Measurable Scenario Description Language Reference

Open Source Version 0.9.1

Last generated: January 21, 2020

Table of Contents

M-SDL Language Reference Manual

1. Introduction

Summary: This topic introduces the Measurable Scenario Definition Language.

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. Foretify can generate these variants automatically, within the constraints that you specify. Foretify then collects and aggregates 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, **top.main**. You control the execution flow of the program by adding scenarios to **top.main**.

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.

This document describes the M-SDL features that Foretify currently supports, except where otherwise noted. For a complete description of M-SDL, see the M-SDL Reference, Open Source version, when it becomes available in 2019.

For more information on using Foretify to measure the safety of your AV, see the Foretellix website .

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:

```
scenario traffic.scenario1:
    my_speed: speed

    keep(my_speed < 50kph)
```

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

```
do parallel():
    car1.drive(path)
    car2.drive(path) with:
        speed(speed: 20kph, faster_than: car1)
```

Structures also define events, for example:

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

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.

Example 1

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

Then it is extended in a different file to add another attribute.

```
# Extend an actor in a separate file
import my_car_group.sdl

extend my_car_group:
    average_speed: speed
```

Example 2 shows how to define a new scenario called **two_phases**. It defines a single actor, **car1**, which is a **green truck**. It uses the **serial** operator 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: path) with:
            speed(speed: 0kph, at: start)
            speed(speed: 10kph, at: end)
        phase2: car1.drive(path: path) with:
            speed(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(name: "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, **get_ahead**, **car1** gets ahead of the **dut.car**. This phase ends within 1 to 5 seconds, as defined by the **duration** parameter, when **car1** gets ahead of **dut.car** by 5 to 15 meters, as defined by the second **position()** modifier.
- In the second phase, **change_lane**, **car1** cuts in front of the **dut.car.** This phase starts when **get_ahead** finishes and ends within 2 to 5 seconds when **car1** is in the same lane as **dut.car**.

Note that both the **serial** and **parallel** operators are used in this scenario. The two phases are run in sequence, but within each phase, the movement of **car1** and **dut.car** are run in parallel.

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: path, min_driving_lanes: 2) # at least two lanes

    do serial():
        get_ahead: parallel(duration: [1..5]s): # get_ahead is a label
            dut.car.drive(path: path) with:
                speed(speed: [30..70]kph)
            car1.drive(path: 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: path)
            car1.drive(path: 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(side: left)        # c1 is a label
        c2: cut_in(side: right)       # c2 is a label
    with:
        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 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 **weather** scenario. The **mix_dangers** scenario has a single attribute of type **weather_type**, which is constrained to be not **clear**, because we want a dangerous situation. This attribute is passed to **weather**.

Example 7

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

scenario dut.mix_dangers:
    weather: weather_type
    keep(weather != clear)

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

Example 8 runs **mix_dangers**. In this case we chose to specify a concrete weather (**rain**) rather than letting it be a random, not-clear 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 syntax 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.

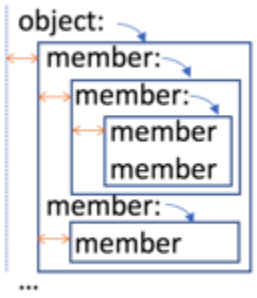


Figure 1 Code blocks

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.

Empty lines and single-line comments do not require a specific indentation.

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 concatenate strings with the plus character (+):

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

And you can continue strings onto multiple lines with the backslash character (\).

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

Inline comments are preceded by a hashtag character (#), and they end at the end of the line. Block comments are allowed. The first line in the block must begin with the /* characters and the last line must 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 a nested comment. */
# This is also a nested comment.
This is the last line of the block comment.
*/

extend top.main:
    do 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:

- Items that you can specify are shown within angle brackets <item>.
- Optional items are shown within square brackets [<item>].
- Items that you must choose between are shown separated by a bar <item> | <item>.
- Items that you can specify in a list are indicated by <item>*, meaning zero or more items of that type in the list, or by <item>+, meaning one or more items in the list. Parameter lists require commas between the parameters.
- Lists that require a separator other than a comma are shown with the separator and an ellipsis ;...

For example, given the syntax:

```
[!]<field-name>:[<type>][with:
    <member>+]
```

- <field-name>: is 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 it.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 identifiers, constants and keywords

User-defined identifiers (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 (_). User-defined identifiers beginning with an underscore are not allowed. The following identifiers are predefined in some contexts:

- **me** refers to the current type (struct, actor or scenario).
- **it** exists in a **with** context, referring to the **with** subject.
- **actor** exists in scenario declarations, and refers to the related actor instance.
- **outer** exists in an **in** context, and refers to the type in which **in** is declared.

3.3.1. Example of predefined identifiers

```
scenario top.C:
    c_val: int

scenario top.B:
    b_val: int
    do c1: C()

scenario top.A:
     a_val: int
     do b1: B()

# Example1:
extend top.A:
    in b1.c1 with:
     keep(outer.a_val == me.b_val)
     keep (me.b_val == it.c_val)

# In the above example:
#    outer: is the A scenario
#    me: is the B scenario
#    it: is the c1 scenario inside the B scenario

# Example2:
extend top.A:
    in b1 with:
        keep(outer.a_val == it.b_val)
        keep(me.a_val == it.b_val)

# In the above example:
#    outer: is the A scenario
#    me: is the A scenario
#    it: is the b1 scenario
```

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

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

actor	and	any	call	cover	def	default	do
else	emit	empty	event	extend	external	false	first_of

if	import	in	is	is also	is first	is only	keep
like	list of	match	mix	modifier	multi_match	not	null
on	one_of	or	parallel	properties	repeat	sample	scenario
serial	struct	soft	true	try	type	undefined	until
wait	when	with					

3.4. Overview of M-SDL constructs

M-SDL constructs can be divided into the following categories, based in part on the context in which they can appear:

- Statements
- Struct, actor or scenario members
- Scenario members
- Scenario invocations
- Expressions

The following sections describe each category and its members briefly.

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.

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**.

```
scenario car.my_scenario1:
    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.

```
scenario car.my_scenario1:
    legal_speed: speed
    keep(legal_speed < 120kph)
```

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

```
scenario car.my_scenario1:
    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. The events **start**, **end** and **fail** are defined for every scenario type. They are emitted whenever a scenario instance starts, ends or fails. In scenarios that invoke other scenarios, each phase may emit its own **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.

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.

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 car1 to be 1 to 5 kph faster relative to car2:

```
do parallel:
    car2.drive(path)
    car1.drive(path) with:
        speed([1..5]kph, faster_than: car2)
```

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

The default extension for M-SDL files is **.sdl**. You can define the search path for M-SDL files by specifying a list of directories to be searched in the **SDL_PATH** environment variable.

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 **top.main** 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 simulator_config.sdl
import my_scenario_top.sdl

extend top.main:
    set_map("hooder.xodr")
    do my_scenario(weather: rain)
```

3.6. Actor hierarchy and name resolution

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

It is recommended to represent scenario libraries as new global actors. For example, you should define simulator-specific scenarios for the **my_sim** simulator inside a global actor type **my_sim.** Then, create a field in the actor **top**. To facilitate readability, it is recommended to use the global actor type's name as the field name. For example, if **my_sim** is a global actor type, it is recommended to declare **my_sim** as follows:

```
extend top:
    my_sim: my_sim
```

When an M-SDL program is executed, the top-level actor's main scenario, **top.main**, is invoked. Extending this 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 path expressions. A path expression comprises steps connected by dots. Each step can be an identifier, a **when()** expression, a method call or an array element reference. The head (the first step in the path) may be a special identifier such as **outer** or **it**. For example, **cut_in.car1** is a path expression referencing a field **car1** in a scenario **cut_in()**.

Scenario invocations can have user defined labels. Automatic labels (implicit labels) are computed for all other invocations. Using labels, the behavior or attributes of a specific scenario or scenario invocation can be controlled from outside the scenario. See Automatic label computation (page 77) for how automatic labels are computed.

3.6.1. 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 actor's 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 (an actor member of **top**) 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. A scenario invocation must be associated with an actor instance.
- Scenario modifiers are searched by the same rules as scenarios.

3.6.2. 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 **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 an ordered collection of values of one type.

- String types hold a sequence of ASCII characters enclosed in double quotes.
- Resource types hold a list of resources such as map junctions and segments.
- 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 as well as reals, represented as 64-bit floating point numbers. Reals are equivalent to C++ **double**.

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. For reals, a decimal point may be used, as well as an exponent notation.

Name	Type
int	32-bit integer
uint	32-bit unsigned integer
int64	64-bit integer
uint64	64-bit unsigned integer
real	64-bit floating point number

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. The unit must be appended to the value without spaces. As shown in the table below, you have a choice of units for the most commonly used types.

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

Physical expressions that use fields rather than constants, such as **start_speed-1kph** are allowed in ranges.

Examples:

```
2meter
1.5s
[30..50]kph
[(start_speed-1kph)..(start_speed+1kph)]
```

Name	Units
acceleration	kphps (= kph per second), mpsps or meter_per_sec_sqr (= meters per second per second)
angle	deg, degree, rad, radian
angular_speed	degree_per_second, radian_per_second
distance	mm, millimeter, cm, centimeter, in, inch, feet, m, meter, km, kilometer, mile
speed	kph, kilometer_per_hour, mph, mile_per_hour, meter_per_second, mps
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 enumerated type **my_driving_style** has two values, **aggressive** and **normal**.

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

3.7.5. List types

A list is a way to describe an ordered collection of similar values in M-SDL. A list can contain any number of elements from a data type, including:

- Previously defined lists (list within list).
- Calls to methods that return the same data type as that of the list.
- The results of an operation, such as the evaluation of a Boolean expression.

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

```
shape2: 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,“”.

Use string interpolation (embedding $() in a string) to construct strings out of non-string values. The **$()** operator converts an expression to a string and inserts it in place. For example:

```
do serial:
     info("There are $(cars.size()) cars.")
```

You can concatenate multiple strings with the + character. Note that both sides of the concatenation operation must be of type **string**, 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. 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**.

3.7.9. 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, for example the operator 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(dut.car.color == green)
```

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_rain()
```

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)
timing(time_of_day: evening, specific_time: 18hr)
```

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

Example

```
import cut_in.sdl

scenario dut.cut_in_and_rain:
    do mix():
        cut_in()
        weather(rain)
        timing(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.

Name	Description
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 that 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_at_kind	start, end, all
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
line	left (Left side of the car), right (Right side of the car), center (Center of the car)
road_condition	paved, gravel, dirt
road_type	unknown, highway, highway_entry, highway_exit, highway_entry_exit
sign_type	speed_limit, stop_sign, yield, roundabout
time_of_day	undefined_time_of_day, sunrise, morning, noon, afternoon, sunset, evening, night, midnight
weather_type	undefined_weather, clear, cloudy, foggy, light_rain, rain, heavy_rain, light_snow, snow, heavy_snow

3.8. M-SDL operators and special characters

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

Operator type	Operators	Description
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
Boolean implication	<exp1> => <exp2>	Returns **true** when the first expression is **false** or when the second expression is **true**. This construct is the same as: **(not exp1) or (exp2)**
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

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 at character @ is used in a qualified event to indicate the pathname of an event, for example **@main_car.arrived**.
- 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.
- The character combination => is a syntactic marker used to bind an event's data to an identifier, so that the data is accessible for other purposes, such as coverage.

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
basic clock	Is the fastest occurring event. In simulation context it is emitted on each callback from the simulator. It is sometimes referred to as **top.clk**.
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 field whose value you want to collect at specific times during a run. A cover item’s value is updated by sampling.
cover group	A group of 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.

Term	Definition
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 concurrently.

Term	Definition
path expression	Comprises steps connected by dots. Each step can be an identifier, a **when()** expression, a method call or an array element reference. The head (the first step in the path) may be a special identifier such as **outer** or **it**.
phase	An informal term that describes a scenario invocation within a scenario in serial execution mode. As a specific example, **do serial** is often used to define the behavior of a scenario. Any scenario invocation within this behavioral definition, whether it is another builtin scenario such as **parallel** or a user-defined scenario, is a phase.
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 **top.main** 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 goal, as expressed by the modifier set and additional constraints.
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 an extension of the top-level scenario **top.main**. 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, and an extension of the top-level scenario **top.main** to define the behavior of the test.
test suite	A set of tests designed to verify a particular behavior and attributes or a set of those.

Term	Definition
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
- **event** declarations
- **when** subtypes
- external method declarations

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

Description

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 @dut.too_close =>col
    cover(col.data.x, event: near_collision, unit: centimeter)
    cover(col.data.y, event: near_collision, unit: centimeter)
    cover(col.data.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, enclosed in square brackets and placed on one or more lines, of zero or more enumerations 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.

Description

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 my_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-name> :
    <member>+
```

Syntax parameters

<type-name>

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**.

Description

At compile time, **extend** modifies the type and thus all instances of the type for the test in which it is included. **extend** allows you to encapsulate attributes or behaviors as an aspect of an object. It also allows you to modify built-in actors, structs and scenarios.

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**.

```
import storm_data.sdl

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(x: 5)
        f2: foo(x: 3)

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 <path-name>
```

Syntax parameters

<path-name>

Is the relative or full (global) path, including the file name, of the file to import. It may contain environment variables, preceded by '$'. The default file extension is **.sdl**. Wildcards are not allowed.

Description

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. The file name as specified is looked up first, and if not found, the name with the extension **.sdl** is attempted.
2. Else look in the current directory.
3. Else look in directories specified by the SDL_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
- **event** declarations
- **when** subtypes
- External method declarations
- Scenario modifier invocations

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

Description

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_type
    p: path
    car1: car
    keep(speed > 10kph)
    speed(speed: s)
    lane(lane: 1)
```

Example 2

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

    keep(lanes > 2)
    path_min_driving_lanes(path: p, min_driving_lanes: lanes)
    path_curve(path: 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. Parentheses are not allowed in scenario declarations.

<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 modifiers

A **do** member that describes the behavior of the scenario is required.

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

Note: Every scenario has a predefined **duration** field of type **time**.

Description

To define the behavior of a scenario, you must use the **do** member to invoke an operator scenario, such as **serial**, a library scenario, such as **car1.drive()**, or a user-defined scenario.

You can control the behavior of a scenario and collect data about its execution by declaring data fields and other members in the scenario itself, in its related actor or structs, or in the test file. For example, the **car** actor has a field of type **speed** that allows you to use the **speed()** scenario modifier to control 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 7) 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
- **event** declarations
- **when** subtypes
- external method declarations

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

Description

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]

struct storm_data:
    storm: storm_type

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

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

Note: cover() is also allowed in the **with** block of field declarations.

Syntax

```
cover(<exp>, [ <param>* ])
```

Syntax parameters

<exp>

Is an expression using objects in the enclosing construct. The expression must be of scalar type. The value of the cover item is the value of the expression when the cover group event occurs.

<param>*

Is a comma-separated list of zero or more of the following:

- name: <name> specifies a name for the cover item. By default, <name> is the same as <exp>, with any sequence of non-alphanumerics replaced by an underscore.

  For example, if you specify **cover(speed1 - speed2)**, the default name for this item is **speed1_speed2**. Note that the operator and surrounding white space was replaced by an underscore. You can change this default name using the name parameter, for example **cover(speed1 - speed2, name: speed_difference)**.

- 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> is explanatory text, enclosed in double quotes, about this collection point.
- ignore: <item-bool-exp> defines values that are to be completely ignored. 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).

Description

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**.

If the range of data that you want to collect 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(speed1, 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 field 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:= sample(map.abs_distance_between_locations(
      dut.car.location, car1.location), @slow.end) with:
        cover(it, text: "car1 position 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_cls, dut_v_cls,rel_v_cls],
    text: "Cross coverage of relative distance and absolute velocity")
```

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:= sample(car1.speed, @start) with:
    cover(it, unit:kph)
```

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 a 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 basic clock is used. 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 (an *event object variable*) with that name in the current scope, the current scenario for example. This notation is allowed only for events with parameters. You can use this variable to access the values of the event parameters, possibly collecting coverage over their values.

Description

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.

The events **start**, **end** and **fail** are defined for every scenario type. They are emitted whenever a scenario instance starts, ends or fails. In scenarios that invoke other scenarios, each phase may emit its own **start**, **end** and **fail**. When a scenario fails, the **fail** event is emitted, and the scenario terminates 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).

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 basic clock is used. 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 basic clock is used. 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, dist: distance)
```

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

```
do serial:
    car2.drive(path1) with:
        on (map.abs_distance_between_locations(dut.car.location,
          car2.location) < 2m):
          emit dut.near_collision(other_car: car2, distance: distance1)
```

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 @dut.too_close =>col
    cover(col.data.x, event: near_collision, unit: centimeter)
    cover(col.data.y, event: near_collision, unit: centimeter)
    cover(col.data.other_car, event: near_collision)
```

5.3. external method declaration

Purpose

Declare a procedure written in a foreign language

Category

Struct, actor, or scenario member

Syntax

```
def <method-name> (<param>*) [: <return-type>] [is empty | is undefined | is <bind-ex
p> |is first <bind-exp> | is only <bind-exp>  | is also <bind-exp> ]
```

Syntax parameters

<method-name>

Is a unique name in the current context.

<param>*

Is a list composed of zero or more arguments separated by commas of the form

```
<param-name>: <param-type> [= <default-exp>]
```

The parentheses are required even if the parameter list is empty.

<return-type>

Specifies the type of the return parameter.

<bind-exp>

Is in the form:

```
external.[e | python | cpp | shell] ( <param>* )
```

Description

The following name but do not define an external method:

- **is empty**
- **is undefined**

Both **empty** and **undefined** let you declare a method without specifying any actions. **undefined** causes an error if the method is called.

The following specify an extension to a previously declared method:

- **is also** appends the specified method to the previously declared method.
- **is first** prepends the specified method to the previously declared method.
- **is only** replaces the previously declared method with the specified method.

You can call external methods using the **call** zero-time scenario. If a method returns a value, you can use it in any place where an expression can be used, such as in constraint expressions, in qualified event expressions, in coverage computations and so on.

When called, these methods execute immediately in zero simulated time.

Example

```
extend dut:

    def break_camera() is also external.e("sensors.camera_malfunction()", "int", "boo
l")

extend top.main:
    do serial:
        call dut.break_camera()
```

5.4. field

Purpose

Declare a field to contain data

Category

Struct, actor, or scenario member

Syntax

```
[!]<field-name>: [ <type> ][= <sample>][ <with-block> ]
```

Syntax parameters

!

The do-not-generate operator (!) specifies that no value should be generated for this field before the run executes. Instead, a value is 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, <type> can be omitted if the type can be 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 basic clock is used. At least one of <event-path> and <bool-exp> must be specified.

If specified, the => <name> clause creates a pseudo-variable (an *event object variable*) with that name in the current scope, the current scenario for example. This notation is allowed only for events with parameters. You can use this variable to access the values of the event parameters, possibly collecting coverage over their values.

<with-block>

Is in a form such as:

```
with [ (<when-subtype>) ]:
    <member>+
```

See below for other syntax options for <with-block>.

<when-subtype> is in the form:

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

Note: When you define a <when-subtype>, you cannot pass other fields as parameters; you must specify those fields on separate lines in the **with** block.

<member>+ depends on the type of the enclosing struct.

For struct or actor fields, <member> is a constraint in the form

```
keep([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

  ```
  keep(<constraint-strength> <constraint-bool-exp>)
  ```

  where <constraint-strength> is either **soft** or **default**.

  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.

- 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: instantiate a **when** subtype without members

```
with(<param>*)
```

option 3: instantiate a **when** subtype with members

```
with(<param>*):
    <member>+
```

option 4: instantiate a **when** subtype with 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 **current_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
    !current_speed: speed

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

Example list field

```
extend traffic:
    my_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 it.size() <= 10)
        keep(soft it.size() >=  2)
        keep(it[0] == first_car) # list indexing
```

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:
        keep(it.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: 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) with:
        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([<constraint-type>] <constraint-boolean-exp>)
```

Syntax parameters

<constraint-type>

Is either **soft** or **default**. **default** is allowed only in scenario fields. If neither **soft** nor **default** is specified, the constraint type is *hard*. See below for a description of these constraint types.

<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 <constraint-bool-exp>:

Type	Operator	Function
Boolean comparison	==, !=, <, <=, >, >=	compare two values
Boolean range or list	**in** [<range-or-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
Boolean implication	<exp1> => <exp2>	returns **true** when the first expression is **false** or when the second expression is **true**. This construct is the same as: **(not exp1) or (exp2)**

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

The order of precedence for compound Boolean operators is (from tightest to least tight): parentheses, constants, path expressions and function calls, negation, comparison and range, **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.

Description

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(it 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))
```

List constraints

You can use the list method **.size()** and list indexing (*list*[*index*]) in list constraint expressions. In the following 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 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 it.size() <= 10)
        keep(soft it.size() >=  2)
        keep(it[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 == 25kph)
```

- 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

Default constraints seem like soft constraints. However, default constraints are only overwritten by a hard constraint directly on that object, whereas soft constraints are ignored if they contradict hard constraints or a later-applied soft constraint.

Below is an example of a default constraint. Adding the constraint **keep(y==0)** causes a contradiction because the value of **x** remains 0. The default constraint holds because there is no direct overriding constraint.

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

In the example below, the default constraint is replaced by a soft constraint. Here, adding the constraint **keep(y==0)** does not cause a contradictions because the value of **x** is changed to some non-zero value.

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

5.6. when subtype

Purpose

Create a conditional subtype in a struct, actor or scenario

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>: <constant-exp> where:

- <field-name> is the name of a field in the base type. Only Boolean or enumerated type fields (the **when** determinants) can be passed. **Note:** If you want to constrain a field that is defined under a **when** subtype, you cannot pass this field as a parameter; you must specify this field on a separate line in the **with** block. See Example constraining a field defined in a when subtype (page 0)
- <constant-exp> is a constant, 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.

Description

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 objects and 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]

struct storm_data:
    storm: storm_type

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

Example scenario subtype

If the following subtypes are defined in a scenario:

```
scenario env.bad_weather:
    kind: weather_type
    when(kind: rain):
        amount: distance with:
            keep(it >= 10cm)
    when(kind: snow):
        amount: distance with:
            keep(it >= 15cm)
```

You can then activate the scenario subtype with:

```
do bad_weather(kind: rain, amount: 15cm)
```

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

Example constraining a field defined in a when subtype

If you want to constrain a field defined under a when subtype, you cannot pass this field as parameter; you must specify this field on a separate line in the **with** block. In the following example, **storm** is the **when** determinant; **snow_depth** is a field in the subtype to be constrained:

```
d: storm_data with(storm: snow):
    keep(it.snow_depth>=15cm)
```

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. Scenario modifiers have parameters that constrain the movement scenario of an actor (movement modifiers) or the selection of a path on a map (map modifiers). To set these parameters, you can pass them on the modifier's argument list, and/or define a **with** block.

Note: **with** blocks can also contain **keep()** constraints. Below the examples is a description of the **with** block syntax.

Example map modifier:

The **path_has_sign()** modifier specifies that the selected path must have a yield sign.

```
extend top.main:
    do a: cut_in_and_slow() with:
        set_map("hooder.xodr")
        path_has_sign(a.path1, sign: yield)
```

Example movement modifier:

The **position()** modifiers specify the position of **car1** at the beginning and end of the phase relative to **dut.car**.

```
do parallel:
    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:
    i1: intercept_1()
    i2: intercept_2()
with:
    p1: position(distance: [5..100]m, behind: dut.car, at: start)
    p2: position(distance: [5..15]m, ahead_of: dut.car, at: end)
    keep(car1.speed < 50kph)
```

The **with** block has two variants:

option 1: members listed as a block

```
with:
    <member>+
```

option 2: members listed on one line

```
with: <member1> [; <member2>;...]
```

6.2. do (behavior definition)

Purpose

Define the behavior of a scenario

Category

Scenario member

Syntax

```
do <scenario-invocation>
```

Syntax parameters

See Scenario invocation (page 75) for a description of <scenario-invocation>.

Description

do is required within a scenario declaration or extension in order to define scenario behavior. Invoking a scenario causes its **do** member to be activated.

You define a scenario's behavior by invoking built-in scenarios, library scenarios, and user-defined scenarios:

- 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 in a user-defined scenario, 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.

Two operator scenarios commonly used to define scenario behavior are **serial** and **parallel**. For further complexity, use the **mix** operator to mix multiple scenarios. For example, a weather scenario can be mixed with a car scenario. See Operator Scenarios (page 79) for a description of these and other operators.

Within a scenario declaration or extension, use **do** once and only once in a scenario declaration or extension. Do not user **do** when invoking any nested scenario. For example **do** is used below to invoke **serial**, but omitted when invoking **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:
    car1: car

    set_map(name: "my map")
    do z: car1.zip()
```

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

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

These multiple **do** clauses are required because they are in separate scenario declarations or extensions, and they execute in sequence. In this example, the sequence is **turn**, **yield**, and **intercept**.

To make your code easily readable, create an explicit, meaningful label for the scenario invoked by **do** as well as all nested scenario invocations. Invocations without an explicit label are labeled automatically.

In the following example, there is an explicit label for the **serial** invocation at the root of the tree. The remaining invocations are labeled automatically. See Automatic label computation (page 77) for how automatic labels (implicit labels) are computed.

```
extend top.main:
    car1: car
    path1: path

    do a: serial():                # explicit label "a"
        car1.drive(path1) with:
            s1: speed(0kph, at: start)
            s2: speed(10kph, at: end)
```

7. Scenario invocation

Summary: This topic describes how to invoke a scenario.

A scenario can only be invoked within another (operator) scenario or within a **do** clause of a scenario definition.

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. If a label is not specified, an automatic label is created. See Automatic labels (page 77) for an explanation of how automatic labels are computed.

<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 21) 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, in some special cases, order-based (<value>,…). In order-based lists, the first value is assigned to the first field in the scenario, and so on. <value> can be a single value or a range. A unit is required if the type is physical.

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.

When invoking an operator scenario, parentheses are allowed, but not required. When invoking all other scenarios, parentheses are required.

<with-block>

Contains one or more of the following:

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

Note: cover() definitions are not allowed in scenario invocations.

The **with** block has two variants:

option 1: members listed as a block

```
with:
    <member>+
```

option 2: members listed on one line

```
with: <member1> [; <member2>;...]
```

Example 1

This example shows the declaration of a scenario called **traffic.two_phases**. **do** is required to define the behavior of **two_phases**, and it invokes the **serial** operator scenario. The nested scenario, **car1.drive**, whose behavior is defined as a library scenario, 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)
```

7.1. Automatic Label Computation

Scenario and modifier activations that have no user-defined labels get automatically generated labels (called implicit labels) according to the following algorithm:

Implicit label access syntax is **label(** <label-path> **)**, where <label-path> is a concatenation of identifiers using a period (.) as separator.

- If the scenario invocation has a path, the first identifier in the path is used
- Else, the scenario name is used
- If the resulting label is not unique, the suffix **(** <num> **)** is added, where <num> is the next sequential number. The numbering starts from **2**.

Example

The following example shows a scenario with no explicitly defined labels and a **keep()** constraint that uses the labels.

```
scenario dut.scen1:
    path: path
    car1: car
    do serial:                                  # label(serial)
        car1.drive(path) with: keep_lane()      # label(serial.car1)
        car1.drive(path) with:                  # label(serial.car1(2))
            lane(1)                             # label(serial.car1(2).lane)
            speed(speed: [30..120]kph)          # label(serial.car1(2).speed)
    with:
        keep(label(serial.car1(2).speed).speed == 45kph)
```

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. Operator scenarios have all the attributes of any scenario, such as **start**, **end** and **fail** events.

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, for example:

```
first_of:
    i1: intercept_1()
    i2: intercept_2()
with:
    p1: position(distance: [5..100]m, behind: dut.car, at: start)
    p2: position(distance: [5..15]m, ahead_of: dut.car, at: end)
    keep(car1.speed < 50kph)
```

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

The **with** block has two variants:

option 1: members listed as a block

```
with:
    <member>+
```

option 2: members listed on one line

```
with: <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 or **keep()** constraints.

Description

This operator runs multiple scenarios in parallel until the first one terminates. The other members, if any, are abandoned, meaning they are stopped without emitting the end event or collecting coverage.

Example

```
first_of:
    i1: intercept_1()
    i2: intercept_2()
with:
    p1: position(distance: [5..100]m, behind: dut.car, at: start)
    p2: position(distance: [5..15]m, ahead_of: dut.car, at: end)
    keep(car1.speed < 50kph)
```

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**. Enclosing parentheses are optional.

<scenario>+

Is a list of one or more scenarios to invoke.

Description

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: <bool>]):
    <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.

Description

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: <bool>]):
    <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.

Description

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 list of the scenarios you want to invoke. Each scenario is on a separate line, indented consistently from the previous line. These invocations can have the full syntax including argument list and member block.

<with-block>

Is a list of one or more scenario modifiers or **keep()** constraints.

Scenario arguments

<param>* is a list of zero or more of the following parameters:

- start_to_start: <time-exp>
- end_to_end: <time-exp>
- overlap: <type>

If no parameters are specified, the parentheses are optional.

The **start_to_start** parameter specifies the time from the start of the primary scenario to start of the secondary scenarios.

The **end_to_end** parameter specifies the time from the end of the primary scenario to the end of the secondary scenarios.

The **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)**.

Description

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 dut.cut_in_with_rain:
    do mix():
        cut_in()
        rainstorm()
```

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

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


scenario dut.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 or **keep()** constraints.

Example

```
one_of:
    i1: intercept_1()
    i2: intercept_2()
with:
    p1: position(distance: [5..100]m, behind: dut.car, at: start)
    p2: position(distance: [5..15]m, ahead_of: dut.car, at: end)
    keep(car1.speed < 50kph)
```

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 or **keep()** constraints.

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.

Description

The **parallel** operator describes the activities of two or more actors that start concurrently. For example:

```
p1: parallel:
    car1.drive(path)
    weather(kind: clear)
```

To describe consecutive activities of multiple actors, invoke **parallel** multiple times from within the **serial** operator. For example:

```
s1: serial:
    p1: parallel:
        car1.drive(path)
        weather(kind: clear)
    p2: parallel:
        car1.drive(path)
        weather(kind: rain)
```

Each invocation of **parallel** is referred to informally as a *phase*. Each activity (or nested scenario) commences at the start of a phase. A phase ends when any of the following conditions is met:

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

Failure of any member causes **parallel** to fail.

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 or **keep()** constraints.

Scenario arguments

<count> is the number of times to repeat the scenarios. If omitted, **repeat** loops until some outer context terminates it.

If no parameters are specified, the parentheses are optional.

Example

```
repeat(3):
    i1: intercept_1()
    i2: intercept_2()
with:
    p1: position(distance: [5..100]m, behind: dut.car, at: start)
    p2: position(distance: [5..15]m, ahead_of: dut.car, at: end)
    keep(car1.speed < 50kph)
```

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 or **keep()** constraints.

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.

Description

In serial execution mode, each member starts when its predecessor ends. The scenario ends when the last member ends. Failure of any member causes **serial** to fail.

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

In the following example, a gap of 3 to 5 seconds is added between the execution of **first_scenario()** and **second_scenario()**.

```
scenario dut.my_scenario:
    do serial():
        fs: first_scenario()
        w1: wait_time([3..5]second)
        ss: 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>+]
```

Syntax parameters

<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.

Description

try fails only if <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 comma-separated list of one or more parameters defined in the event declaration in the form <param-name>: <value>. Passing parameters by position is not allowed.

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 basic clock is used. 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.

Description

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

Examples

```
do serial:
    w1: wait @top.my_event
    w2: wait (a > b) @top.my_event
    w3: 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

```
do serial:
    w1: wait_time([3..5]second)

    # max is a field defined with type time and constrained to a value
    w2: 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>(<param>*)
```

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>.<method-name>

<param>*

Is a comma-separated list of method parameters.

Example

```
extend dut:

    def break_camera() is also external.e("sensors.camera_malfunction()", "int", "boo
l")

extend top.main:
    do serial:
        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

Description

This scenario ends the current scope of the current scenario, emitting the **end** event and collecting coverage. You can optionally print a message.

Example

```
do 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.

Description

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.

Description

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** moves a vehicle along a specified path, optionally with a scenario modifier such as **speed**. For example:

```
do serial:
    car1.drive(path1) 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 to 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 or **keep()** constraints.

The **with** block has two variants:

option 1: members listed as a block

```
with:
    <member>+
```

option 2: members listed on one line

```
with: <member1> [; <member2>;...]
```

Example

```
do parallel:
    car1.drive(path1, adjust: true) with:
        p1: position([5..100]meter, behind: dut.car, at: start)
        p2: position([5..15]meter, ahead_of: dut.car, at: end)

    car2.drive(path1, adjust: true) with:
        collides(true)
```

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**.

```
do serial():
    d1: car1.drive(path1)
    d2: car1.drive(path1)
```

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 modifiers.

Scenario modifiers invoked outside the **do** behavioral definition 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 invoked in

- The definition of other modifiers
- The **with** block of a scenario invocation
- At the top level of a scenario declaration, 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 or scenario modifiers, where the members are listed on separate lines as a block or on the same line as **with:**, separated by semi-colons.

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.

Description

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 the following example, the speed and lane constraints apply to the **car1.drive** movement in the **get_ahead** phase of the **dut.cut_in** scenario.

```
do cut_in() with:
    in label(cut_in).label(get_ahead.car1) with: speed([50..75]kph); lane(1)
```

*Example using **outer***

You can use the keyword **outer** to constrain a nested scenario from an enclosing scenario or test. In the following example, the constrained speed is defined in a **drive_attributes** struct that is instantiated in the test file.

Notes:

- In this example, no speed constraints are applied to **car1.drive()** in the nested scenario. This avoids the possibility of constraint contradictions when it is nested within different scenarios.
- Because the invocations of the nested and enclosing scenarios are labeled explicitly as **nested** and **enclosing**, and **car1.drive()** is labeled as **start_up**, the scenario pathname for **car1.drive()** is **enclosing.nested.start_up**.

```
struct drive_attributes:
    my_speed: speed with:
        keep(it <= 70kph)

scenario dut.nested_scenario:
    car1: car
    path: path

    do serial:
        start_up: car1.drive(path)

scenario dut.enclosing_scenario:
    drv_attr: drive_attributes
    my_car: car
    my_path: path

    do nested: dut.nested_scenario(car1: my_car, path: my_path)

extend top.main:

    do enclosing: dut.enclosing_scenario() with:
        in enclosing.nested.start_up with:
            speed(outer.drv_attr.my_speed)
```

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 basic clock is used. 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.

<member>

Is a zero-time scenario such as **end**, **fail** or a zero-time messaging scenario.

Example

```
do serial:
    car1.drive(path) with:
        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>, <inv-event2>

<inv-event1> is a scenario invocation event that you want to synchronize with <inv-event2> (another scenario invocation event).

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**.</td>

<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.

Description

The **synchronize()** modifier accepts two events. You can synchronize more events by using multiple statements.

Example

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

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

13.4. until

Purpose

End an invoked scenario 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 basic clock is used. 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.

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


# Example 2

do serial:
    phase1: car1.drive(path1) with:
        speed(40kph)
        on @e1:
            end()
    phase2: car1.drive(path1) 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:

```
do serial:
    car1.drive(path) with:
        speed([30..70]kph)
```

Here are some examples of other scenario modifiers:

```
do parallel:
    truck1.drive(path)
    car1.drive(path) with:
        # 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 scenario or a scenario 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 require two vehicles moving in parallel. They may also have multiple parameters such as **faster_than** and **slower_than**, but at most you can specify only one. These two 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

Parameters can be passed by name or by order. If no arguments are passed, the defaults are applied. For example, the following three **lane()** invocations are supported and have the same effect.

```
lane(lane: 1)
lane(1)
lane()
```

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

```
do serial:
    car1.drive(path) with:
        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.

<av-side> is **left** or **right**. <av-side> is randomized if not specified.

Examples

```
do serial:
    car1.drive(path) with:
        # Change lane one lane to the left
        change_lane(side: left)


do serial:
    car1.drive(path) with:
        # 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. You must specify a speed unit.

Example

```
do serial:
    car1.drive(path) with:
        change_speed([-20..20]kph)
```

14.4. collides

Purpose

Allow or disallow an actor to collide with another object.

Category

Scenario modifier

Syntax

```
collides([allow_collides: ]<bool>)
```

Parameters

<bool> is either **true** or **false**.

Description

By default, all actors disallow collisions (**collides(false)**). This means that they take defensive measures to avoid collisions. When set to **true**, the actor moves regardless of surrounding traffic and may collide.

Example

```
do serial:
    car1.drive(path) with:
        collides(true)
```

14.5. keep_lane

Purpose

Specify that the actor stay in the current lane.

Category

Scenario modifier

Syntax

```
keep_lane()
```

Example

```
do serial:
    car1.drive(path) with:
        keep_lane()
```

14.6. keep_position

Purpose

Maintain absolute position of the actor for the current period.

Category

Scenario modifier

Syntax

```
keep_position()
```

Example

```
do serial:
    car1.drive(path) with:
        keep_position()
```

14.7. keep_speed

Purpose

Maintain absolute speed of the actor for the current period.

Category

Scenario modifier

Syntax

```
keep_speed()
```

Example

```
do serial:
    car1.drive(path) with:
        keep_speed()
```

14.8. 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: <a
v-side>][at: <event>])
```

Parameters

<lane> is the lane to drive in, either a single integer value (reals are rounded) 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.

<av-side> is **right** or **left**.

<event> is **start**, **end** or **all**. The default is **all**, meaning that the specified speed is maintained throughout the current period.

Description

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

```
do serial:
    car1.drive(path) with:
        # Drive in left-most lane
        lane(1)


do parallel:
    car2.drive(path)
    car1.drive(path) with:
        # Drive one lane left of car2
        lane(left_of: car2)


do parallel:
    car2.drive(path)
    car1.drive(path) with:
        # At the end of this phase, be either one or two lanes
        # to the right of car2
        lane([1..2], right_of: car2, at: end)


do parallel:
    car2.drive(path)
    car1.drive(path) with:
        # Be either one left, one right or the same as car2
        lane([-1..1], right_of: car2)


do parallel:
    car2.drive(path)
    car1.drive(path) with:
        # Be in the same lane as car2
        lane(same_as: car2)
```

14.9. lateral

Purpose

Set location inside the line along the lateral axis.

Category

Scenario modifier

Syntax

```
lateral([distance: <distance>][line: <line>][at: <event>])
```

Parameters

<distance> is the offset from reference line. The default is [-0.1..0.1]meter.

<line> is the reference line the offset is measured from, either **right** (the right side of the car), **left**(the left side of the car) or **center** (the center of the car). The default is **center**.

<event> is **start**, **end** or **all**. The default is **all**, meaning that the specified distance is maintained throughout the current phase.

Example

```
do serial:
    car1.drive(path) with:
        # Have that distance at the start of the phase
        lateral(distance: 1.5meter, line: right, at: start)
```

14.10. 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>], [a
t: <event>])
```

Parameters

<distance> is a single value or a range with a distance unit.

<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.

Description

The **position()** modifier lets you specify the position of an actor relative to the start of the path or relative to another actor. You can specify the position by distance or time (but not both).

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

```
do serial:
    car1.drive(path) with:
        # Absolute from the start of the path
        position([10..20]meter)


do parallel:
    car1.drive(path)
    car2.drive(path) with:
        # 40 meters ahead of car1 at end
        position(40meter, ahead_of: car1, at: end)


do parallel:
    car1.drive(path)
    car2.drive(path) with:
        # Behind car1 throughout
        position([20..30]meter, behind: car1)


do parallel:
    car1.drive(path)
    car2.drive(path) with:
        # Behind car1, measured by time
        position(time: [2..3]second, behind: car1)
```

14.11. 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. You must specify a speed unit.

<car> is the instance name 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.

Description

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

```
do serial:
    car1.drive(path) with:
        # Absolute speed range
        speed([10..20]kph)


do parallel:
    car1.drive(path)
    car2.drive(path) with:
        # Faster than car1 by [1..5]kph
        speed([1..5]kph, faster_than: car1)


do serial:
    car1.drive(path) with:
        # Have that speed at end of the phase
        speed(5kph, at: end)


do parallel:
    car1.drive(path)
    car2.drive(path) with:
        # 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, where one field is of type **path** and the other is of type **sub_path**. These constraints apply not to two separate paths, but to a path and a segment of that path.

When an MSDL tool choses a location on a map, it must take into account all the constraints associated with the path and select a random location (the path itself) out of all the appropriate locations in the map.

For example, if you specify a minimum of two lanes, only locations with at least two lanes are considered. If you require the car to reach 60kph and to abide the law, then only locations that allow a legal speed of more than 60kph are considered. If you require the car to reach 50kph at some point, then only paths that are long enough for the car to accelerate to that speed are considered.

15.1. path_curve

Purpose

Specify that the path has a curve.

Category

Scenario modifier

Syntax

```
path_curve([path: ]<pathname>, [min_radius: ]<radius>, [max_radius: ]<radius>, [side:
]<av-side>)
```

Parameters

<pathname> is the name of a path instance in the scenario.

<radius> is a value of type **distance**.

<av-side> is a value of type **av_side**, one of **right** or **left**.

Example

```
do serial:
    car1.drive(path1) with:
        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: ]<field-name>, [path2: ]<field-name>)
```

Parameters

<field-name> is the name of a field in the scenario. There must be two fields of type **path**.

Example

```
do serial:
    car1.drive(path1) with: 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: ]<field-name>, [path2: ]<field-name>)
```

Parameters

<field-name> is the name of a field in the scenario of type **path**. There must be two fields.

Example

```
do serial:
    car1.drive(path1) with: 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: ]<field-name>, [requests: ]<list of point>, tolerance: <toleranc
e>)
```

Parameters

<field-name> 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 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("hooder.xodr")
        path_explicit(a.path1,
          [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: ]<field-name>, [path2: ]<field-name>)
```

Parameters

<field-name> is the name of a field in the scenario of type **path**. There must be two fields.

Example

```
do serial:
    car1.drive(path1) with: 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: ]<field-name>, [sign: ]<sign-type>)
```

Parameters

<field-name> 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
- roundabout

Example

```
do serial:
    car1.drive(path1) with:
        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: ]<field-name>)
```

Parameters

<field-name> is the name of a field in the scenario of type **path**.

Example

```
do serial:
    car1.drive(path1) with:
        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: ]<field-name>, [min_path_length: ]<min-distance>, [max_path_lengt
h: ]<max-distance>, [allow_junction: ]<bool>)
```

Parameters

<field-name> 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

```
do serial:
    car1.drive(path1) with:
        path_length(path: path1, min_path_length: 150meter,
          max_path_length: 175meter, allow_junction: 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: ]<field-name>, [max_lanes: ]<int>)
```

Parameters

<field-name> is the name of a field in the scenario of type **path**.

<int> is an integer value specifying the maximum number of lanes.

Example

```
do serial:
    car1.drive(path1) with:
        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: ]<field-name>, [min_driving_lanes: ]<int>)
```

Parameters

<field-name> 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

```
do serial:
    car1.drive(path1) with:
        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: ]<field-name>, [min_lanes: ]<int>)
```

Parameters

<field-name> is the name of a field in the scenario of type **path**.

<int> is an integer value specifying the minimum number of lanes.

Example

```
do serial:
    car1.drive(path1) with:
        path_min_lanes(path1, 2) # Needs at least two lanes
```

15.12. path_over_highway_junction

Purpose

Specify that the path pass through a junction on a highway.

Category

Scenario modifier

Syntax

```
path_over_highway_junction([junction: ]<field-name>, start_type: <road_type>, end_typ
e: <road_type>, distance_before: <distance>, distance_after: <distance>, distance_in:
<distance>, path: <path>)
```

Parameters

<field-name> is a field in the scenario of type **junction**.

<road_type> is a field in the scenario of type **road_type** or one of:

- unknown
- highway
- highway_entry
- highway_exit
- highway_entry_exit

<distance> is a value or a range with a distance unit.

<path> is a field of type **path** specifying the intersecting road at the junction.

Example

```
do serial:
    car1.drive(path1) with:
        path_over_highway_junction(junction: junction, start_type: highway, end_type:
highway_exit, distance_in:[5..10]m, path:path2)
```

15.13. path_over_junction

Purpose

Specify that the path pass through a junction.

Category

Scenario modifier

Syntax

```
path_over_junction([junction: ]<field-name>, [direction: ]<direction>, distance_befor
e: <distance>, distance_after: <distance>, distance_in: <distance>)
```

Parameters

<field-name> 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 car.traverse_junction:
    path1: path
    car1: car
    junction1: junction
    direction1: direction

    do serial:
        car1.drive(path1) with:
            path_over_junction(
              junction1,
              direction1,
              distance_before: [5..10]meter,
              distance_after: [5..10]meter)
```

15.14. 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: <field-name>, sp_more_lanes_path_length: <distance>, mo
re_lanes_path: <sub-path>)
```

Parameters

<field-name> 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 **sub_path**.

Example

```
scenario dut.scenario1:
    path1: path
    path1a: sub_path
    car1: car

    do serial:
        car1.drive(path1) with:
            path_over_lanes_decrease(path: path1,
              sp_more_lanes_path_length: 20meter,
              more_lanes_path: path1a)
```

15.15. path_over_speed_limit_change

Purpose

Specify that the path pass through a change in the speed limit.

Category

Scenario modifier

Syntax

```
path_over_speed_limit_change(path: <field-name>, path1_legal_speed: <speed>
    path2_legal_speed: <speed>)
```

Parameters

<field-name> is a field of type **path** specifying the intersecting road at the junction.

<speed> is a value or a field in the scenario of type **speed**.

Example

```
do serial:
    car1.drive(path1) with:
        path_over_speed_limit_change(path: path1, path1_legal_speed: 80kph, path2_lega
l_speed: 50kph)
```

15.16. paths_overlap

Purpose

Specify that two path instances must overlap.

Category

Scenario modifier

Syntax

```
paths_overlap([path1: ]<field-name>, [path2: ]<field-name>)
```

Parameters

<field-name> is the name of a field in the scenario of type **path**. There must be two fields.

Example

```
do serial:
    car1.drive(path1) with: paths_overlap(path1, path2)
```

15.17. path_same_dest

Purpose

Specify that two paths have the same destination.

Category

Scenario modifier

Syntax

```
path_same_dest([path1: ]<field-name>, [path2: ]<field-name>)
```

Parameters

<field-name> is the name of a field in the scenario of type **path**.

Example

```
do serial:
    car1.drive(path1) with:
        path_same_dest(path1, path2)
```

15.18. path_same_origin

Purpose

Specify that two paths have the same origin.

Category

Scenario modifier

Syntax

```
path_same_origin([path1: ]<field-name>, [path2: ]<field-name>)
```

Parameters

<field-name> is the name of a field in the scenario of type **path**.

Example

```
do serial:
    car1.drive(path1) with:
        path_same_origin(path1, path2)
```

15.19. set_map

Purpose

Specify the map used in the test

Category

Scenario modifier

Syntax

```
set_map([name: ]<string>)
```

Parameters

<string> is the name of a map for the test must be specified.

Description

The **set_map()** modifier accepts a filename as the only parameter.

Example

```
do serial:
    car1.drive(path1) with:
        set_map("hooder.xodr")
```

16. Change log

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

16.1. Version 0.9.1

The following changes appear in version 0.9.1:

- Expanded the description of list elements. See List types (page 25).
- Updated the syntax for **emit** to show that parameters must be passed by name, not position. See emit (page 94).
- Updated the description of **weather_type** and **time_of_day**. Added **road_type**. See Predefined AV types (page 27)
- Documented the map modifiers **path_same_origin()**, **path_over_highway_junction()** and **path_over_speed_limit_change()**. See Map modifiers (page 126).
- Identified the parameters that can be passed by position in builtin and library scenarios by enclosing the parameter name in square brackets in the Syntax section of each scenario description. For an example, see the path parameter for drive() (page 102)
- Clarified the difference between soft and default constraints. See Soft constraints and default constraints (page 67).
- Added a definition of *basic clock* and replaced all references to **top.clk** with basic clock. See Terminology (page 33)
- Noted that **cover()** definitions are not allowed in scenario invocations, including operator scenario invocations, or in scenario modifiers. **cover()** definitions are allowed in field declarations. See Scenario Invocation (page 75).
- Added **me** and **actor** to the list of predefined identifiers and added an example. See User-defined identifiers, constants and keywords (page 15).
- Documented the new scheme for implicit labels, which uses **label()**. See Automatic labels (page 77).
- Changed the **no_collides()** movement scenario modifier to **collides()**. See collides (page 116).
- Added a definition of “phase” as an informal term. See Terminology (page 33) and parallel (page 88).
- Restricted the use of “top-level scenario” to refer to **top.main**.

- Corrected the example for the use of **outer** in the **in** scenario modifier. See in modifier (page 106).
- Removed the defaults for the first parameter of **change_speed()**, **position()**, and **speed()**. Marked both parameters of **change_lane()** as optional. See Movement-related modifiers (page 112).
- Added a physical unit for speed, **meter_per_second** or **mps**. See Physical types (page 23).
- Documented the default file extension as **.sdl** and the use of the SDL_PATH environment variable. See M-SDL file structure (page 20).
- Documented the **name** option of **cover()**. See cover (page 50).
- Documented the **lateral()** movement modifier. See lateral (page 120).
- Clarified the syntax for declaring and calling an external method. See external method declaration (page 56).
- Simplified the syntax for **with** blocks in scenario and scenario modifier invocation. Passing **with** members as parameters is no longer allowed for scenario and scenario modifier invocation. See drive (page 102), Operator scenarios (page 79), Scenario invocation (page 75), and Scenario modifier invocation (page 71).
- Added the physical types **temperature** and **weight**. See Physical types (page 23).
- Clarified that empty lines and single-line comments do not require a specific indentation and modified the description and example of multi-line comments. See Lexical conventions (page 12).
- Clarified that in **when** subtype declarations, the value specified for a when determinant field must be a constant. Also clarified the syntax for applying constraints to fields in a **when** subtype. See when subtype (page 68).
- Updated the list of keywords. See User-defined identifiers, constants and keywords (page 15).
- Clarified that parentheses are not allowed in scenario declarations, but they are required in all scenario invocations, except operator scenario invocations. Changed code examples accordingly. See scenario (page 45) and Scenario invocation (page 75).
- Clarified the description and example of global actors. See Actor hierarchy and name resolution (page 21).
- Added a definition of “path expression”. See Terminology (page 33).
- Added **sample()** to examples that show how to sample a field at a specified event. See field (page 58).
- Added a note that enclosing parentheses for boolean expressions are optional for the

if operator. See if (page 81).

- Clarified that identifiers beginning with an underscore character are not allowed. See User-defined identifiers, constants and keywords (page 15).
- Clarified that physical expressions, such as **start_speed-1kph** are allowed in ranges. See Physical types (page 23).

16.2. Version 0.9

This version includes minor edits.

16.3. 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.

A. M-SDL Frequently Asked Questions

A.1 Preface

The evolution of M-SDL is influenced by ongoing interaction with professionals from standard bodies, the car industry, transportation companies, academia and other interested parties. These interactions highlighted some aspects of the language that require further clarification and sometimes refinement.

This appendix provides clarifications and more complete examples to explain some of the features *included in this version of the language manual*. Notably:

- Modeling techniques using scenarios, modifiers and events
- Checking, especially temporal checking using scenarios as monitors
- Clarifying the usage of fields and modifiers

In addition, the appendix presents some novel concepts that are discussed here for the first time. These *differ from the features described in this version of the language manual*. They are clearly marked by "*not in v0.91*". The new concepts include:

- Capturing Key Performance Indicators
- A new way to model map topology
- A refined version of type inheritance and extension

Details of the above features are still being worked out. As they mature, they will be introduced into the reference manual.

A.2 Questions about language behavior

A.2.1 What happens if a scenario cannot be executed

Question: What happens if a scenario cannot execute as specified? For example, what happens if the duration of a drive is not enough to reach the required speed?

```
drive(duration: 1s) with:
  speed(0kph, at: start)
  speed(150kph, at: end)
```

Users often write contradicting constraints or modifiers. This results in a contradiction, usually causing a scenario failure and a corresponding error message.

A contradiction can be a logical contradiction when, for example, you say **keep(color==green)** in one place, and elsewhere you say **keep(color == blue)**. Or it can be that some requested parameters contradict physics when, for example, the specified time, speed and distance contradict the implicit constraint connecting them.

Or it can be that some modifiers cannot be satisfied because of some aspect of the detailed physical simulation or because of **ego** (DUT) behavior. In all of these cases, the scenario fails with an error, usually stopping or not even starting the run.

There is a **try** operator if you want to do something else when a run-time scenario failure occurs.

A.2.2 How to define ego is stopped for a period of time

Question: How can I define a scenario where the **ego** is stopped for a period of time, such as 1 to 10 seconds?

You can simply write:

```
drive(duration: [1..10]s) with: speed(0kph)
```

Consider the following example:

```
scenario car.stop_and_go:
  do serial:
    a: drive() with: speed(100kph, at: start)
    b: drive(duration: [1..10]s) with: speed(0kph)
    c: drive() with: speed(80kph, at: end)
```

In phase **b** the car is fully stopped. This means that in phase **a**, it slows down from 100kph to a halt, and in phase **c** it accelerates to 80kph.

A.2.3 How to unify actors across scenarios

Question: Suppose I define **cut_in** and **cut_out** scenarios separately, each having a **car** actor. If I compose them together in one scenario, **cut_in_and_out**, then I would expect the scenario generator to generate two different car instances. How can I tell the generator to use the same car instance in the nested scenarios?

The simplest way to do that is to define a single **car** actor and make both scenarios use it. One way to do this is by parameter passing. For instance, if both scenarios have a parameter **v1: car,** you can write:

```
scenario top.together:
  c: car
  do serial:
    cut_in(v1: c)
    cut_out(v1: c)
```

Or alternatively you can write equality constraints, making the two cars the same:

```
scenario top.together_2:
  do serial:
    c1: cut_in()
    c2: cut_out()
  keep(c1.v1 == c2.v1)
```

A.2.4 How to refer to different objects with the same name

Question: Suppose I have scenario **A** with a parameter **v1** and it invokes scenario **B**, which has a parameter with the same name. How do I refer to them correctly?

In M-SDL, there is no confusion between the **v1** of the invoked scenario and the **v1** of the invoking scenario because in constraints, the invoked scenario is referred to as **it**. For instance:

```
scenario s:
```

```
v1: car
do serial:
  …
  overtake() with: keep(it.v1.color != v1.color)
      # it here is the overtake scenario
```

A.3 General modelling questions

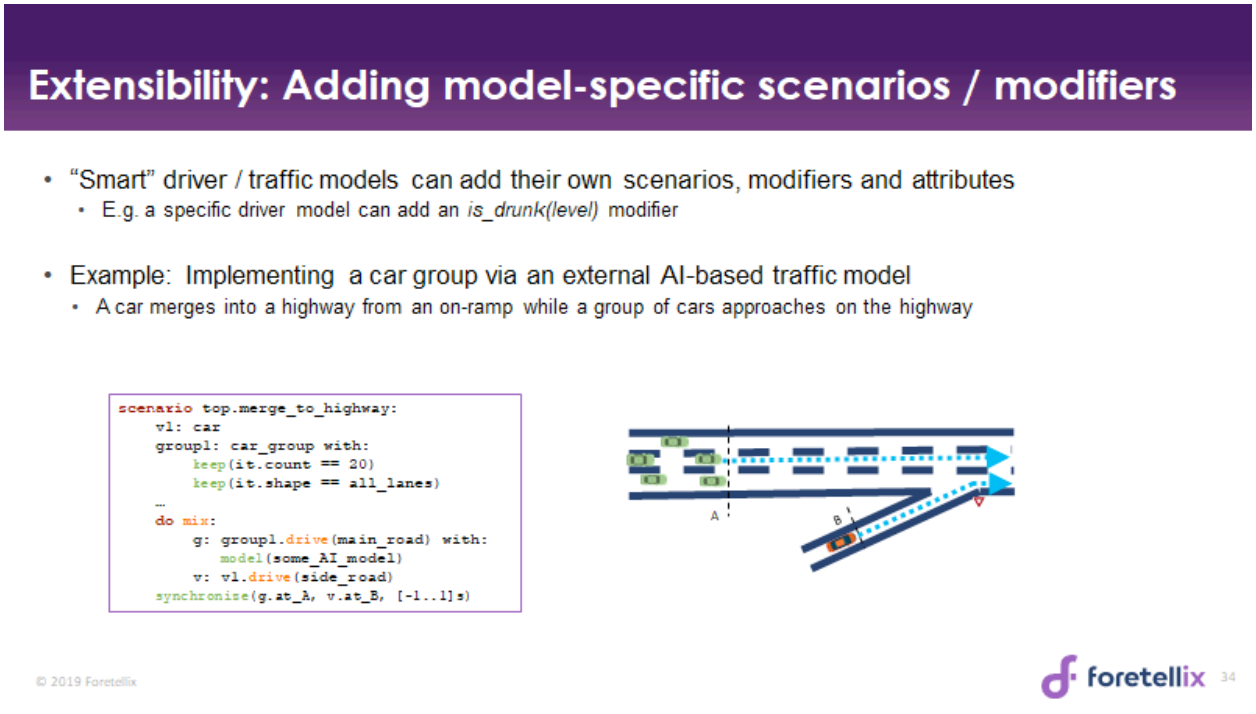
A.3.1 How to specify a variable number of actors

Question: How can I specify a variable number of actors, for example between 1 and 10 pedestrians crossing a crosswalk? I don't want to specify 10 separate cases with varying numbers of actors.

The best way to do that is by defining a new actor group, where the group is itself an actor. For instance, create a **car_group** actor to encapsulate a group of cars, and a **person_group** actor to encapsulate a group of pedestrians.

These group actors can also be connected to external models representing group behaviors, for example a machine-learned traffic model.

The following slide gives a hint at how to use group actors.



You can also create multiple unique actors using a list, as in the following field declaration:

```
persons: list of person
```

(The list syntax is not included in the v0.9 manual.)

However, we currently assume the first solution will be more popular.

A.3.2 How to specify multiple cases in a scenario

Question: How can I model a scenario where either a pedestrian or a bicycle crosses the crosswalk?

There are two ways to do that:

- Using the **one_of** operator.
- Using conditional inheritance.

Using one_of:

The **one_of** operator lets you choose between multiple, completely different behaviors. Using **one_of**, you can write the scenario as follows:

```
scenario traffic.somebody_crossing_1:
  dut_path: path
  crossing_path: path
  … # Define the two paths correctly

  do mix:
    dut.car.drive(dut_path)
    one_of:
      pedestrian_crossing(the_path: crossing_path)
      bicycle_crossing(the_path: crossing_path)

scenario traffic.pedestrian_crossing:
  p: person
  the_path: path

  do p.move() with: path(the_path)

scenario traffic.bicycle_crossing:
  b: bicycle
  the_path: path;

  do b.drive() with: path(the_path)
```

Using conditional inheritance:

Section A.8, Types, inheritance and related topics describes how to define conditional subtypes that depend on attributes of the parent type. This feature lets you define a hierarchy of subtypes, and still generate any item in an inheritance hierarchy, either randomly or using constraints.

Note: This feature (not in v0.9) is an improved version of the v0.9 **when** inheritance.

With conditional inheritance, you can define an actor hierarchy such as:

- moveable # Any moveable actor
  - person
  - animal
  - vehicle
    - car
    - truck
    - scooter
    - bicycle

  …

In addition, you can use inheritance to define scenarios within an actor. Thus, each moveable actor has a **move()** scenario, but the vehicles also inherit from **move()** to create the **drive()** scenario, and persons also inherit from **move()** to create **walk()**, **run()** and **crawl()**.

With that brief introduction, you can now define:

```
scenario traffic.somebody_crossing_2:
  dut_path: path
  crossing_path: path
  … # Define the two paths correctly

  # Define "m" to be any moveable, and then constrain it to just a
  # person or a bicycle. Otherwise it may also be randomized to be
  # a scooter, a truck etc.
  m: moveable with:
    keep(it.is(person) or it.is(bicycle))

  do mix:
    dut.car.drive(dut_path)
    m.move(crossing_path)
```

Note: Use **one_of** when you want to choose among multiple, unrelated scenario invocations. Use conditional inheritance to write a scenario invocation once, and then choose randomly some actor from the inheritance tree.

A.3.3 How to refer to the rise/fall/first-time of events

Question: In one example, the event **too_close** is defined by a condition that may be evaluated to true or false. But there may be different interpretations of when such event occurs, including:

- Only when the condition is true for the first time
- At each point in time where this condition is true
- At each point in time where this condition is true and was not true the point in time before.

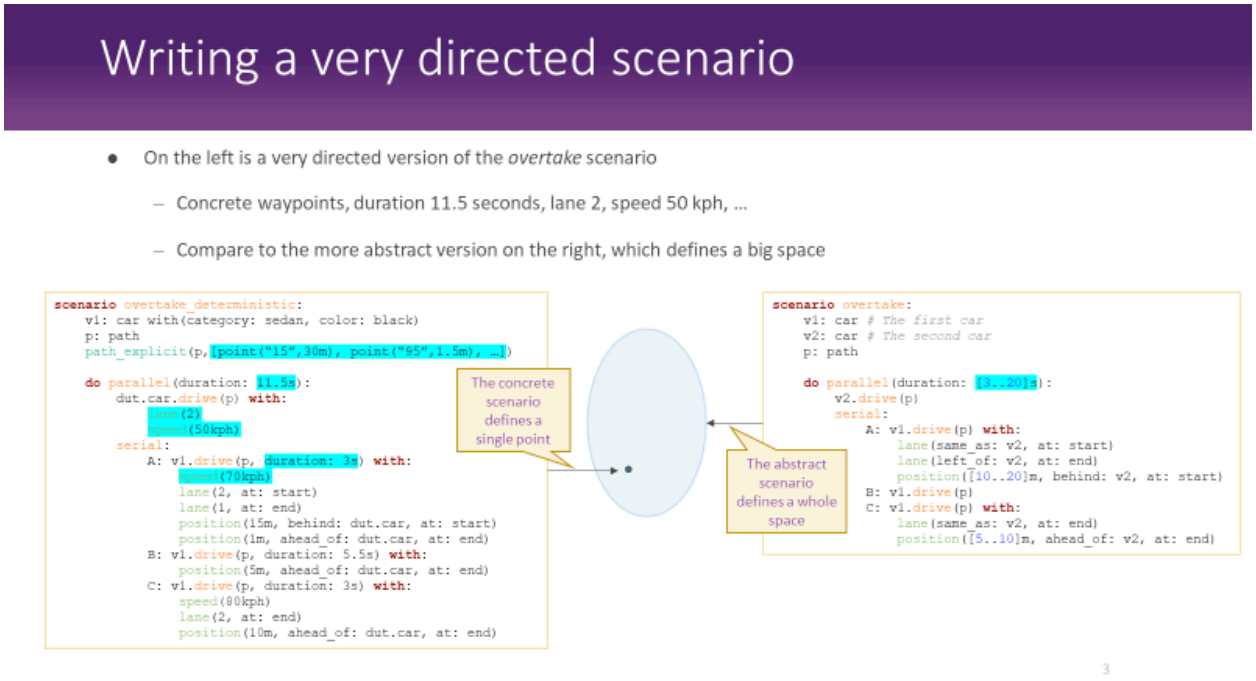
The v0.9 event interpretation is at each point in time where this condition is true. The following are extended qualified-event capabilities (not in v0.9):

- **first(<qualified-event>)** happens only at the first time this qualified event happens in the lifetime of the surrounding context.
- **rise(<bool-exp>)** happens if the expression is true now but was not true in the previous occurrence of that event.
- **fall(<bool-exp>)** happens if it is not true now but was true in the previous occurrence of that event.

A.3.4 Very directed scenarios, concrete scenarios and final values

Question: What is the difference between these three?

M-SDL scenarios can be very abstract or very directed. Very directed scenarios specify almost everything explicitly. For instance, see the example on the left below.



After generating either an abstract or a directed scenario, the result is a *concrete* scenario, where every parameter has a value that obeys all constraints and value specifications. Some people also call a very directed M-SDL scenario a concrete scenario.

In any case, there is an even more concrete scenario than that: what happened in an actual run.

In very simple, sterile cases, the two are one and the same. However, in many realistic cases, the two will differ. Here are two examples:

- Suppose the scenario specified that **car1** arrives at a junction within [-1..1]second of the **ego** car. Some concrete scenario is created, and then the system starts to run according to it. But if suddenly the **ego** car starts going faster or slower, **car1** may need to speed

up, thus potentially changing various parameters, as long as they remain within constraints.

- Suppose the scenario requested on-the-fly generation depending on the result of some previous operation, where that result may depend on partially unpredictable factors such as the behaviour of the **ego**. Then, the initial concrete scenario only contains the initial parameters; other parameters are made concrete at run-time.

In both cases there is probably a need to send on-the-fly changes or corrections to the simulation engine via an API.

A.4 Topology-related questions

A.4.1 How to specify the details of an intersection

Question: How can I specify the shape of the intersection, such as how many roads meet at the intersection, with how many lanes, at which angle? How can I specify the position of **ego** with respect to the intersection, for example, 10 meters before the intersection?

Topology in M-SDL is under revision. The new direction, currently referred to as the *new topology*, is the replacement of the current **path_*()** constraints. After a brief introduction to the new topology, the above question is answered.

About the new topology:

Customers have extremely diverse needs for referring to features related to topology and static scenes. These features have various levels of complexity and sometimes exist only in customer-specific maps.

The new topology (not in v0.9) works as follows:

- Users can define a variety of road elements with arbitrary attributes. Road elements are structs inheriting from **road_element.**
- Road elements typically represent segments of a road. The path of a car, for example, is specified as a sequence of road elements.
- If some road elements only appear in custom maps, not in OpenDrive maps, users can read in instances of these road elements from the custom maps and associate them with areas in the base map. The base map is usually an OpenDrive-derived map.
- The basic library contains typical road elements, such as roads, lanes, and junctions with typical attributes, such as length, width, and number of entries. These elements are read in using the same mechanism.

Road elements in a basic library can be extended by expert users. Then scenario writers can just refer to those road elements in the **path()** modifiers of **drive()** scenarios, for instance.

Although this FAQ talks about finding locations on an existing map, M-SDL can also allow the creation of imaginary maps according to path requests, if such a map generator is available.

Examples:

Assume we have already defined the following road element types:

- **road**
- **path_by_forest**: a stretch of road going along a forest
- **rough_surface**: a stretch of road going along a rough surface

Assume also that we already know how to read in the related information from various maps so that the road roughness comes from an OpenCRG file, for example.

Then we can define in a scenario the following fields:

```
r: road with: keep(it.lanes > 2)
f: path_by_forest with: keep(it.density == high)
s: rough_surface with: keep(it.roughness > 7)
```

We can then specify the road element types in **path()** modifiers inside the **do** part of the corresponding scenario:

```
drive() with:
  path(r) # Drive along the multi-lane road chosen for field r

drive() with:
  path(f) # Drive along a dense forest
  path(s) # It should also be on a rough surface
  path(r: at: start) # At the start of this phase, also be in this
road
```

Similarly, assume that we have a **junction** road element, defined by an expert user, that has a list of entry road elements with the right attributes as described below:

```
struct junction: road_element:
  …
  entries: list of junction_entry

struct junction_entry: road_element:
  lanes: uint   # Number of lanes
  angle_to_next: angle   # Angle to next entry (clock-wise)
  road: road
```

Note: The syntax **struct <name>: <parent-name>:** (not in v0.9) is the new syntax for simple (unconditional) inheritance. It replaces the v0.9 **like** inheritance syntax. See also Section A.8.1, M-SDL inheritance recap.

Then the scenario writer can write something like:

```
# Define the junction we want
j: junction
keep(j.entries.size() >= 5)  # Has at least 5 entries
keep(j.entries.filter(it.lanes > 1).size() > 3)
    # Has more than three multi-lane entries
keep(j.entries.filter(it.angle_to_next in [5..15]deg).size() >2)
    # Has more than two low-angle-to-next entries
```

```
# Define the entry we want to that junction
e: junction_entry
keep(e in j.entries)  # It is one of the junction's entries
keep(e.angle_to_next in [10..15]deg)
       # It is one of those low-angle-to-next entries
```

Note: The **<list>.filter(<bool-exp>)** list method (not in v0.9) returns a list of the items in the original list that satisfy the Boolean expression.

We can then use the defined junction in the **do** part of the scenario:

```
car1.drive() with:
  path(e, at: end)  # It is in e at the end of the phase
  position(10m, behind_point: j.center, at: end)
      # It is 10 meters from the center of j at end of phase
```

A.4.2 How to specify direction

Question: How can I define the direction of another actor relative to the **ego** vehicle, such as "between 10 and 45 degrees"? For example, how can I define an intersecting cut-in where another actor emerges from a side street or exits from a parking lot and then cuts in in front of the **ego** vehicle?

The modifier for that is **direction()** (not in v0.9). The above scenario would be written like:

```
scenario dut.intersecting_cut_in:
  car1: car
  … # more about the setting

  do mix:
    dut.car.drive()
    car1.drive() with:
      direction([10..45]deg, relative_to: dut.car)
```

direction() lets you specify an angle in degrees or radians as well as the reference point relative to which the angle is measured. There are various ways to specify that reference:

- **relative_to:** <actor> is relative to the direction of that actor, as in the example above.
- **relative_to_north** is relative to the absolute north.
- **relative_to_lane** is relative to the direction of the lane at my current location, with **me** being the car doing the driving.
- **relative_to_path:** <path> is relative to that path.
- **relative_to_direction_to:** <actor> is relative to the line-of-sight from **me** to that actor.
- **relative_to_direction_to_point:** <point> is relative to the line-of-sight from **me** to that point.

In all of these, **0deg** means the same direction as the other point of reference, **90deg** means coming from the right, and so on.

A.4.3 What position(ahead_of: ...) means

Question: Does **ahead of** mean distance along a path, or just longitudinal distance?

ahead of distance always means along a path. For example **position(200m, ahead_of: car1)** means 200 meters ahead of **car1** along the path.

A.5 Coverage, KPIs, checking and scenario failure

Please see this post for an overview of coverage and performance metrics and how they are different. They are used for two different things:

- Coverage evaluates which part of the scenario space have we exercised our AV in. This is expressed via coverage and an overall coverage grade.
- Performance evaluates how well the AV behaved in these scenarios. This is expressed via raw KPIs (Key Performance Indicators) and context-dependent performance grades.

The rest of this section discusses:

- What is coverage and how to capture it for subsequent analysis
- What are KPIs and how to capture them for subsequent analysis
- How to check for DUT (**ego**) errors and scenario failures

A.5.1 How to capture coverage

Coverage, especially scenario functional coverage, measures what we have tested so far: which scenarios were actually exercised, and, for each such scenario, what were the values of its various parameters.

When you define coverage, you can specify what the coverage items (parameters) are, into which buckets (groups of values) to split each item, which items to cross, and what weight to give each bucket. This is all captured in a top-down verification plan, which describes what needs to be tested.

The **cover()** construct defines a coverage item. The first parameter is the expression to be covered. There are also several optional parameters like the descriptive text of that item, the range of values and how to split it into buckets, the event (group) to which this item belongs and so on.

Here is an example:

```
scenario dut.cut_in_and_slow:
  side: left_or_right  # Did the car cut in from left or right
  …
  cover(side) # Cover the value of side (2 buckets)

  # Sample the ego speed at the end of the slow phase
  !dut_v_slow_end: speed = sample(dut.car.speed,  @slow.end)

  # Cover it in kph, only when in [0..200] kph, and split into
  # buckets of 10kph each (20 buckets)
```

```
    cover(dut_v_slow_end, unit: kph, range: [0..200], every: 10,
        text: "Speed of dut at slow end (in kph)")

    # Cross the side and the ego speed (2 * 20 buckets)
    cover([speed, dut_v_slow_end])
```

A.5.2 How to capture KPIs

Coverage is captured via the **cover()** construct (see above). To record non-coverage metrics such as Key Performance Indicators (KPIs), use the **record()** construct (not in v0.9).

record() is similar to **cover()**. It specifies the expression to record, an optional name and descriptive text, the event on which to capture it (by default: the end event of the scenario) and so on.

Consider the following code snippet:

```
scenario dut.cut_in_and_slow:
  …
  # Sample the time-to-collision KPI at the end of change_lane
  !ttc_at_end_of_change_lane:=
      sample(dut.car.get_ttc_to(v1), @change_lane.end)

  # Record the KPIs into the cut_in_and_slow.end metric group
  record(ttc_at_end_of_change_lane,
      text: "Time to collision of ego car to cut-in car " +
      "at end of the change_lane phase")
```

This value is then recorded into that metric group, making it available for later multi-run analysis.

The name coverage group is replaced with the name metric group, since such a group, like **cut_in_and_slow.end**, can contain both coverage and performance metric items.

The standard library will include a set of predefined API methods like the above **get_ttc_to()**. Users can add their own using the **def** construct as well, of course.

In addition to various per-scenario metric groups like the one above, it makes sense to create global metric groups. Here are some recommended metric groups:

- A metric group called **top.end_of_run**, sampled once at the end of the run. It should contain various metrics for the whole run, such as **min_ttc**, **max_deceleration** and so on.
- A metric group called **top.collision**, sampled upon every collision. It should contain various metrics for that collision, such as whether the DUT car was involved, the relative speeds at the time of collision and so on.

The various metric items can then be aggregated and analyzed offline.

A.5.3 How to check for DUT errors and scenario failures

DUT errors vs. scenario failures

DUT errors are different from scenario failures:

- A DUT error means the DUT (**ego**) did something wrong.
  - You indicate that by calling the zero-time scenario **dut.error().**
  - This usually indicates a bug or problem in the DUT that needs to be fixed.
- A scenario failure means that the scenario did not happen according to its definition.
  - You may indicate that by calling the zero-time scenario **fail()**, but most often scenario failure is detected automatically, for instance in the case where one of the required parameters, like speed, could not be reached.
  - A scenario failure sometimes indicates a bad scenario definition, for instance missing constraints.
  - It may also indicate a scenario that is hard to achieve, so you may need to run it multiple times to increase the probability of it happening.

The three main ways to check

Checking for DUT errors and scenario failures are usually done in one of three ways:

- Checking a condition upon some event.
  - This is done using the **on** modifier.
  - This is the most common simple way, and it is described below.
- Checking a condition inline inside a scenario execution.
  - This is possible but less recommended, because it is harder to change from the outside the scenario.
- Using the **match** operator, as described below in Section A.6, Using match to do temporal checking.
  - This is used for doing temporal checks, such as “during scenario X, the **ego** should first do Y and then Z, else it is a DUT error”.

Checking upon an event occurrence

The following is an example of checking upon an event occurrence. DUT errors are often tied to a KPI, such as checking whether that KPI crossed some error threshold. For instance, in **cut_in_and_slow** scenario above, we may want to say:

If the TTC at end of **change_lane** was smaller than 0.7s, then this is a DUT error, meaning the **ego** should have braked earlier.

Here is how you say that:

```
scenario dut.cut_in_and_slow:
  …
  # Sample the time-to-collision KPI at the end of change_lane
  !ttc_at_end_of_change_lane:=
      sample(dut.car.get_ttc_to(v1), @change_lane.end)

  # Record the KPIs into the cut_in_and_slow.end metric group
  record(ttc_at_end_of_change_lane,
```

```
        text: "Time to collision of ego car to cut-in car " +
        "at end of the change_lane phase")

    # Upon the end event, check that it is below the threshold,
    # else issue DUT error
    on @end:
      if ttc_at_end_of_change_lane < 0.7s:
        dut.error("Ego too close to other car at end of " +
            " change_lane. " + "TTC: $(ttc_at_end_of_change_lane)")
```

Notes:

- The **sample(<exp>, <qualified-event>)** notation samples the value of the expression at the time the event happened. See page 51 in the v0.9 manual.
- The **on** modifier often appears in a separate extension of **cut_in_and_slow** in another file, in order to facilitate separation of aspects.
- Note that external methods such as those in C++ / Python can also raise events. This lets you put the heavy-lifting checking code in an external method and have it raise an event upon error.
- There should probably be tool-specific ways to specify what happens upon each DUT error such as stop the run, notify and continue, treat as warning and so on, as well as tool-specific multi-run analysis capabilities. Both are outside the domain of the language itself.
- A standard library will probably contain a set of basic checks for most simple cases, such as collision, almost-collision, braking-too-hard and so on.
- To cause a scenario failure rather than a DUT error simply use **fail()** rather than **dut.error().**

For more complex checks that require tracking activity over time, the on modifier is not enough. Such checks require using the match operator.

A.6 Using match to do temporal checking

As discussed in the previous section, simple checking can be done using events and the **on** modifier. Sometimes, however, the conditions for checking involve tracing a sequence of occurrences, known as temporal checking. This capability is provided by the **match()** operator. **match()** lets you express both scenario failures and DUT errors under complex conditions. For instance, you may want to say "during X, first Y should happen and then Z, else this is a DUT error" (or a scenario failure).

The following examples address some common cases.

A.6.1 Example 1: Checking inside the same scenario

Consider the following:

- In phase1, the **ego** is driving behind **v1** at certain speeds and distances
- In phase 2, **v1** slows to much below the allowed speed. Check that **ego** still drives behind it, and maintains legal distance.

Here is how to write this in M-SDL:

```
scenario dut.example_1:
  v1: car
  do serial:
    phase1: parallel:
      a: v1.drive() with:  speed([90..100]kph)
      b: dut.car.drive() with:
        speed([90..100]kph)
        lane(same_as: v1)
        position(time: [2..3]s, behind: v1)
    phase2: parallel(duration: [3..7]s):
      c: v1.drive() with: speed([50..70]kph, at: end)
      match:
        m: dut.car.drive() with:
          lane(same_as: v1)
          position(time: [2..3]s, behind: v1)
      else:
        dut.error("Ego did not keep distance")
```

The **match** syntax (section 8.3 in the v0.9 manual) has an optional **then** part, executed if the match succeeded, and an **else** part executed if the match failed. The match body, marked by the label **m** here, is simply a scenario invocation like any other, except that here it is executed in the passive interpretation, meaning it is being monitored to see if it happened.

The **match** operator lets you decide how to respond to a failure, while code outside **match** simply causes the scenario fail if it did not happen. For instance, compare the following snippet:

```
do serial:
  x
  y
  z
```

with this one:

```
do serial:
  x
  y

  match:
    z
  else:
    dut.error("…")
```

Suppose we are monitoring what happened, using the passive interpretation of the language. And suppose **x** and **y** happened but **z** did not. Then:

- According to the first snippet, the scenario simply did not happen.
- According to the second snippet, the scenario happened, and there is a DUT error.

A.6.2 Example 2: Checking inside the same scenario (narrower check)

In reality, the **ego** could have decided to do multiple things in phase 2 of the example above, such as slow down, move to the right lane and perhaps other options.

In example 1, we said "check that it slowed down". In example 2, we want to say "in the scenario where it stays in the same lane as v1, check that it slowed down". Here is how to write that:

```
scenario dut.example_2:
  v1: car
  do serial:
    phase1: parallel:
      a: v1.drive() with:  speed([90..100]kph)
      b: dut.car.drive() with:
        speed([90..100]kph)
        lane(same_as: v1)
        position(time: [2..3]s, behind: v1)
    phase2: parallel(duration: [3..7]s):
      c: v1.drive() with: speed([50..70]kph, at: end)
      d: dut.car.drive() with:
        lane(same_as: v1)
      match:
        m: dut.car.drive() with:   # This is what we check
          position(time: [2..3]s, behind: v1)
      else:
        dut.error("Ego did not keep distance")
```

The **drive()** invocation labeled by **d** was added to ensure the **ego** remains in the lane. If **d** does not happen, meaning the **ego** does not stay in the same lane as **v1**, then scenario **dut.example_2** did not happen, and we are not going to check anything or issue any DUT errors.

A.6.3 Example 3: Checking from outside

Suppose we want to first define a "car ahead slows down" scenario with two vehicles **v1** and **v2**. Then we say from outside the scenario that **v2** is the **ego** and that we want to check it, as in example 2. Here is how to do that:

```
# This is the scenario we want to check
scenario traffic.front_car_slows:
  v1: car
  v2: car
  do serial:
    phase1: parallel:
      a: v1.drive() with:  speed([90..100]kph)
      b: v2.drive() with:
        speed([90..100]kph)
        lane(same_as: v1)
        position(time: [2..3]s, behind: v1)
    phase2: v1.drive(duration: [3..7]s) with:
      speed([50..70]kph, at: end)
```

```
# This scenario does the checking from above
scenario dut.example_3:
  front_car: car
  do parallel:
    f: traffic.front_car_slows(v1: front_car, v2: dut.car)
    match: # Invoke this as a sub-scenario
      wait @f.phase1.end # Wait until the end of phase1 (of f)
      m: dut.car.drive() with:
        lane(same_as: v1)
        position(time: [2..3]s, behind: v1)
    else:
      dut.error("Ego did not keep distance")
```

In example 3, we invoke the **front_car_slows()** scenario, and in parallel do a **match**. We synchronize the behavior we want to check to the right place using **wait**.

The **else** part is executed the first time the match fails, and you can refer to the state of the world at that time. For instance, you can change the **dut.error()** message to:

```
      dut.error("Ego did not stay behind at right distance at " +
              "time $(top.get_time()), " +
              "distance was $(dut.car.distance_to(front_car))")
```

Assuming of course that these methods exist and have the appropriate behavior.

A.6.4 Example 4: Checking from outside for two possible behaviors

Suppose we really want to say: when **v1** slows down, then the **ego** should either slow down behind it or move to the left. Here is how we say that, assuming **front_car_slows()** stays as is:

```
scenario dut.example_4:
  front_car: car
  do parallel:
    f: traffic.front_car_slows(v1: front_car, v2: dut.car)

    match: # Invoke this as a sub-scenario
      wait @f.phase1.end # Wait until the end of phase1 (of f)
      m: one_of:
        stay_behind: dut.car.drive() with:
          lane(same_as: v1)
          position(time: [2..3]s, behind: v1)
        go_left: dut.car.go_to_the_left_of(front_car)
    else:
      dut.error("Ego did not behave correctly when " +
                "front car slowed down")

scenario dut.go_to_left_of:
  v1: car
  do mix(inside)
    a: dut.car.drive()
    b: dut.car.change_lane() with:
      lane(same_as: v1, at: start)
```

```
      lane(left_of: v1, at: end);
    c: dut.car.drive() with:
      signal(left)
  synchronize(c.start, b.start, [-5..-3]s)
  synchronize(c.end, b.start, [1..5]s)
```

Example 4 also shows how to check the relations between several time intervals. In this case, we want to say that **change_lane** and the signaling both happened within the **go_to_left_of**, but also specify how their starts and ends synchronize (see the diagram below).

```
|------- go_to_left_of ----------|
       |--- change_lane ---|
   |--- signal ---|
```

A.6.5 Example 5: Making sure something did not happen

Question: How can I filter out cases where actors that should not exist in the scene are present, for example there should be no lead vehicle in front of **ego** at an intersection?

match allows both a **then** part and an **else** part, so you can specify for example:

```
car1: car
do parallel:
  # <the actual action, which does not involve car1>
  match:
    car1.drive([1..10]m, ahead_of: dut.car)
  then:
    fail("Can't have a car ahead of the ego at that point")
```

Keep in mind that **fail()** causes a scenario failure, not a DUT error, unlike most of the examples above. This means that whenever a car ahead of the **ego** is present, the scenario will be excluded; it will terminate without creating any coverage or error messages.

A.7 Parameters, variables and modifiers

A.7.1 What is the difference between parameters and variables?

There are three kinds of object fields:

- Parameters
  - Examples: the side of an overtake, the color of a car.
  - They do not have **!** (the do-not-generate attribute) in front of them.
  - They are assigned a value at the start of the scenario execution and never change.
  - They are influenced by constraints, such as **keep(car1.color != green).**
- Sampled variables
  - Example: The minimal time-to-collision during a scenario execution.
  - They have the syntax **!<name>: [<type>]=sample(<exp>, <qualified-event>).**
  - They may be updated multiple times during the run, depending on the sampling event.
  - They are used to hold KPIs and other computed values.
- State variables

- Examples: the current velocity of a car.
- They have the syntax **!name: <type>** and usually reside in an actor.
- They are updated automatically, possibly on every cycle, reflecting the actual state as it is received from the simulator, for example.
- They are influenced indirectly by modifiers, for instance **speed([1..5]kph, faster_than: car1)**. Modifiers constrain the permissible values of M-SDL actor fields, such as cars. These constraints should influence the behavior of the same actors in the simulation, and thus update the values captured in state variables.

Note: Constraints only influence parameters. They can also refer to variables, but those are considered input-only to constraint expressions.

A.7.2 What kinds of modifiers are there?

Modifiers in general influence the behavior of a scenario invocation, but do not take time by themselves. More precisely, they are stateless; there are no state variables associated with modifier instances. Like scenarios, they live within the namespace of the actor in which they are defined.

There are several kinds of modifiers:

- State modifiers, like **speed()**, **position()** and so on.
  - These constrain the state variables of the corresponding actor.
  - You can specify **at: start** or **at: end** to specify that the corresponding state variable constraints only apply at the start or end of the corresponding phase; otherwise, they apply throughout.
- Path modifiers, like **path_min_lanes()** or the new **path().**
  - These also constrain some location-related state variables, such as where the car is during the phase.
  - Like state modifiers, they also support **at: start** and **at: end**.
  - They have the indirect effect of choosing various road elements on the map.
- Event-related modifiers like **on**, **until** and **synchronize** use events to execute a block, exit the current invocation or achieve synchronization, respectively.
- The **in** modifier provides a way to insert a block of modifiers into an existing invocation.
- User-defined modifiers can contain anything that is legal in a scenario, except for the behavior (**do**) part. This, of course, includes modifier invocations.

Some modifiers can only be embedded in specific scenarios. For example, **lane()** must be embedded in a **drive()** scenario. To designate that association, use the **of** keyword (not in v0.9), as in the following user-defined modifier:

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

You can then use this new modifier like this, for example:

```
car1.drive() with: speed_and_lane(15kph, 1)
```

A.8 Types, inheritance and related topics

A.8.1 M-SDL inheritance recap

Modeling in M-SDL involves extensive use of object types, including structs to capture data, actors to represent actors in the environment and scenarios to capture actor behavior. These types are defined hierarchically, where more complex and specific types inherit from simpler and more general ones.

Syntax:

```
struct|actor|scenario <type-name>: <supertype-name>:
   <members>
```

Example:

```
struct junction: road_element:
   …
```

The above defines a new type **junction**, which is a subtype of **road_element. junction** inherits all the attributes of **road_element**, and may add new attributes as well as override some, such as external method bindings. This is called simple or unconditional inheritance, and it works like single inheritance in most OO languages.

When declaring a generatable instance of a type defined using simple inheritance, the instance is guaranteed to be of that type, and not any of its subtypes.

The syntax introduced here replaces the syntax of **like** inheritance in the v0.9 manual.

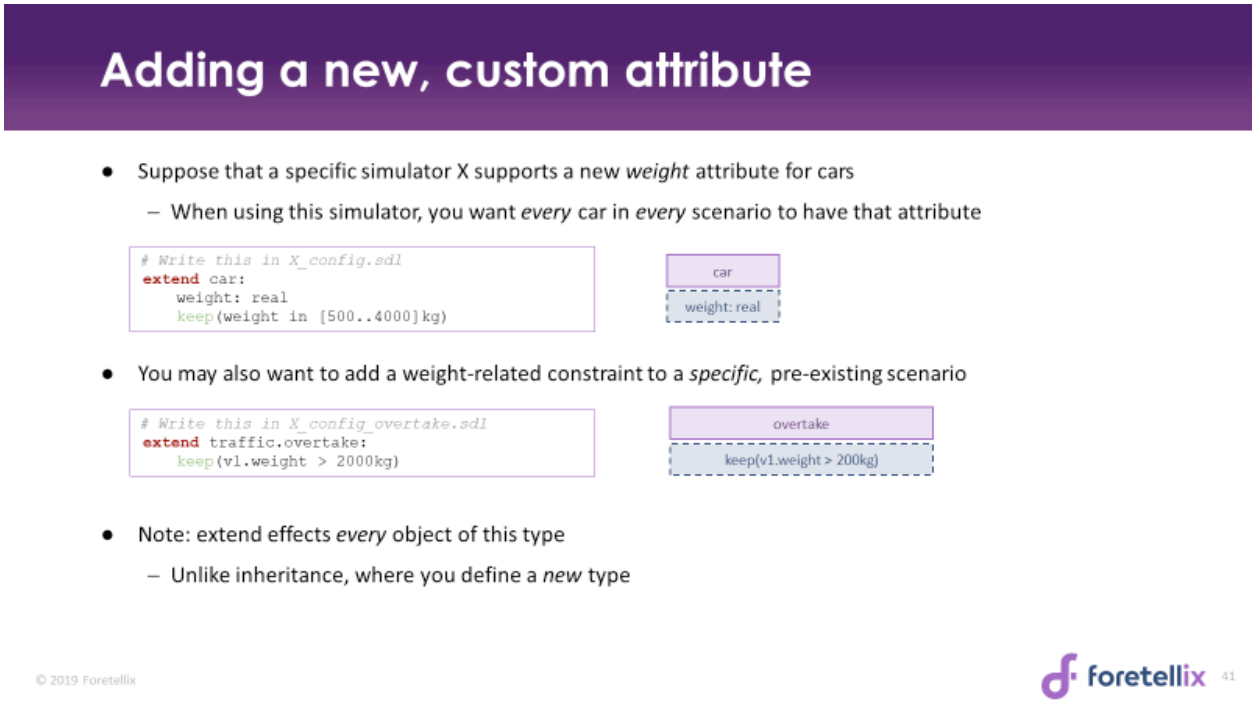
A.8.2 Extensibility

Verifying AVs requires considering specifics out of an infinite list of models, behaviors, environmental conditions and so on. They cannot all be in a standard library, however comprehensive. Standard library types can be reused by inheriting from them, but creating new subtypes for everything becomes unmanageable. The right approach is to have a reasonably comprehensive set of scenarios/actors/attributes, and a built-in mechanism for letting you extend those on demand.

That is why M-SDL provides a way to extend any type, including structs, actors, scenarios and enumerated types. (See section 4.2 of the v0.9 manual).

For instance, even if the **car** actor has all the reasonable predefined attributes, there are bound to be cases where you want to add more attributes such as fields, constraints, events and so on.

Consider the following example:



Unlike inheritance, where you define a new type, extension modifies the type being extended. All instances of the type get the newly added attributes. For instance, adding a weight attribute to **car** adds this to every **car** instance in every scenario.

Extending is particularly helpful when somebody else already created a library of inter-related actors and scenarios, and you want to:

- Add some new attributes because you are now using a new simulator that supports these, or because your project now suddenly cares about them.
- Add some new constraints because you cannot support anything outside these constraints.
- Add some new scenarios to existing actors or extend existing scenarios.
- Add more values to existing enumerated types.
- Add more method calls upon some event using the **on** modifier.

The **extend** feature lets you employ an aspect-oriented style of writing code, where each source file contains one aspect, but that aspect may need to touch multiple objects.

For instance, you may want to devote a file, **measure_comfort.sdl**, to just comfort-related checks. In that file, you:

- Extend various actors or scenarios and call some **save_comfort_data()** methods upon comfort-related events.
- Extend top and call a **summarize_comfort_data()** method upon the **run_end** event.
- Extend the DUT and define the related metric group.

Extension is a powerful mechanism that should be used in a disciplined way. You should organize extensions in a clear logical structure and group related extensions together.

A.8.3 Conditional Inheritance

When doing thorough verification, quite often you need to take into account multiple categorizations. For example, to name just a few categories, a vehicle can be:

- A truck, a car, a motorcycle and so on.
- An emergency-vehicle or non-emergency-vehicle.
- Single_track (like a motorcycle and a scooter), or multi_track.
- An electric vehicle or an internal-combustion vehicle.

This need is addressed by conditional inheritance – the ability to decide the type of an object conditioned on some of its field values.

To capture this, initially all we need to do is to add some attributes to vehicle:

```
actor vehicle:
  category: vehicle_category  # truck, car, motorcycle etc.
  emergency_vehicle: bool
  track_kind: track_kind  # single_track or multi_track
```

Not all combinations of these fields are legal. While any vehicle can be an emergency vehicle or not, only motorcycles and scooters, for example, can be **single_track**. To ensure that, we need to add the following constraint:

```
keep((track_kind == single_track) ==
    (category in [motorcycle, scooter]))
```

We can then specify in a scenario, for example:

```
v1: vehicle  # All these attributes are randomized
v2: vehicle with(category: truck)  # Could be emergency or not
v3: vehicle with(track_kind: single_track)  # Motorcycle/scooter
```

Using vehicle attributes to define multiple categories is a good start, but sometimes we need extra attributes that are applicable only to a specific kind (a specific value of an attribute). For instance:

- A truck has **num_of_trailers,** assuming for now that this is unique to trucks.
- An emergency vehicle has **emergency_vehicle_kind**, such as police, fire_brigade and so on.
- An electric vehicle has **remaining_charge**.

This is where conditional inheritance comes in.

Conditional inheritance is expressed using the following syntax:

```
struct|actor|scenario <type-name>: <supertype-name>(<field>:<value>):
   <members>
```

When declaring a conditional subtype, the condition always sets a single field to a specific value. Only Boolean and enumerated field values can be used in conditions.

For instance, you can define:

```
actor truck: vehicle(category: truck):
  num_of_trailers: uint with: keep(it in [0..2])
```

This defines a new actor type called **truck**, which is defined to be a vehicle when the category attribute has the value **truck**. Any vehicle that has **category == truck** automatically becomes of type **truck**. Thus, **truck** is a conditional subtype of vehicle. The expression **category == truck** is the condition determining if a vehicle is of type **truck**.

Now, if you write:

```
t: truck
```

When **t** is generated, it also gets 0 to 2 trailers randomly. If you write:

```
t: truck with(num_of_trailers: 1)
```

then **t** has exactly one trailer. If you write:

```
v: vehicle with(category: truck)
```

v is also a truck with 0 to 2 trailers, but you cannot access the **truck** specific features of **v**. If you write:

```
v: vehicle with:
  keep(it.category in [truck, motorcycle])
```

v is either a truck or a motorcycle. If it is a truck, it also has 0 to 2 trailers.

Conditional inheritance and the syntax introduced in this section replaces the concept of **when** inheritance in the v0.9 manual.

A.8.4 Scenarios and conditional inheritance

Suppose you define:

```
actor emergency_vehicle: vehicle(emergency_vehicle: true):
  emergency_kind: emergency_vehicle_kind
```

You can then define scenarios specific to **emergency_vehicle**, such as:

```
scenario emergency_vehicle.use_emergency_indicators:
  siren: bool  # Siren is turned on
  flashing_lights: bool  # Flashing lights are turned on
  …
```

You can now describe an emergency vehicle that drives while flashing its emergency lights:

```
e: emergency_vehicle
do parallel:
  e.drive() with: speed(50..70]kph)
```

```
e.use_emergency_indicators(siren: false, flashing_lighs: true)
```

Since the emergency vehicle **e** does not have a category defined, it could perhaps be a truck, in which case it also has 0 to 2 trailers. In general, a single object can belong to multiple conditional subtypes at the same time.

Because scenarios are typed objects themselves, they can be declared using conditional inheritance. Suppose, for instance, that a person has a **move** scenario, defined as follows:

```
scenario person.move:
  mode: movement_mode  # e.g. run, walk, crawl etc.
  …
```

You can then conditionally inherit from it, like this:

```
scenario person.run: move(mode: run):
  …
```

A.8.5 Using conditional inheritance

When working with conditional subtypes, checking the actual type of an object is sometimes required. In other situations, the actual type is known but is different from the declared type, so an explicit cast operation is required. These needs are addressed by the following expressions:

- The **is()** Boolean operator
- The **as()** cast operator
- The **let** modifier
- The **if** modifier

The is() Boolean operator

Syntax:

```
<path-to-object>.is(<type-name>)
```

Example:

```
if (v.is(truck)):
  …
```

is() returns **true** if the object is of the specified type, **false** otherwise. **is()** can be used in any context that accepts a Boolean expression.

If **v** is a truck, then the body of the **if** comes into effect. If **v** is not a truck, then the body of the **if** is skipped.

The as() cast operator

Syntax:

```
<path-to-object>.as(<type-name>)
```

Example:

```
keep(v.as(truck).num_of_trailers == 0)
```

as() performs a type cast, declaring the type of the object to be the specified type-name. If the object is generated, using **as()** implies a constraint on the object's type during generation. If the object cannot be of the casted type, an error is raised. A compile-time error is raised if the declared type contradicts the cast type; otherwise, a contradiction error is raised during generation.

There is a subtle difference between using **as()** and using a constraint on the condition field. Consider the following example:

```
v1 : vehicle
keep(v1.category == truck)
v2 : vehicle
keep(v2.as(truck).num_of_trailers == 0)
```

While both **v1.is(truck)** and **v2.is(truck)** return true, using cast in **v2** allows access to the truck specific features of **v2**. This often requires binding the cast result to a variable, so that the narrowed type can be accessed repeatedly. The **let** modifier described below addresses this need.

The let modifier

Syntax:

```
let [<var> := ] <expression> with:
  <members>
```

Example:

```
let t:= v.as(truck) with:
  keep(t.num_of_trailers == 1) # t here is a truck!
```

let binds the result of an expression to a variable, which is **it** by default but could be specified using the **:=** operator. The scope of the variable is the nested code block.

let is particularly useful when cast is performed, as it provides access to the result of the cast, exposing the subtype-specific features.

The if modifier

if was introduced as an operator scenario in v0.9 of the manual. It is also provided as a modifier here. Using **if** in combination with **is()** is useful for creating code related to conditional subtypes, as in the example below.

A.8.6 An example scenario

Let's see how conditional subtypes are used in practice. Consider the following **emergency_vehicle_approaching** scenario:

The **ego** is driving behind a vehicle **v**, which then slows down. At the same time, an emergency vehicle **e** approaches on the lane to the left. The **ego** has to decide whether to slow down or overtake **v**.

Here is how you can write it, assuming the mentioned sub-scenarios already exist:

```
scenario dut.emergency_vehicle_approaching:
  v: vehicle
  e: emergency_vehicle

  do mix:
    front_car_slows(v1: v)
    car_approaches_from_left(v1: e)
```

This may be a fairly important scenario to verify. The appearance of an emergency vehicle may influence the behavior of the **ego** in various ways. In fact, we would probably like to vary several scenario parameters, and cover them as well:

- The category of **v**, such as truck, car, motorcycle and so on.
- If a truck, how many trailers it has, because this may influence overtake decisions.
- The category of **e**.
- Relative speeds, distances and so on

Here is how you would specify some of these variations:

```
scenario dut.wrapper:
  do s: emergency_vehicle_approaching()
  in s:
    # Below are several options:
    # v should be a truck
    keep(v.is(truck))

    # v should be a truck with 3 trailers
    let v.as(truck) with:
      keep(it.trailers == 3)

    # If v is a truck, have 2 trailers
    if (v.is(truck)):
      let trk := v.as(truck) with:
        keep(trk.trailers == 2)

    # e should be a truck
    keep (e.is(truck))
```