. A concrete scenario has a concrete value for every attribute.
constraint	A restriction on the possible values for a field or a movement made for the purpose of a specific test or run.
cover item	A constrainable field whose value you want to collect at specific times during a run.
cover group	A group of constrainable fields whose values are collected at the same time during a run.
coverage collection point	An abstract term for a cover item or cover group. A verification plan identifies coverage collection points or attributes whose values you want to collect.
coverage goal	The percentage of runs that need to execute successfully in order to declare that the behavior tested is safe.

coverage hole	An aspect of a behavior defined in a scenario for which no coverage data has been collected.
coverage metrics	The data collected that lets you determine whether the coverage goals have been met.
directed scenario	A scenario whose fields have been constrained to a narrower range of values or to a specific value.
dut	The device under test. For AVs, the dut is also known as the ego.
dut error	An unsafe behavior of the dut.
execution platform	The platform on which an M-SDL program executes, such as a simulator, possibly with hardware-in-the-loop, or even the AV on an actual track.
functional scenario	A scenario that evaluates the compliance of a system or component with specified functional requirements.
generation	A part of the planning process that creates the data structure and assigns values to fields according to the specifications provided in type declarations and constraints. Fully random generation assigns values to fields depending on their defined data type (the legal values). Constrained random generation assigns values within the restrictions defined in constraints.
grading/ checking	The process of determining how well the dut performed in a particular run or set of run according to some performance criteria such as safety, comfort and so on.
library	A set of predefined data types, including enumerated types, actors, scenarios and structs that is available separately from an M-SDL tool.
multi-test	An M-SDL tool's facility for creating multiple test files from a single test and the value tuples of its constrainable fields.
nested scenario	A scenario invoked from within another scenario.
parallel	A set of actions in a scenario that are executed in parallel.
planning	An M-SDL tool's process of creating a program tree and determining the sequencing of actions.

program tree	A hierarchical instance tree created during planning when a scenario is invoked from the predefined, top-level scenario in an M-SDL tool. An instance of that scenario and its members, including nested scenarios and their members, recursively.
raw metrics	Coverage metrics that are not mapped to a verification plan.
scenario	A description of the attributes and behavior of one or more actors.
scenario failure	An error indicating that a run did not meet the scenario modifiers set in a scenario.
seed	A numeric value that is used to initiate generation. Using the same seed for multiple runs results in the same generated values. This behavior is useful when you want to re-execute a run. Using different seeds results in different generated values.
serial	A set of actions in a scenario that are executed in sequence.
test	The description of what to run, including the top-level scenario. A test and a seed together determine a run.
test file	An M-SDL file containing the definition of a test containing the specification of an execution platform, a map, the top-level scenario and possibly constraints on the attributes or movement described in the scenario.
test suite	A set of tests designed to verify a particular behavior and attributes or a set of those.
regression	A set of tests designed to ensure that previously defined and tested behavior still performs after a change.
run	A single execution of a test.
run group	Multiple executions of a test with varied constraints on the behavior and attributes of a scenario.
vplan	A verification plan that defines the behavior to be tested, the coverage collection points and the coverage goals.

4. Statements

Summary: This topic describes M-SDL statements.

4.1. actor

Purpose	Declare an active object with activities (behaviors)
Category	Statement
Syntax	`actor actor-name [:` `  member+]`
Syntax parameters	*actor-name* must be different from any other defined actor, type, or struct name because the namespace for these constructs is global.
	member+ is a list of one or more of the following: • field declarations • **keep** constraints • **cover** definitions • events • **when** subtypes • external method declarations Each member must be on a separate line and indented consistently from **actor**.

Like structs, actors are compound data structures that you create to store various types of related data. In contrast to structs, actors also are associated with scenarios. Thus, an actor is a collection of both related data and declared activity.

Each actor has by default a scenario called **start()**. Since this scenario starts automatically at the beginning of a run, you can activate a scenario or monitor a scenario for coverage by adding it to any actor's **start()** scenario. However, as a general rule, if you want to activate or monitor a scenario for the purposes of a particular test run, it is recommended that you do so by extending the main scenario of the top-level actor, **top.main**.

All actors have a **lifetime()** scenario that runs throughout the life of an actor instance. You can extend this scenario to look for events that might occur over the lifetime of an actor. For example:

```
extend dut.lifetime:
    event near_collision is \
            (col.distance < 3meter) @dut.near_collision =>col
    cover(col.distance, event: near_collision, unit: centimeter)
    cover(col.other_car, event: near_collision)
```

Actors support **when** subtypes, as well as extension by **extend**. They do not support **like** inheritance.

Example when subtype

To maintain composability, it is recommended to declare **when** subtypes using the **extend** construct.

The **when()** expression specifies that when a field in **car** of type **category** has the value **semi_trailer**, the field **max_speed** must be equal to 50 kph. This limit does not apply when the field **category** has a different value, for example, **sedan**.

```
extend car:
    when(category: semi_trailer)
        keep(max_speed == 50kph)
```

4.2. enumerated type

Purpose	Define a scalar type with named values
Category	Statement

Syntax	`type type-name : [member*]`
Syntax parameters	*type-name* must be different from any other defined actor, struct or type name because the namespace for these constructs is global.

	*member** is a comma-separated list of zero or more enumerations on one line in the form:
	`enum-name[=exp]`
	Each *enum-name* must be unique within the type.
	exp evaluates to a constant value. If no *exp* is specified, the first name in the list is assigned a value of 0, the next 1 and so on.
	You can declare a type without defining any values using the following syntax:
	`type type-name : []`

Example

In this example, the **type** statements define the types of vehicles in a verification environment and the driving style.

```
type car_type: [sedan = 1, truck = 2, bus = 3]

# by default, aggressive = 0, normal = 1
type driving_style: [aggressive, normal]
```

4.3. extend

Purpose	Add to an existing type or subtype of a struct, actor or scenario

Category	Statement
Syntax	`extend type:` `    member+`
Syntax parameters	*type* is the name of a previously defined struct, actor or scenario.
	member+ is a list of one or more new members of the type. For enumerated types, the list is comma-separated, enclosed in square brackets, and can be placed on one line. For compound types, each member must be placed on a separate line and indented consistently from **extend**.

Example extend enumerated type

Because two values are already defined for this type (aggressive=0, normal=1), erratic has the value of 2, unless explicitly assigned a different value.

```
extend driving_style: [erratic]
```

Example extend struct

This example extends the struct named **storm_data** with a field for wind velocity. This extension applies to all instances of **storm_data**.

```
extend storm_data:
    !wind_velocity: speed
```

Example extend scenario

When **car.bar** is extended, the scenario instances are performed in sequence: f1, f2, f3.

```
scenario car.foo:
    x: int with:
        keep(default it == 0)
    y: int with:
        keep(it != x)

scenario car.bar:
    do serial():
        f1: foo(5)
        f2: foo

extend car.bar:
    do f3: foo(y: 10)
```

4.4. import

Purpose	Load an M-SDL file into the M-SDL environment
Category	Statement
Syntax	`import file-name`
Syntax parameters	*file-name* is the name of the file to import. The default file extension is **.sdl**. Wildcards are not allowed.

If you have built a scenario hierarchically, with a higher level scenario calling a lower-level one, and the coverage definitions in a separate file, you can use **import** statements to import all these files into a test file.

Imports must come before any other statements in an SDL file. The order of imports is important, as a type must be defined before it is referenced.

If a specified file has already been loaded, the statement is ignored. For files not already loaded, the search sequence is:

1. If a full path is specified (for example, **/x/y/my_file**), then use that full path.
2. Else look in the current directory.
3. Else look in directories specified by the MSDL_PATH environment variable.
4. Else look in the directory in which the importing file resides.

5. Else this is an error.

Example

This example shows the **import** statements from a typical test. The first import line is the simulator configuration file, and the second loads **cut_in_and_slow_top.sdl**, the top-level SDL source file for the test.

```
import sumo_config
import cut_in_and_slow_top
```

The **cut_in_and_slow_top.sdl** file in turn has the following **import** statements:

```
import cut_in_and_slow
import cut_in_and_slow_cover
import cut_in_and_slow_planned_cover
```

4.5. modifier

Purpose	Declare a modifier for scenarios
Category	Statement
Syntax	**modifier** *name*: *member+*
Syntax parameters	*name* is in the form *actor-name.modifier-name.* The *modifier-name* must be unique among that actor's scenarios and scenario modifiers, but it can be the same as the scenario or scenario modifier of a different actor.

member+ is a list of one or more of the following:

- Field declarations
- **keep** constraints
- **cover** definitions
- Event declarations
- **when** subtypes
- External method declarations
- Scenario modifier invocations

  Each member must be on a separate line and indented consistently from **modifier**.

Scenario modifiers constrain or modify the behavior of a scenario; they do not define the primary behavior of a scenario.

Scenario modifiers cannot include scenario invocations or **do**.

Example 1

```
modifier car.speed_and_lane:
    s: speed
    l: lane
    keep(speed > 10kph)
    speed(s)
    lane(1)
```

Example 2

```
modifier map.curving_multi_lane_highway:
    p: pathname
    lanes: uint

    keep(lanes > 2)
    path_min_driving_lanes(p, lanes)
    path_curve(p, max_radius:11m, min_radius:6m, side: left)
```

4.6. scenario

Purpose	Declare an ordered set of behaviors by an actor
Category	Statement
Syntax	`scenario name [:` `  member+]`
Syntax parameters	*name* is in the form *actor-name.scenario-name.* The scenario name must be unique among that actor's scenarios, but it can be the same as the scenario of a different actor.
	member+ is a list of one or more of the following: • field declarations • **keep** constraints • **cover** definitions • events • **when** subtypes • external method declarations • A scenario invocation using **do** • A scenario modifier invocation Each member must be on a separate line and indented consistently from **scenario**. **Note**: Every scenario has a predefined **duration** field of type **time**.

The M-SDL built-in scenarios perform tasks common to all scenarios, such as implementing serial or parallel execution mode or implementing time-related actions such as **wait**. Library scenarios describe relatively complex behavior, such as the **car.drive** scenario, and scenario

modifiers, that let you control speed, distance between other vehicles and so on. By calling these scenarios, you can describe more complex behavior, such as a vehicle approaching a yield sign or another vehicle moving at a specified speed. In this manner, complex behavior is described by a hierarchy of scenario invocations. For further complexity, multiple scenarios can be mixed. For example, a weather scenario can be mixed with a car scenario.

Two common mode of execution are serial and parallel. (See Operator Scenarios (page 69) for a description of other modes.) In either case, a scenario can have any number of members. In serial execution mode, each member starts when its predecessor ends. In parallel execution mode, all members within each phase start concurrently. The next phase starts when the last member of the preceding phase ends. In either execution mode, the scenario ends when the last member (of the last phase) ends. Failure of any member causes the scenario to fail.

You can control the behavior of scenarios and collect data about their execution by declaring data fields and other members in the scenario itself or in its related actor or structs. For example, declaring a field of type **speed** in a car actor allows control over the speed of a car.

When you declare or extend a scenario, you must associate it with a specific actor by prefixing the scenario name with the actor name in the form *actor-name**.**scenario-name*.

Note: Invoking the generic form of the scenario (*actor.scenario*) is not allowed.

See Example scenarios (page 4) for examples of how to create and reuse scenarios.

4.7. struct

Purpose	Define a compound data structure
Category	Statement
Syntax	**struct** *type-name* [**like** *base-struct-type*] [**:** *member+*]
Syntax parameters	*type-name* must be different from any other defined type, struct, or actor name because the namespace for these constructs is global.
	base-struct-type is the name of a previously defined struct whose members you want this struct type to inherit.

member+ is a list of one or more of the following:

- field declarations
- **keep** constraints
- **cover** definitions
- events
- **when** subtypes
- external method declarations

  Each member must be on a separate line and indented consistently from **struct**.

Structs are compound data structures that you can create to store related data of various types. For example, the AV library has a struct called **car_collision** that stores data about the vehicles involved in a collision.

Structs support both **like** inheritance and **when** subtypes. Both these mechanisms let you easily create variations of a base struct type. For example, you can create a new struct subtype based on **car_collision** that contains additional data of interest to your DUT. When you create a new struct subtype, the original base type is not modified. In contrast to both these mechanisms, if you **extend** a struct type, the original base type is modified accordingly.

In general, **like** inheritance is recommended unless you want to choose one of several children at generation or at runtime.

Example struct declaration

This example defines a struct named **my_car_status** with do-not-generate fields called **time** and **current_speed**.

```
struct my_car_status
    !time: time
    !current_speed: speed
```

*Example struct **like** inheritance*

This example adds a field to store the type of the other car involved in the collision to the base type struct **collision_data**.

```
struct my_collision_data like collision_data:
    !other_car_category: car_category
```

*Example struct **when** inheritance*

This example adds a field for **snow_depth** when the storm is a snowstorm. To add specific features to **storm_data** only when **snow** use: **when (storm: snow).**

```
type storm_type: [rain, ice, snow]
type snow_depth: int

struct storm_data:
    storm: storm_type

    when(storm: snow):
        depth: snow_depth
```

5. Struct, actor or scenario members

Summary: This topic describes members that can be defined within structs, actors or scenarios.

5.1. cover()

Purpose	Define a coverage data collection point
Category	Struct, actor, or scenario member
Syntax	`cover( exp, [ param* ])`
Syntax parameters	*exp* evaluates to the name of a field in the enclosing construct. The field must be a scalar type. The value of the cover item is the value of the field when the cover group event occurs.

*param** is a comma-separated list of zero or more of the following:

- **unit:** *unit* specifies a unit for a physical quantity such as time, distance, speed. The field's value is converted into the specified unit, and that value is used as the coverage value.
  Note: You must specify a unit for cover items that have a physical type.
- **range:** *range* specifies a range of values for the physical quantity in the unit specified with **unit**.
- **every:** *value* specifies when to slice the range into subranges. If the range is large, for example [0..200], you might want to slice that range into subranges every 10 or 20 units.
- **event:** *event-name* specifies the event when the field is sampled. The default is the **end** event of the scenario. Items that have the same sampling event are aggregated into a group.
- **text:** *string* displays explanatory text, enclosed in double quotes, about this collection point.
- **ignore:** *item-bool-exp* defines values that are to be completely ignored. They do not appear in the statistics at all. The expression is a Boolean expression that can contain only a coverage item name and constants.

  If the **ignore** expression is **true** when the data is sampled, the sampled value is ignored (not added to the bucket count).

Coverage is a mechanism for sampling key parameters related to scenario execution. Analyzing aggregate coverage helps determine how safely the AV behaved and what level of confidence you can assign to the results.

For example, to determine the conditions under which a **cut_in_and_slow_down** scenario failed or succeeded, you might need to measure:

- The speed of the dut.car
- The relative speed of the passing car
- The distance between the two cars

You can specify when to sample these items. For example, the key events for this scenario are the **start** and **end** events of the **change_lane** phase.

Cover items that have the same sampling event are aggregated into a single cover group. The default event for collection coverage is **end**.

Note: If the cover item is a physical type, time, speed, distance and so forth, you must specify a unit.

If you want, you can specify a range of data that you want to collect. If the range is large, you might want to slice that range into subranges or *buckets*. For example, if you expect the dut car to travel at a speed between 10 kph and 130 kph, specifying a bucket size of 10 gives you 12 buckets, with 10 kph – 19 kph as the first bucket.

You can also specify an explanatory line of text to display about the cover item during coverage analysis.

This example defines a line of display text, a unit of measurement, a range and a range slice for the field **speed**:

```
cover(speed, unit: kph,
    text: "Absolute speed of ego at change_lane start (in km/h)",
    range: [10..130], every: 10)
```

You can declare a field and define coverage for it at the same time. The following examples are equivalent.

```
    # Example 1
    current_speed: speed with:
        cover(it, unit: kph)

    # Example 2
    current_speed: speed
    cover(current_speed, unit: kph)
```

Example field and cover declaration

The following example extends the **dut.cut_in_and_slow** scenario to add a do-not-generate pseudo-variable called **rel_d_slow_end**. It assigns that variable the value returned by the **map.abs_distance_between_locations()** method at the **end** event of the **slow** phase of the scenario. It then defines coverage for that field, including a unit, display text and so on.

```
extend dut.cut_in_and_slow:
    !rel_d_slow_end:= map.abs_distance_between_locations(
      dut.car.location, car1.location) @slow.end with:
        cover(it, text: "How ahead is car1 relative to dut" +\
                                  "at slow end (in centimeter)",
            unit: centimeter, range: [0..6000], every: 50,
            ignore: (rel_d_slow_end < 0 or rel_d_slow_end > 6000))
```

Combining coverage from different items

You can combine the coverage of two or more items by specifying the items as a list. This coverage, sometimes called *cross coverage*, creates a Cartesian product of the two cover vectors, showing every combination of values of the first and second items, every combination of the third item and the first item, and so on.

Example 1

The following example creates a Cartesian product of three cover vectors at the start of the **change_lane** phase of a scenario: the relative distance between two cars, the absolute velocity of the DUT vehicle, and the relative speed of the other vehicle.

```
cover([rel_d_change_lane_start,
    dut_v_change_lane_start,
      rel_v_change_lane_start]
    text: "Cross coverage of relative distance," + \
      "dut's absolute velocity and" + \
      " relative speed of other vehicle",
    unit: kph)
```

Example 2

You can only cross cover items that have the same sampling event. To overcome this limitation, define a secondary cover point with the common sampling event. For example, if you want to include a field **car1.speed** in a cross with other items that are sampled at the start of a scenario, you have to define a second field with that sampling event and cover the second field. The reason is that the default sampling event for **car1.speed** is **end**, not **start**.

```
!speed1:= car1.speed @start with:
    cover(speed1)
```

5.2. event

Purpose	Signify a point in time.
Category	Struct, actor, or scenario member
Syntax	`event event-name [(param+)] [ is qualified-event ]`
Syntax parameters	*event-name* is a name unique in the enclosing construct.
	param+ is comma-separated list of one or more fields in the form *field-name*: *type-name*.
	qualified-event has the format [*bool-exp*][@*event-path* [=>*name*]]. If *event-path* is missing, the expression is checked on **top.clk** (every callback from the simulator). If *bool-exp* is missing, **true** is assumed. At least one of *event-path* and *bool-exp* must be specified. If specified, the =>*name* clause creates a pseudo-variable with that name in the current scenario – the *event object variable*. The variable is used to access the event fields, which is useful for collecting coverage over their values.

Events are transient objects that represent a point in time and can trigger actions defined in scenarios. You can define an event within a struct, but more typically within an actor or scenario.

Scenarios can emit events. Events are used to:

- Cause scenarios waiting for that event to proceed.
- Assign a sampled value to a field when that event occurs.
- Collect coverage data when that event occurs.

Every scenario, and every phase of every scenario has three predefined events, **start, end** and **fail**. The **fail** event ends the scenario without emitting its **end** event or collecting coverage.

An event declaration can include an event expression (using “is”). Event expressions have two parts:

- A Boolean expression.
- An event path (a path expression that evaluates to another event in the program tree).

An event expression evaluates to **true** and the event is emitted if the Boolean expression is **true** when the event specified by the event path occurs. If no event path is specified, the default event is every callback from the simulator (**top.clk**). If no Boolean expression is specified, the default is **true**.

You can declare events without an event expression. However, the event does not occur unless it is raised by an **emit** action. For example the **arrived** event declared here must be emitted explicitly:

```
event arrived
```

If you define an event with an event path leading to another event but no Boolean expression, the event occurs when the event specified by the path occurs. This type of event is called a bound event. In this example, **car_arrived** occurs when the **arrived** event in **main_car** occurs.

```
event arrived is @main_car.arrived
```

If you define an event with a Boolean expression but no event path, the expression is checked every time there is a callback from the simulator. The following event occurs when **snow_depth** is greater than 15 centimeters arrives at the specified location.

```
event deep_snow is (snow_depth > 15cm)
```

Events can have parameters. Event parameters are accessible by referencing the pseudo-variable declared by => *name*, in the scope of the enclosing object. For example, an event **near_collision** with parameters is defined in the **dut** actor with two parameters, **other_car** and **distance**:

```
extend dut:
    event near_collision(other_car: car, distance: distance)
```

This event can be emitted by the **on** modifier, which is looking for near collisions:

```
on (map.abs_distance_between_locations(dut.car.location,
      car2.location) < 2m):
      emit dut.near_collision(car2, ...)
```

In the **lifetime** scenario of the **dut**, the local **near_collision** is bound to the actor's **near_collision** event and creates a pseudo variable **col** to facilitate coverage collection:

```
extend dut.lifetime:
    event near_collision is \
            (col.distance < 3meter) @dut.near_collision =>col
    cover(col.distance, event: near_collision, unit: centimeter)
    cover(col.other_car, event: near_collision)
```

5.3. external method declaration

Purpose	Declare a procedure written in a foreign language
Category	Scenario member
Syntax	**def** *method-name* **(***param**) [**:** *return-type*] **is** \ [[*extend*] **external** "*binding-string*" | **empty** | **undefined**]
Syntax parameters	*method-name* is a unique name in the current context.
	*param** is a list composed of zero or more arguments of the form *param-name: param-type* [=*default-exp*] separated by commas. The parentheses are required even if the parameter list is empty.
	return-type specifies the type of the return parameter.

	extend specifies an extension to a previously declared method: • **also** appends the specified method to the previously declared method. • **first** prepends the specified method to the previously declared method. • **only** replaces the previously declared method with the specified method.
	binding-string (in double quotes) specifies the language binding. The syntax of this binding is specific to each language.
	empty or **undefined** let you declare a method without specifying any actions. **undefined** causes an error if the method is called.

These methods execute immediately when called in zero simulated time.

Example

```
def check_errors() is also external "python method check_dut_internals"
```

5.4. field

Purpose	Declare a field to contain data
Category	Object member
Syntax	`[!]field-name:[ type ][= sample][ with-block ]`
Parameters	The do-not-generate operator (!) specifies that no value should be generated for this field before the run executes. Instead, a value is generated or assigned during the run.
	field-name is a unique name within the enclosing struct, actor or scenario.

type is required unless *sample* is present. *type* is any data type or a list of any of these. Use **list of** *type* for lists.

When *sample* is specified, omit *type*, so that *type* is derived from the type of the sampling expression.

sample is in the form:

```
sample( exp, qualified-event)
```

where *exp* is an expression that evaluates to the field whose value you want to sample and *qualified-event* is in the form:

```
[ bool-exp ][ @event-path [=>name]]
```

If *bool-exp* is missing, **true** is assumed. If *event-path* is missing, the expression is checked on **top.clk** (every callback from the simulator). At least one of *event-path* and *bool-exp* must be specified.

If specified, the **=>***name* clause creates a pseudo-variable with that name in the current scenario (the *event object variable*). The variable is used to access the event fields, which is useful for collecting coverage over their values.

with-block is in the form:

```
with [(subtype)]:

    modifier+
```

And *subtype* is in the form:

```
field: value [, field: value,...]
```

See below for the syntax options for **with**.

For struct or actor fields, *member* is a constraint in the form [**soft**] *constraint-bool-exp.* Within the constraint expression, use the implicit variable **it** to refer to the field if it is scalar, or [**it**].*field-name* if **it** is a struct or actor field and *field-name* is the field that you want to constrain.

For scenario fields, *member* is either:

- A constraint in the form *constraint-strength constraint-bool-exp* where *constraint-strength* is either **soft** or **default**. Within the constraint expression, use **it** to refer to the field if it is scalar, or [**it**].*field-name* if **it** is a struct or actor field and *field-name* is the field that you want to constrain.
- A coverage definition in the form **cover it**, where **it** is a scalar field, or **cover** [**it**]**.***field-name*, where **it** is a struct or actor field and *field-name* is the field that you want to cover.
- A scenario modifier.

The **with** block has four variants:

option 1: members listed as a block

```
with:
    member+
```

option 2: members passed as parameters

```
with(param+)
```

option 3: members passed as parameters and listed as a block

```
with(param+):
    member+
```

option 4: optional parameters and members listed on one line

```
with[(param+)]: member1 [; member2;...]
```

Example scalar field declarations

The following example shows an actor with two fields of type **speed**. The first field is named **speed** and holds a value specifying the current speed. The name **speed** is preceded by the do-not-generate operator, which prevents it from receiving a value during pre-run generation. Most likely it is assigned various values while a scenario is running. The second field is named **max_speed** and it has a soft constraint. It receives a value during pre-run generation of 120 kph, unless it is constrained or assigned otherwise.

```
actor my_car:
    # Current car speed
    !speed: speed

    # Car max_speed
    max_speed: speed with:
        keep(soft it == 120kph)
```

Example list field

```
extend traffic:
    cars: list of car
```

Example list field with constraints

```
actor my_car_convoy:
    first_car: car
        cars: list of car with:
        # the list will have between 2 and 10 items
        # the first item is first_car
        keep(soft cars.size() <= 10)    # it.size() can also be used
        keep(soft cars.size() >=  2)    # it.size() can also be used
        keep(cars[0] == first_car)
```

Example string field

The default value of a field of type string is **null**. This is equivalent to an empty string,“”.

```
struct data:
    name: string with:
        keep(it == "John Smith")
```

Example actor or struct field declarations

The following example shows a field of type **my_car** with the name **car1** instantiated in the **traffic** actor. The constraint on **car1**’s **max_speed** field overrides the earlier **soft** constraint on the same field.

```
extend traffic:
    car1: my_car with(max_speed: 60kph)
```

Example scenario field

This example shows a struct field **storm_data** instantiated in a scenario called **env.snowstorm**. Constraints are set on **storm_data**’s two fields, and the **wind_velocity** field is monitored for coverage.

```
type storm_type: [rain, ice, snow]

struct storm_data:
    storm: storm_type
    wind_velocity: speed

scenario env.snowstorm:
    storm_data with:
        keep(it.storm == snow)
        keep(soft it.wind_velocity >= 30kph)
        cover(it.wind_velocity, unit=kph)
```

Example sampling

This code samples the value of car1.speed at the **end** event of the get_ahead phase of the cut_in scenario.

```
extend dut.cut_in:
    !speed_car1_get_ahead_end := sample(car1.speed, @get_ahead.end)
    cover(it text: "Speed of car1 at get_ahead end (in kph)", unit: kph,
        range: [10..130], every: 10)
```

5.5. keep()

Purpose	Define a constraint
Category	Struct, actor, or scenario member
Syntax	**keep(** [*keyword*] *constraint-boolean-exp***)**

Syntax parameters

keyword is either **soft** or **default**.

soft suggests default values for a field. Soft constraints are order dependent and are not met if they conflict with hard constraints or with soft constraints that have already been applied. The **soft** keyword cannot be used in compound Boolean expressions.

default specifies default values for a field. A default constraint is ignored if it conflicts with hard constraints.

If no keyword is specified, the constraint is a hard constraint. The constraint is either met or a constraint contradiction is issued.

constraint-boolean-exp is a simple or compound Boolean expression that returns either **true** or **false** when evaluated at runtime.

The following operators can be used in *bool-exp*:

Boolean comparison	==, !=, <, <=, >, >=	compare two values
Boolean range or list	**in [***range* | *list***]**	constrain a scalar to a range of values or a list to be a subset of another list
Boolean negation	!, not	negate a Boolean expression

The order of precedence for compound Boolean operators is: **and**, **or**, **=>**. A compound expression containing multiple Boolean operators of equal precedence is evaluated from left to right, unless parentheses(()) are used to indicate expressions of higher precedence.

There are also Boolean constants and path expressions leading to Boolean variables. Additionally, Boolean expressions can include calls to external methods returning Boolean values.

The part of the planning process that creates data structures and assigns values to fields is called *generation*. This process follows the specifications you provide in **type** declarations, in field declarations, and in **keep** statements. Those specifications are called *constraints*.

The degree to which generated values for a field are random depends on the constraints that are specified. A field's values can be either:

- Fully random— without explicit constraints, for example:

```
    tolerance: int
```

- Fully directed— with constraints that specify a single value, for example:

```
keep(my_speed == 50kph) # my_speed is set to 50 kph
```

- Constrained random— with constraints that specify a range of possible values, for example:

```
keep(my_speed in [30..80]kph) # my_speed is restricted to a range
```

Simple Boolean constraints

You can add **keep()** constraints to fields inside structs, scenarios, or actors, for example:

```
my_speed: speed with:
    keep(my_speed in [30..80]kph)
```

Compound Boolean constraints

Compound Boolean constraints define relationships between two or more fields. For example, if an object has thee fields:

- **legal_speed:** the legal speed allowed in that road
- **lawful_driver:** a driver who follows the laws
- **current speed**: the current speed of the vehicle driven by lawful driver

You can define the current speed in relation to the other fields, so that a lawful driver implies the current speed is less than or equal to the legal speed:

```
keep(lawful_driver => (current_speed <= legal_speed))
```

Examples

```
    # both constraint expressions must evaluate to true
    keep(x <= 3 and x > y)

    # at least one constraint expression must evaluate to true
    keep(x <= 3 or x > y)

    # if the first expression evaluates to true, the second one must
    # also evaluate to true
    keep((x <= 3) => (x > y))
```

The order of precedence for compound Boolean operators is: **and**, **or**, **=>**. A compound expression containing multiple Boolean operators of equal precedence is evaluated from left to right, unless parentheses(()) are used to indicate expressions of higher precedence.

List constraints

You can use the list method **.size()** and list indexing (*list*[*index*]) in list constraint expressions.

Example

The constraints specify that the list size is between 2 and 10, inclusive. The third constraint in this example specifies that the first car in the list must be the object **first_car**.

```
actor car_convoy:
    first_car: car
    cars: list of car with:
        keep(soft cars.size() <= 10)
        keep(soft cars.size() >= 2)
        keep(cars[0] == first_car)    # list indexing
```

Relative strength of keep constraints

You can define the strength of **keep** constraints:

- A **hard** constraint must be obeyed when the item is generated. If a hard constraint conflicts with another hard constraint, a contradiction error is issued. For example, if the following constraint is applied and the generator cannot assign the value 25 to the field **current_speed**, an error is issued:

```
current_speed: speed with:
    keep(it == 25)
```

- A **soft** constraint must be obeyed unless it contradicts a hard constraint, or a later-

specified soft constraint. However, soft constraints are ignored without issuing an error. For example, if the following constraint is applied and the generator assigns the value green to the field color, no error is issued:

```
color: car_color with:
    keep(soft it!= green)
```

- A **default** constraint must be obeyed unless another hard constraint directly on that object specifies a different value. For example, the following specifies a default constraint:

```
x: int with:
    keep(default it == 0
```

Note: You can apply default constraints only to fields within scenarios, not within actors or structs.

Soft constraints and default constraints

Below is an example of the difference between soft constraints and default constraints.

```
scenario car.foo:
    x: int with:
        keep(default it == 0)
    y: int with:
        keep(it!= x)
        keep(x == 5)      # overwrite the default: x will be 5
```

Default constraints seem like soft constraints. However, default constraints are only overwritten by a hard constraint directly on that object. The following code causes a contradiction:

```
extend car.foo:
    keep(y == 0)   # causes a contradiction
```

The contradiction occurs because **x** is constrained to 0 via a default constraint, and there is no hard constraint on **x** to change the default value. There are two hard constraints on **y**, one setting its value to 0 and the other requiring its value to be different from **x**. There is no value for **y** that satisfies both constraints.

On the other hand, if **x** is constrained to 0 via a soft constraint instead of a default constraint, then **x** gets a non-0 value and no contradiction occurs.

5.6. when subtype

Purpose	Create a struct, actor or scenario subtype
Category	Struct, actor or scenario member
Syntax	`when(param+):` `    member+`
Syntax parameters	*param+* is a comma-separated list of one or more items in the form *field-name*: *value* where • *field-name* is the name of a field in the base type. Only Boolean or enumerated type fields can be used. • *value* is one of the legal values for the field referred to by *field-name*. Ranges are not allowed. You can pass more than one parameter if you want to create a subtype based on more than one field of the base struct type.
	member+ is a list of one or more new members of the subtype. You can add any struct member, or (in a scenario) any scenario member.

All objects, structs, actors and scenarios, support **when** subtypes. When subtypes let you:

- Explicitly reference a field that determines the subtype
- Create multiple, orthogonal subtypes
- Use random generation to generate lists of objects with varying subtypes

Example struct subtype

This example adds a field for **snow_depth** when the storm is a snowstorm.

```
type storm_type: [rain, ice, snow]

type snow_depth: distance with:
    keep(it in [0..15]cm)

struct storm_data:
    storm: storm_type

    when(storm_type: snow):
        depth: snow_depth
```

Example scenario subtype

If the following subtypes are defined in a scenario:

```
scenario env.bad_weather:
    kind: weather_kind
    when(kind: rain):
        strength: real [0.0..1.0]
    when(kind: snow):
        strength: real [0.0..1.0]
```

You can then activate the scenario subtype with:

```
do bad_weather(kind: rain, strength: 0.4)
```

It is clear here that **strength** refers to the field defined under **kind: rain** and not to the one defined under **kind: snow**.

Example defining a field as subtype

In field definitions, you can use the **with()** syntax to define a when subtype, and also to refer to fields inside that **when** subtype. For example:

```
d: storm_data with(storm: snow, depth: 10cm)
```

Example using when subtype in a path

You can specify **when()** in a path to condition it on a **when** subtype. For example:

```
in x.y.c.when(category: truck).z with:
    # The following applies only if c is a truck
    speed([45..60]kph)
```

6. Scenario members

Summary: This topic describes members that can belong only to scenarios, not to actors or structs.

6.1. Scenario modifier invocation

You can invoke scenario modifiers when invoking a scenario as well as from some operator scenarios.

To invoke one or more modifiers, define a **with** block in one of these constructs.

Note: **with** blocks can also contain **keep()** constraints and **cover()** definitions.

Example scenario invocation:

```
car1.drive(path, adjust: true) with:
    position(distance: [5..100]m, behind: dut.car, at: start)
    position(distance: [5..15]m, ahead_of: dut.car, at: end)
```

Example operator scenario

```
first_of:
    intercept_1()
    intercept_2()
with:
    position(distance: [5..100]m, behind: dut.car, at: start)
    position(distance: [5..15]m, ahead_of: dut.car, at: end)
```

The **with** block has four variants:

option 1: members listed as a block

```
with:
    member+
```

option 2: members passed as parameters

```
with(param+)
```

option 3: members passed as parameters and listed as a block

```
with(param+):
    member+
```

option 4: optional parameters and members listed on one line

```
with[(param+)]: member1 [; member2;...]
```

6.2. do (behavior definition)

Purpose	Define the behavior of a scenario
Category	Scenario member
Syntax	`do scenario-invocation`

do defines scenario behavior by composing the behavior out of the specified built-in and user-defined scenarios. Invoking a scenario causes its **do** members to be invoked.

Within a scenario declaration or extension, use **do** only for the top-level scenario. Omit **do** for an embedded (nested) scenario. For example **do** is used below for **serial**, but omitted for **turn** and **yield**:

```
scenario car.zip:
    p: path
    do serial():
        t: turn()
        y: yield()
```

To execute scenario **zip**, you need to extend **top.main**, again with **do**:

```
extend top.main:
    set_map(my map)
    do z: car.zip()
```

If you extend scenario **zip**, you use **do** again:

```
extend car.zip:
    do intercept()
```

These multiple **do** clauses are allowed, and execute in sequence. In this example, the sequence is **turn**, **yield**, and **intercept**.

It is recommended to add at least one explicit label at the top-level scenario invocation. In the following example, there is an explicit label for the **serial** invocation at the root of the tree. The remaining invocations are labelled implicitly. If there is ambiguity (more than one object with the same name) a count is added at the end of the label in the form !*n*.

```
extend top.main:
    do a: serial():                # explicit label "a"
        car1.drive(path) with:     # a!car1
            speed(0kph, at: start) # a!car1!speed
            speed(10kph, at: end)  # a!car1!speed!2
```

If there is no explicit label at the top level, the implicit label is the scenario name. Without the explicit label in the example above, the implicit label for **car1.drive** is **serial!car1**.

7. Scenario invocation

Summary: This topic describes how to invoke a scenario.

Purpose	Invoke a scenario
Category	Scenario invocation
Syntax	`[label-name:] scenario-name(param*) [with-block]`
Syntax parameters	*label-name* is an identifier that has to be unique within the scenario declaration.
	scenario-name is the name of the scenario you want to invoke, optionally including the path to the scenario. See Actor hierarchy and name resolution (page 17) for an explanation of how names without explicit paths are resolved. **Note:** Invoking the generic form of the scenario (*actor-type.scenario*) is not allowed.
	*param** is a list of zero or more field constraints, enclosed by parentheses. The list can be name-based (*field-name*: *value, …*) or order-based (*value, …*) where the first value is assigned to the first field in the scenario, and so on. In the list, a name-based parameter can follow an order-based parameter, but not vice-versa. Thus, **turn(x:3, y: 5)**, **turn(3, y: 5)**, and **turn(3, 5)** are legal and assign the same values, but **turn(x:3, 5)** is not allowed. *value* can be a single value or a range. A unit is required if the type is physical.

with-block contains one or more of the following:

- **keep()** constraint
- **cover()** definition
- scenario modifier

Note: If you add a **cover()** definition to a scenario invocation, the coverage data is added to the invoking scenario, not to the invoked scenario.

The **with** block has four variants:

option 1: members listed as a block

```
with:
    member+
```

option 2: members passed as parameters

```
with(param+)
```

option 3: members passed as parameters and listed as a block

```
with(param+):
    member+
```

option 4: optional parameters and members listed on one line

```
with[(param+)]: member1 [; member2;...]
```

Example 1

This example shows the declaration of a scenario called **traffic.two_phases**. It invokes the **serial** operator scenario with **do**. (At this level, this is the top-level scenario.) The lower-level scenario, **car1.drive** is invoked without do.

```
scenario traffic.two_phases:   # Scenario name
    # Define the cars with specific attributes
    car1: car with:
        keep(it.color == green)
        keep(it.category == truck)

    path: path

    # Define the behavior
    do serial():
        phase1: car1.drive(path) with:
            spd1: speed(0kph, at: start)
            spd2: speed(10kph, at: end)
        phase2: car1.drive(path) with:
            speed([10..15]kph)
```

Example 2

```
# Same as adding "cover turn.side"
# adds coverage to the current scenario, not to turn's coverage
do car1.turn() with:
    cover(it.side)
```

8. Operator scenarios

Summary: This topic describes built-in scenarios that invoke lower-level scenarios that serve as operands.

Operator scenarios invoke lower-level scenarios that serve as operands. These scenarios sometimes enforce implicit constraints on their operands. The **serial** operator is an example of an operator scenario. It takes as operands one or more scenarios and executes them in sequence.

Most operators have implicit **serial**. In other words, if they have more than one invocation in them, the invocations are put inside an implicit **serial**. Only **parallel**, **first_of**, **one_of**, and **mix** do not have implicit **serial**.

You can invoke scenario modifiers from some operator scenarios. To do this, define a **with** block as follows:

```
first_of:
    intercept_1()
    intercept_2()
with:
    position(distance: [5..100]m, behind: dut.car, at: start)
    position(distance: [5..15]m, ahead_of: dut.car, at: end)
    keep(car1.speed < 50kph)
```

Note: **with** blocks can also contain **keep()** constraints and **cover()** definitions.

The **with** block has four variants:

option 1: members listed as a block

```
with:
    member+
```

option 2: members passed as parameters

```
with(param+)
```

option 3: members passed as parameters and listed as a block

```
with(param+):
    member+
```

option 4: optional parameters and members listed on one line

```
with[(param+)]: member1 [; member2;...]
```

8.1. first_of

Purpose	Run multiple scenarios in parallel until the first one terminates
Category	Operator scenario

Syntax	`first_of:`
	`    scenario-list`
	`[with-block]`

Syntax parameters	*scenario-list* is a list of two or more scenarios. Each scenario is on a separate line, indented consistently from the previous line.
	with-block is a list of one or more scenario modifiers, **keep()** constraints or **cover()** definitions.

8.2. if

Purpose	Invoke a scenario depending on a condition
Category	Operator scenario

Syntax	**if (***bool-exp***):** *scenario+* **[else if (***bool-exp***):** *scenario+***]** **[else:** *scenario+***]**
Syntax parameters	*bool-exp* is an expression that evaluates to **true** or **false**.
	scenario+ is a list of one or more scenarios to invoke.

Note: Both the **else if** and the **else** clauses are optional. Multiple **else if** clauses are allowed.

Example

```
if x < y:
    cut_in()
    interceptor()
else if (x == y):
    two_cut_in()
else:
    out("x > y")
```

8.3. match

Purpose	Monitors a scenario and ends on first success or failure
Category	Operator scenario

Syntax	`match([anchored|floating] [, cover: true]):`
	`    monitored-scenario`
	`[then:`
	`    success-scenario ]`
	`[else:`
	`    fail-scenario ]`

Syntax parameters	*monitored-scenario* is the passive scenario that you hope to match with the scenario that is actually running.
	success-scenario is a scenario that informs you of the successful coverage collection.
	fail-scenario is a scenario that informs you of the failure of the match,
Scenario arguments	An **anchored** match tracks a single occurrence of the monitored scenario. It invokes *success-scenario* if matched, and *fail-scenario* otherwise. If abandoned, neither sub-scenario is invoked. **anchored** is the default.
	A **floating** match tracks all possible occurrences of *monitored- scenario*. It invokes *success-scenario* upon a first match (and ends). It invokes *fail-scenario* if abandoned before a match was found. Failures of tracked *monitored-scenario* are silently ignored.
	cover must be set to **true** for coverage to be collected. Because you might have several **match()** expressions monitoring the same scenario instance, coverage collection is off (**false**) by default.

Scenarios can be either active or passive. Both active and passive scenarios are interpreted as instructions for collecting coverage data, but only active scenarios actually execute. Passive scenarios are only monitored for coverage.

In order for coverage data from a passive scenario to be collected, the passive scenario must match a scenario that is actually executing. Matching a scenario means that every condition was met at the right time for the passive scenario to play out in its entirety.

match() invokes a monitor on *monitored-scenario,* the passive scenario. If a successful match occurs, *success-scenario* is invoked and coverage data for the monitored scenario is collected. Upon failure, *fail-scenario* is invoked. **match()** itself ends upon success or failure of the sub-scenarios.

Note: The failure of monitored-scenario does not fail **match()**. A failure within success-scenario or fail-scenario will fail **match()**.

8.4. multi_match

Purpose	Monitors a scenario for as long as the enclosing context exists
Category	Operator scenario

Syntax	`multi_match([anchored|floating] [, cover: true]):`
	`  *monitored-scenario*`
	`[then:`
	`  *success-scenario* ]`
	`[else:`
	`  *fail-scenario* ]`

Syntax parameters	*monitored-scenario* is the passive scenario that you hope to match with the scenario that is actually running.
	success-scenario is a scenario that informs you of the successful coverage collection.
	fail-scenario is a scenario that informs you of the failure of the match,
Scenario arguments	An **anchored** match tracks a single occurrence of the monitored scenario. It invokes *success-scenario* on every matched occurrence, and fail-scenario on every failed occurrence. If abandoned, neither sub-scenario is invoked. **anchored** is the default.

	A **floating** match tracks all possible occurrences of *monitored- scenario*. It invokes *success-scenario* on any match. It invokes *fail-scenario* if abandoned before a first match was found. Intermediate failed matches are silently ignored.
	cover must be set to **true** for coverage to be collected. Because you might have several **match()** expressions monitoring the same scenario instance, coverage collection is off (**false**) by default.

In contrast to **match()** , which ends on first success, **multi-match()** tracks monitored-scenario for as long as the enclosing context exists. If declared at the top level, **multi-match()** continues throughout the run.

8.5. mix

Purpose	Invoke a secondary scenario and mix it into the current scenario
Category	Operator scenario
Syntax	**mix**[**(***param**)]**:** *scenario-invocation+* [*with-block*]
Syntax parameters	*scenario-invocation* is a semi-colon separated list of the scenarios you want to invoke. These invocations can have the full syntax including argument list and member block.
	with-block is a list of one or more scenario modifiers, **keep()** constraints or **cover()** definitions.

Scenario arguments	*param* is a list of zero or more of the following: • A **start_to_start** parameter**.** • An **end_to_end** parameter. • An **overlap** parameter. If no parameters are specified, the parentheses are optional.
	The **start_to_start** *time-exp* is an expression of type time that specifies the time from the start of the primary scenario to start of the secondary scenarios.
	The **end_to_end** *time-exp* is an expression of type time that specifies the time from the end of the primary scenario to the end of the secondary scenarios.
	An **overlap** parameter is one of the following: • **any** specifies that there is no constraint on the amount of overlap between operands. This is the default. • **full** specifies that the operands fully overlap: start_to_start <= 0, end_to_end >= 0. • **equal** specifies that the operands have equal intervals: start_to_start == 0, end_to_end == 0 . • **inside** specifies that the secondary scenarios are inside the primary scenario: start_to_start >= 0, end_to_end =< 0. • **initial** specifies that the secondary scenarios cover at least the start of the primary scenario: start_to_start <= 0. • **final** specifies that the secondary scenarios cover at least the end of the primary scenario: end_to_end >= 0. **Note**: The various overlap parameters apply separately for each secondary operand. The secondary operands do not have to overlap each another in the manner described. For example, **mix(a,b,c,d)** is really **mix(mix(mix(a,b),c),d).**

mix is non-symmetrical: the first operand is primary determines the time and the context. The other operands are secondary. For example, given the scenario

```
scenario env.cut_in_with_rain:
    do mix():
        cut_in()
        rainstorm()
```

rainstorm begins and ends when **cut_in** begins and ends. This is different from

```
scenario env.cut_in_with_rain:
    do mix():
        rainstorm()
        cut_in()
```

The order of all the operands after the first is not important. The following are the same:

```
scenario env.cut_in_with_rain_and_pedestrian:
    do mix():
        cut_in()
        rainstorm()
        pedestrian_crossing()

scenario env.cut_in_with_rain_and_pedestrian:
    do mix():
        cut_in()
        pedestrian_crossing()
        rainstorm()
```

8.6. one_of

Purpose	Choose a sub-invocation randomly from a list
Category	Operator scenario

Syntax	``` one of: scenario-list [with-block] ```
Syntax parameters	*scenario-list* is a list of two or more scenarios. Each scenario is on a separate line, indented consistently from the previous line.
	with-block is a list of one or more scenario modifiers, **keep()** constraints or **cover()** definitions.

8.7. parallel

Purpose	Execute activities in parallel within one or more phases
Category	Operator scenario
Syntax	``` parallel[(duration: time-exp)]: invocation-list [with-block] ```
Syntax parameters	*invocation-list* is a list of the scenarios that you want to invoke in parallel. These invocations can have the full syntax including argument list and member block.
	with-block is a list of one or more scenario modifiers, **keep()** constraints or **cover()** definitions.
Scenario arguments	*time-exp* is an expression of type time specifying how long the activities occur. If no parameters are specified, the parentheses are optional.

The **parallel** operator describes several parallel activities, each of which lasts from the start of a phase to its end. A phase ends when any of the following conditions is met:

- If all of the scenario's sub-scenarios end, the scenario ends.
- If a **duration** parameter is specified for a phase, then it ends when the specified duration has been reached. For example:

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

8.8. repeat

Purpose	Run multiple scenarios in sequence repeatedly
Category	Operator scenario
Syntax	**repeat(**[*count*]**):** *scenario+* [*with-block*]
Syntax parameters	*scenario+* is a list of one or more scenarios.
	with-block is a list of one or more scenario modifiers, **keep()** constraints or **cover()** definitions.
Scenario arguments	*count* is the number of times to repeat the scenarios. If omitted, **repeat** loops until some outer context terminates it.

8.9. serial

Purpose	Execute two or more scenarios in a serial fashion
Category	Operator scenario

Syntax	`serial[(duration: time-exp)]:`
	`    scenario+`
	`[with-block]`

Syntax parameters	*scenario+* is a list of the scenarios that you want to invoke.
	with-block is a list of one or more scenario modifiers, **keep()** constraints or **cover()** definitions.
Scenario arguments	*time-exp* is an expression of type time specifying how long the activities occur. If no parameters are specified, the parentheses are optional.

The default execution for all scenarios you create is serial, with no gaps. You can add gaps between scenarios using the **wait** or **wait_time** scenarios.

Example

The following scenario declarations are the same.

```
scenario my_scenario:
    do first_scenario()
    do second_scenario()

scenario my_scenario:
    do serial():
        first_scenario()
        second_scenario()
```

8.10. try

Purpose	Run a scenario, handling its failure by invoking the else-scenario
Category	Operator scenario
Syntax	**try:** *scenario*+ [**else:** *else-scenario*+]
	scenario+ is a list of the scenarios that you want to invoke.
	else-scenario+ is a list of the scenarios you want to invoke if *scenario*+ fails.

try fails only if the *else-scenario*+ fails.

9. Event-related scenarios

Summary: This topic describes scenarios that perform event-related actions.

Other built-in scenarios perform event-related actions, such as waiting for an event or emitting an event. The **wait** scenario, for example, pauses the current scenario until the specified event occurs.

9.1. emit

Purpose	Emit an event in zero-time
Category	Built-in scenario
Syntax	`emit event-path[(param+)]`
Syntax parameters	*event-path* has the format [*field-path.*]*event-name.*
Scenario arguments	*param+* is a list of one or more parameters that are defined in the event declaration.

9.2. wait

Purpose	Delay action until the qualified event occurs
Category	Built-in scenario
Syntax	`wait qualified-event`

Syntax parameters	*qualified-event* has the format [*bool-exp*][**@***event-path* [**=>***name*]]. If *event-path* is missing, the expression is checked on **top.clk** (every callback from the simulator). If *bool-exp* is missing, **true** is assumed. At least one of *event-path* and *bool-exp* must be specified. If specified, the **=>***name* clause creates a pseudo-variable with that name in the current scenario – the *event object variable*. The variable is used to access the event fields, which is useful for collecting coverage over their values.

Note: any scenario has these predefined events: **start**, **end**, **fail**.

Examples

```
wait @my_event
wait (a > b) @my_event
wait (a > b)
```

9.3. wait_time

Purpose	Wait for a period of time
Category	Built-in scenario
Syntax	**wait_time(***time-exp***)**
Scenario arguments	*time-exp* is an expression of type time specifying how long to pause the scenario invocation. **Note**: Time granularity is determined by the simulator callback frequency.

Examples

```
wait_time([3..5]second)

# max is a field defined with type time and constrained to a value
wait_time(max)
```

10. Zero-time scenarios

Summary: This topic describes scenarios that execute in zero time.

10.1. call

Purpose	Call an external method
Category	Built-in scenario
Syntax	`call method(params)`
Syntax parameters	*method* is the name of a declared external method. If the method is not in the current context, the name must be specified as *path.name*.
	params is a comma-separated list of method parameters.

Example

```
call dut.break_camera()
```

10.2. dut.error

Purpose	Report an error and print message to STDOUT
Category	Built-in scenario
Syntax	`dut.error(string)`

Scenario arguments	*string* is a message that describes a **dut** error that occurred, enclosed in double quotes.

10.3. end

Purpose	End the current scope of the current scenario and optionally print a message
Category	Built-in scenario
Syntax	`end([string])`
Scenario arguments	*string* is an informational message

Example

```
cut_in with:
    on @foo:
        end()
```

10.4. fail

Purpose	Fail the current scope of the current scenario and optionally print a message
Category	Built-in scenario
Syntax	`fail([string])`

Scenario arguments	*string* is an informational message to facilitate debugging

Note: If not inside a **try** operator, the failure propagates up the invocation tree, failing the invoking scenarios.

10.5. Zero-time messaging scenarios

Purpose	Print message to STDOUT
Category	Built-in scenario

Syntax	`out(string)`
	`info(string)`
	`debug(string)`
	`trace(string)`

Scenario arguments	*string* is an informational message, enclosed in double quotes.

The following constructs are used to print messages at various levels of verbosity:

Name	Description	Visibility
Out	Used to report major events and messages	Always
Info	More detailed reporting	Low verbosity and above
Debug	Verbose information that may be useful for debug	Medium verbosity and above
Trace	Most detailed information used to trace execution	High verbosity only

Actual reporting visible during runtime is determined by the implementation, based on the desired verbosity level you set.

11. Movement scenarios

Summary: This topic describes scenarios that describe a continuous segment of movement.

Scenarios that describe a continuous segment of movement by a single actor are called movement scenarios. **car.drive** is a top-level scenario that moves a vehicle along a specified path, optionally with a scenario modifier such as **set_speed**. For example:

```
dut.car.drive(path) with:
    speed([30..70]kph)
```

11.1. drive

Purpose	Describe a continuous segment of movement by a car actor
Category	Movement scenario (**car** actor)
Syntax	**drive(path:** *path*[**, exactly:** *bool*, **adjust:** *bool*]**)**[*with-block*]
Parameters	**path** is the actual path (road) on which the drive is performed. This parameter is required.
	exactly specifies whether to perform the drive from the start to the end of the *path* or just some subset on it. Default: **false**.
	adjust specifies whether we perform automatic adjustment to achieve the desired synchronization specified for this drive (if any). Default: **false**.
	with-block is a list of one or more scenario modifiers, **keep()** constraints or **cover()** definitions.

The **with** block has four variants:

option 1: members listed as a block

```
with:
    member+
```

option 2: members passed as parameters

```
with(param+)
```

option 3: members passed as parameters and listed as a block

```
with(param+):
    member+
```

option 4: optional parameters and members listed on one line

```
with[(param+)]: member1 [; member2;...]
```

Example

```
car1.drive(path1, adjust: true) with:
    position([5..100]meter, behind: dut.car, at: start)
    position([5..15]meter, ahead_of: dut.car, at: end)

car2.drive(path1, adjust: true) with:
    no_collide(ref: car1)
```

12. Implicit movement constraints

Summary: This topic describes constraints that apply implicitly.

Consecutive movement scenarios of the same actor obey some obvious implicit constraints. In the following example, the consecutive **drive** scenarios of **car1** imply that the location, speed, and so on at the end of **d1** are the same as those at the start of **d2**.

```
serial():
    d1: car1.drive()
    d2: car1.drive()
```

Furthermore, an actor can appear in any set of (potentially overlapping) movement scenarios. In fact, the movement of the same actor can be sliced in multiple ways. For example:

- A **traverse_junction** scenario specifies three consecutive **drive** scenarios: **enter**, **during** and **exit.**
- A **tire_punctured** scenario also specifies three consecutive **drive** scenarios: **before**, **during** and **after.** In **after**, the car drives more slowly and erratically.

In a particular test, a specific car can be active in both **traverse_junction** and **tire_punctured.** These scenarios can be in arbitrary relation to each other: they might completely or partially overlap. It is the job of the planner to solve them all together.

13. Scenario modifiers

Summary: This topic describes scenario modifiers that constrain or modify the behavior of a scenario.

Scenario modifiers do not define the primary behavior of a scenario. Instead, they constrain or modify the behavior of a scenario for the purposes of a particular test. Scenario modifiers are especially useful if you just want to group together a group of related constraints or coverage definitions.

Scenario modifiers appearing outside the top-level scenario are moved into an (implicit) top-level **serial**. For example:

```
scenario top.foo:
    i: int
    set_map(my_map)
    do cut_in()
```

Means:

```
scenario top.foo:
    i: int
    do serial():
        cut_in() with:
            set_map(my_map)
```

Modifiers can only be used in

- The definition of other modifiers
- The **with** block of a scenario invocation
- At the top level of a scenario invocation, where it is equivalent to defining it within a **with** block attached to the invoked scenario
- The body of an **in** scenario modifier

You can invoke scenario modifiers with a path, such as **map.path_length()**, or give labels to modifiers, such as **s1: speed()**, but that is rarely needed.

13.1. in modifier

Purpose	Modify the behavior of a nested scenario
Category	Scenario modifier
Syntax	`in scenario-path with-block`
Syntax parameters	*scenario-path* is the path to the nested scenario instance whose behavior you want to modify. You can specify a conditional path, using **when()**.
	with-block is a list of one or more field constraints, cover definitions or scenario modifiers. Expressions in *with-block* can refer to values in the calling context (where it was invoked). This is done by using **outer** in a path expression.

All modifiers inserted via the **in** modifier are added to the original set of modifiers. Together they influence the behavior of the scenario. If there is any contradiction between them, then an error occurs.

Example

```
in x.y.my_drive with:
    speed([50..75]kph)
    lane(1)
```

Example with conditional path

The following two examples are equivalent. If **x** has color green, then find its field **y**, and then in it, add these modifiers.

```
# Example 1
in x.when(color: green).y with:
    speed([50..75]kph)

# Example 2
in x.when(color: green).y with(speed: ([50..75]kph))
```

*Example using **outer***

```
selected_lane: int
in x.y.my_drive with:
    speed([50..75]kph)
    lane(outer.selected_lane) # this modifies my_drive based on the value
                              # of the field selected_lane, within the "in"
                              # calling context
```

13.2. on qualified event

Purpose	Execute actions when an event occurs
Category	Scenario modifier
Syntax	**on** *qualified-event*: *member*
Syntax parameters	*qualified-event* has the format [*bool-exp*][@*event-path* [**=>***name*]]. If *event-path* is missing, the expression is checked on **top.clk** (every callback from the simulator). If *bool-exp* is missing, **true** is assumed. At least one of *event-path* and *bool-exp* must be specified. If specified, the **=>***name* clause creates a pseudo-variable with that name in the current scenario – the *event object variable*.
	member is a zero-time scenario such as **end, fail** or a zero-time messaging scenario.

Example

```
on @near_collision:
    info("Near collision occurred.")
```

13.3. synchronize

Purpose	Synchronize the timing of two sub-invocations
Category	Scenario modifier
Syntax	**synchronize (slave:** *inv-event1*, **master:** *inv-event2* [, **offset:** *time-exp*] **)**
Scenario arguments	*inv-event1* is a scenario invocation event that you want to synchronize with another scenario invocation event *inv-event2.* An invocation event has the form *invocation-label*[.*event*], where *invocation-label* is the label of some scenario invocation in the current scope. The default event is the **end** event of that invocation, but you can also specify **start**, for example, **drive1.start**.
	time-exp is an expression of type time. It signifies how much time should pass from the master event to the slave event. If negative, the slave event should happen *before* the master event.

Example

In this example, x.start should end five seconds after y.start.

```
x: cut_in()
y: intercept()

parallel():
    cut_in()
    intercept()
with: synchronize(x.start, y.start, offset: 5s)
```

13.4. until

Purpose	End an action when an event occurs
Category	Scenario modifier
Syntax	`until( qualified-event)`
Syntax parameters	*qualified-event* has the format [*bool-exp*][@*event-path* [**=>***name*]]. If *event-path* is missing, the expression is checked on **top.clk** (every callback from the simulator). If *bool-exp* is missing, **true** is assumed. At least one of *event-path* and *bool-exp* must be specified. If specified, the **=>***name* clause creates a pseudo-variable with that name in the current scenario – the *event object variable*.

Example

The **until** modifier has the same functionality as **on** *qualified-event* **: end()**, so the following two examples are the same.

```
# Example 1

do serial:
    phase1: car1.drive() with:
        speed(40kph)
        until(@e1)
    phase2: car1.drive() with:
        speed(80kph)
        until(@e2)


# Example 2

do serial:
    phase1: car1.drive() with:
        speed(40kph)
        on @e1:
            end()
    phase2: car1.drive() with:
        speed(80kph)
        on @e2:
            end()
```

14. Movement_related scenario modifiers

Summary: This topic describes scenario modifiers that specify or constrain attributes of movement scenarios.

Scenario modifiers specify or constrain attributes of movement scenarios such as **car.drive**. They must appear as members of other scenario modifiers or as members of a movement scenario after **with:**. For example:

```
dut.car.drive(path) with:
    speed([30..70]kph)
```

Here are some examples of other scenario modifiers:

```
# Drive behind truck1
position([20..50]meter, behind: truck1)

# Drive faster than truck1
speed([0.1..5.0]mph, faster_than: truck1)

# Drive one lane left of truck1
lane ([1..1], left_of: truck1)
```

The following scenario modifiers set an attribute throughout a period such as a phase:

```
speed()    # The speed
position() # The y (longitude) position
lane()     # The lane
```

The following scenario modifiers specify how an attribute changes over a period:

```
change_speed() # The change in speed
change_lane()  # The change in lane
```

The scenario modifiers that set a speed, position, and so on can be either absolute or relative. For example:

```
speed([10..15]kph)  # Absolute
speed([10..15]kph, faster_than: car1) # Relative
speed([10..15]kph, slower_than: car1) # Relative
```

The relative versions usually have multiple parameters such as **faster_than** and **slower_than**, but at most you can specify only one. This constraint is checked at compile time.

All these modifiers have an optional **at:** parameter, with the following possible values

- **all** – this constraint holds throughout this period (default)
- **start** – this constraint holds at the start of the period
- **end** – this constraint holds at the end of the period

14.1. acceleration

Purpose	Specify the rate of acceleration of an actor
Category	Scenario modifier
Syntax	**acceleration(acceleration:** *acceleration-exp***)**
Parameters	*acceleration-exp* is either a single value or a range appended with an acceleration unit. The unit is **kphps** (kph per second) or **mpsps** (meter per second per second).

Example

```
acceleration(5kphps)
# This accelerates by 5kph every second
# For example, from 0 to 100kph in 50 seconds.
```

14.2. change_lane

Purpose	Specify that the actor change lane
Category	Scenario modifier
Syntax	`change_lane(lane: value, side: av_side)`
Parameters	*value* is the number of lanes to change from, either a single value or a range. The default is 1.
	side is **left** or **right**.

Examples

```
# Change lane one lane to the left
change_lane(side: left)

# Change the lane 1, 2 or 3 lanes to the right
change_lane([1..3], right)
```

14.3. change_speed

Purpose	Change the speed of the actor for the current period
Category	Scenario modifier
Syntax	`change_speed(speed: speed)`

Parameters	*speed* is either a single value or a range with a speed unit. The default is [0..100]kph.

Example

```
change_speed([-20..20]kph)
```

14.4. keep_lane

Purpose	Specify that the actor stay in the current lane
Category	Scenario modifier
Syntax	`keep_lane()`
Parameters	None

Example

```
keep_lane()
```

14.5. keep_position

Purpose	Maintain absolute position of the actor for the current period
Category	Scenario modifier
Syntax	`keep_position()`

Parameters	None.

Example

```
keep_position()
```

14.6. keep_speed

Purpose	Maintain absolute speed of the actor for the current period
Category	Scenario modifier
Syntax	**`keep_speed()`**
Parameters	None.

Example

```
keep_speed()
```

14.7. lane

Purpose	Set the lane in which an actor moves
Category	Scenario modifier

Syntax	`lane([lane: lane] [right_of | left_of | same_as: car] | [side_of: car, side: av_side ] [at: event])`
Parameters	*lane* is the lane to drive in, either a single value or a range. The left-most lane is 1. Negative numbers mean to the right. The default is 1.
	car is a named instance of the car actor, for example car2.
	event is **start**, **end** or **all**. The default is **all,** meaning that the specified speed is maintained throughout the current period.

When **right_of** is specified, the context car should be slower than *car* by the specified value in the relevant period. **left_of** and **same_as** contradict **right_of**, so you cannot use them together.

Examples

```
# Drive in left-most lane
lane(1)

# Drive one lane left of lane 1
lane(left_of: car1)

# At the end of this phase, be either one or two lanes
# to the right of car1
lane([1..2], right_of: car1, at: end)

# Be either one left, one right or the same as car1
lane(in [-1..1], right_of: car1)

# Be in the same lane as car1
lane(same_as: car1)
```

14.8. no_collide

Purpose	Specify that the actor not collide with the referenced car

Category	Scenario modifier
Syntax	`no_collide(ref: car)`
Parameters	A car actor instance must be specified.

Example

```
no_collide(car1)
```

14.9. position

Purpose	Set the position of an actor along the x (longitude) dimension
Category	Scenario modifier
Syntax	`position(distance: distance | time: time, [ahead_of: car | behind: car], [at: event])`
Parameters	*distance* is a single value or a range with a distance unit. The default is [5..100]meters.
	time is a single value or a range with a time unit.
	car is a named instance of the car actor, for example car2.
	event is **start**, **end** or **all**. The default is **all,** meaning that the specified speed is maintained throughout the current period.

When **ahead_of** is specified, the context car must be ahead of *car* by the specified value in the relevant period. **behind** contradicts **ahead_of**, so you cannot use them together.

Examples

```
# Absolute from the start of the path
position([10..20]meter)

# 40 meters ahead of car1 at end
position(40meter, ahead_of: car1, at: end)

# Behind car1 throughout
position([20..30]meter, behind: car1)

# Behind car1, measured by time
position(time: [2..3]second, behind: car1)
```

14.10. speed

Purpose	Set the speed of an actor for the current period
Category	Scenario modifier
Syntax	**speed(speed:** *speed*, [**faster_than:** *car* | **slower_than:** *car*] [, **at:** *event*])
Parameters	*speed* is either a single value or a range with a speed unit. The default is [0..100]kph.
	car is the path of the car actor, for example car2.
	event is **start**, **end** or **all**. The default is **all,** meaning that the specified speed is maintained throughout the current period.

When **faster_than** is specified, the context car must be faster than *car* by the specified value in the relevant period. **slower_than** contradicts **faster_than**, so you cannot use them together.

Examples

```
# Absolute speed range
speed([10..20]kph)

# Faster than car1 by [1..5]kph
speed([1..5]kph, faster_than: car1)

# Have that speed at end of period
speed(5kph, at: end)

# Really either slower or faster than car1
speed([-20..20]kph, faster_than: car1)
```

15. Map-related scenario modifiers

Summary: This topic describes scenario modifiers that constraint the map or paths on the map.

Map-related scenario modifiers usually handle a parameter of type **path.** This parameter is the name of a field in the scenario representing a path in the current map. Some map constraints specify two path parameters, in which case there must be two fields in the scenario of type **path**.

15.1. path_curve

Purpose	Specify that the path has a curve
Category	Scenario modifier
Syntax	`path_curve(path: pathname, max_radius: radius, min_radius: radius, side: side)`
Parameters	*pathname* is the name of a path instance in the scenario.
	radius is a number of type **distance**.
	side is a value of type **av_side**, one of **right** or **left**.

Example

```
path_curve(path1, max_radius:11meter, min_radius:6meter, side: left)
```

15.2. path_different_dest

Purpose	Specify that two paths have different destinations

Category	Scenario modifier
Syntax	**path_different_dest(path1:** *pathname*, **path2:** *pathname***)**
Parameters	*pathname* is the name of a field in the scenario. There must be two fields of type **path**.

Example

```
path_different_dest(path1, path2)
```

15.3. path_different_origin

Purpose	Specify that two paths have different origins
Category	Scenario modifier
Syntax	**path_different_origin(path1:** *pathname*, **path2:** *pathname***)**
Parameters	*pathname* is the name of a field in the scenario of type **path.** There must be two fields.

Example

```
path_different_origin(path1, path2)
```

15.4. path_explicit

Purpose	Specify a path using a list of points from a map
Category	Scenario modifier
Syntax	**path_explicit(path:** *pathname*, **requests:** *list of point*, **tolerance:** *tolerance*)
Parameters	*pathname* is the name of a field in the scenario of type **path.**
	list of point is a list of points from a specific map. Use the **map.explicit_point()** method (defined in the ***e*** language) to translate an OpenDRIVE **road id** and **offset** to a point. **map.explicit_point()** has four parameters: • an OpenDrive segment id as a string • a subsegment – not used now; set to 0 • an offset – the distance from the start of the road • the lane number
	tolerance is an unsigned integer representing a percentage of the total path. For example, if the path length is 100 meter and the tolerance is 5, then the difference between the planned way point and the input way point is within 5 meters. The default is 0.

Example

This example specifies the **hooder.xodr** map. The first point is on the “-15” road and 20 meter from the start on the first lane. The second is 130 meter from the start.

```
extend top.main:
    do a: cut_in_and_slow with:
        set_map("/maps/hooder.xodr")
        path_explicit(a.path,
          [map.explicit_point("-15",0,20meter,1),
             map.explicit_point("-15",0,130meter,1)],
          tolerance:1)
```

15.5. path_facing

Purpose	Specify that two paths approach from opposite directions
Category	Scenario modifier
Syntax	**path_facing(path1:** *pathname*, **path2:** *pathname***)**
Parameters	*pathname* is the name of a field in the scenario of type **path.** There must be two fields.

Example

```
path_facing(path1, path2)
```

15.6. path_has_sign

Purpose	Specify that the path has a sign
Category	Scenario modifier
Syntax	**path_has_sign(path:** *pathname*, **sign:** *sign-type)*

Parameters	*pathname* is the name of a field in the scenario of type **path.**
	sign-type is one of the values of the enumerated type **sign_type**: **speed_limit**, **stop_sign**, **yield** or **roundabout**.

Example

```
path_has_sign(path1, sign: yield)
```

15.7. path_has_no_signs

Purpose	Specify that the path have no signs
Category	Scenario modifier
Syntax	**`path_has_no_signs(path:`** *`pathname`***`)`**
Parameters	*pathname* is the name of a field in the scenario of type **path.**

Example

```
path_has_no_signs(path1)
```

15.8. path_length

Purpose	Specify the length of a path and whether it might have an intersection
Category	Scenario modifier

Syntax	**path_length(path:** *pathname*, **min_path_length:** *min-distance*, **max_path_length:** *max-distance*, **allow_junction:** *bool***)**
Parameters	*pathname* is the name of a field in the scenario of type **path.**
	min-distance is a value of type **distance**. The default is 120meter.
	max-distance is a value of type **distance**. The default is 150meter.
	bool is **true** or **false**. The default is **true**.

Example

```
path_length(path1, min_path_length: 150meter,
      max_path_length: 175meter, true)
```

15.9. path_max_lanes

Purpose	Specify the maximum number of driving lanes in a path
Category	Scenario modifier
Syntax	**path_max_lanes(path:** *pathname*, **min_lanes:** *int***)**
Parameters	*pathname* is the name of a field in the scenario of type **path.**
	int is an integer value specifying the maximum number of lanes.

Example

```
path_max_lanes(path1, 2) # Needs no more than two lanes
```

15.10. path_min_driving_lanes

Purpose	Specify the minimum number of driving lanes in a path
Category	Scenario modifier
Syntax	`path_min_driving_lanes(path: pathname, min_driving_lanes: int)`
Parameters	*pathname* is the name of a field in the scenario of type **path.**
	int is an integer value specifying the minimum number of driving lanes.

Example

```
path_min_driving_lanes(path1, 2) # Needs at least two driving lanes
```

15.11. path_min_lanes

Purpose	Specify the minimum number of lanes in a path
Category	Scenario modifier
Syntax	`path_min_lanes(path: pathname, min_lanes: int)`
Parameters	*pathname* is the name of a field in the scenario of type **path.**
	int is an integer value specifying the minimum number of lanes.

Example

```
path_min_lanes(path1, 2) # Needs at least two lanes
```

15.12. path_over_junction

Purpose	Specify that the path pass through a junction
Category	Scenario modifier
Syntax	**path_over_junction(junction:** *junction*, **direction:** *direction*, **distance_before:** *distance*, **distance_after:** *distance*, **distance_in:** *distance*)
Parameters	*junction* is a field in the scenario of type **junction**.
	direction is a field in the scenario of type **direction** or one of: **other**, **straight** # (-20..20] degrees **rightish** # (20..70] degrees **right** # (70..110] degrees **back_right** # (110..160] degrees **backwards** # (160..200] degrees **back_left** # (200..250] degrees **left** # (250..290] degrees **leftish** # (290..340] degrees
	distance is a value or a range with a distance unit.

Example

```
scenario my_car.traverse_junction:
    junction: junction
    direction: direction

    path_over_junction(
        junction,
        direction,
        distance_before: [5..10]meter,
        distance_after: [5..10]meter)
```

15.13. path_over_lanes_decrease

Purpose	Specify that the number of lanes in a path must decrease
Category	Scenario modifier
Syntax	**path_over_lanes_decrease(path:** *pathname*, **sp_more_lanes_path_length:** *distance*, **more_lanes_path:** *sub-path* **)**
Parameters	*pathname* is the name of a field in the scenario of type **path.**
	distance is the length of the path that has more lanes.
	sub-path is the segment of the path that has more lanes. It must be of type **path**.

Example

```
path1: path
path1a: path

path_over_lanes_decrease(path: path1,
  sp_more_lanes_path_length: 20meter,
  more_lanes_path: path1a)
```

15.14. paths_overlap

Purpose	Specify that two path instances must overlap
Category	Scenario modifier
Syntax	`paths_overlap(path1: pathname, path2: pathname)`
Parameters	*pathname* is the name of a field in the scenario of type **path.**

Example

```
paths_overlap(path1, path2)
```

15.15. path_same_dest

Purpose	Specify that two paths have the same destination
Category	Scenario modifier
Syntax	`path_same_dest(path1: pathname, path2: pathname)`
Parameters	*pathname* is the name of a field in the scenario of type **path.**

Example

```
path_same_dest(path1, path2)
```

15.16. set_map

Purpose	Specify the map used in the test
Category	Scenario modifier
Syntax	`set_map(name: string)`
Parameters	The name of a map for the test must be specified.

Example

```
set_map("/maps/hooder.xodr")
```

16. Change log

Summary: This topic will show all significant changes to this manual by version.

16.1. Version 0.9

This version includes minor edits.

16.2. Version 0.8

The following changes appear in version 0.8:

- The **agent** type is changed to type **actor** because the term **actor** is used more commonly by our target audience.
- The **cover** modifier can be declared in **actor** and **struct** types, not just in scenarios.
- A new modifier, **until(** *qualified-event* **)**, is defined. It has the same functionality as **on** *qualified-event* **: end()**, but it is more readable.
- The syntax for declaring enumerated types and for declaring modifiers is now correct.
- The import statement description is updated with the default .sdl description and search sequence.
- Indentation units and the use of tabs is now clear.
- Integers in hexadecimal and readable decimal (100_000) format are supported.
- Use of the \ character either to escape a character within a string or to continue a string over multiple lines is clarified.
- The definition of “test” and the difference between “concrete scenario” and “directed scenario” have been clarified.
- The effect of adding modifiers using the **in** modifier is described.
- The tolerance parameter of the **path_explicit** modifier has been redefined.