foretellix

Measurable Scenario Description Language Reference

Version 20.07

Last generated: July 2020

Table of Contents

M-SDL Language Reference Manual

1. Introduction

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*. A scenario is a timed sequence of actions by one or more *actors*, such as cars, pedestrians, environmental conditions and the AV itself.

Using the Measurable Scenario Description Language (M-SDL), you identify actors and capture their behavior in scenarios. For simplification and productivity, M-SDL tools such as Foretify provide predefined actors, including the AV, other vehicles, a set of possible routes and the environmental conditions. They also provide basic scenarios that are associated with those actors, such as drive(), set_map(), and set_weather(). M-SDL libraries build on these basic behaviors to create more complex scenarios that describe, for example, another car overtaking the AV and cutting in, another car intercepting the AV at a traffic sign, or the AV entering a highway. You can then mix these scenarios to capture more complex situations, for example, the AV approaching a traffic sign while overtaking a bicycle, or the AV entering a highway in the evening during a snowstorm.

Because M-SDL scenarios are abstract, parameterized, incomplete descriptions, you can require an M-SDL tool to create many concrete variants of a scenario by varying such parameters as speed, vehicle type, lighting conditions and so on. By allowing random generation of a parameter, or by using constraints that specify a particular value or a range of legal values for a parameter, you guide the M-SDL tool to generate interesting variants automatically. In a similar manner, you can randomize or control the overlap of multiple scenarios in a scenario mix.

You can use a scenario description in passive mode to monitor a test run on any execution platform and determine whether the criteria for successful completion of that scenario have been met. The M-SDL tool then collects and aggregates parameter data from successful test runs, thus enabling you to measure the safety of your AV. Assuming that you have specified goals for each scenario or type of scenario, you can easily identify scenarios that require more testing.

M-SDL is a mostly declarative programming language. The only scenario that executes automatically is the top-level scenario. You control the execution flow of the program by adding scenarios to that top-level scenario. M-SDL tools provide various built-in scenarios that, for example, let you choose the scenario to execute based on a condition, or randomize the selection of a scenario from a list.

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

The language definition provided in this manual is accurate at the time of publication. M-SDL is expected to evolve over time in order to align with relevant standards as they emerge. To see a list of changes to this manual since the last release, refer to Change log (page 171).

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 – represent real world entities. They are like structs, but also have associated scenarios.
- Scenarios – describe the behavior of actors.
- Modifiers – modify the behavior of scenarios.

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.

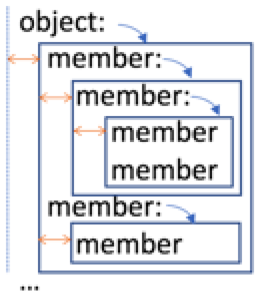

Figure 1 Code blocks

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>+]
```

- The do-not-generate operator ! specifies that no value should be generated for this field before the run executes. Instead, a value is generated or assigned during the run.
- <field-name>: is 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:

- In the second example, the **start_speed** field holds a value of type **speed**. Within the **with** block, **it** is an implicit variable that refers to the current item, in this case, **start_speed**. A **keep** constraint is added to require the speed to be less than 100 km per hour.
- 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 a digit or an underscore are not allowed.

3.4. Predefined identifiers

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.

3.4.1. 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:
    in1: 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:
    top_val: int

extend top.A:
    in2: in b1 with:
        keep(actor.top_val == it.b_val)
        keep(me.a_val == it.b_val)

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

3.4.2. Scoping rules for accessing objects within **it**

To access an object such as a field within **it**, you can either use **it.**<object-name> or just <object-name>. In the latter case, if the containing object (scenario, struct, actor) has an object with the same name, the object within **it** takes precedence. This scoping rule applies not just to fields but to all other objects within **it**, such as events and methods.

You can access an object in the containing object, even if it has the same name as an object within **it**, by using the appropriate identifier. For example, if you want to access a **me** field that is shadowed by an **it** field, you should explicitly use: **me.**<field>. The last line in the following example is the same as **keep(it.duration == me.duration)**.

```
scenario car.x:
    do serial():
        a()
        b()
    with:
        keep(duration == me.duration)
```

Note: The special syntax of path expression starting with a dot ('.') is **not allowed.** For example, **keep(.duration == me.duration)** is not allowed.

3.4.3. Predefined constants

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

3.4.4. Keywords

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

actor	and	any	as	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	label	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				

3.5. 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.

3.5.1. 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.

3.5.2. 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**.

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.

3.5.3. 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.

3.5.4. 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.

3.5.5. 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.

3.6. 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)
```

3.7. 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 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 100) for how automatic labels are computed.

3.7.1. 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.

3.7.2. 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.

Example

```
actor z:
    x: int

scenario z.y:
    keep(x == 2) # resolves to z.x
```

3.8. 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.

3.8.1. 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, either positive or negative. The compiler currently supports, for example:

```
123.45
123.45e+6
123.45E+6
.45e+06
```

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

3.8.2. 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.

Arithmetic expressions involving physical types are resolved using dimensional analysis. For example, distance/time resolves to speed.

Examples:

```
2meter
1.5s
[30..50]kph
[(start_speed-1kph)..(start_speed+1kph)]
6m/3s
```

Name	Units
acceleration	kphps (= kph per second), mpsps or meter_per_sec_sqr (= meters per second per second)

Name	Units
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

3.8.3. Boolean types

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

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

3.8.4. 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]
```

3.8.5. 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:

- 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])
```

3.8.6. String type

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:
     log_info("There are $(cars.size()) cars.")
```

You can concatenate multiple strings with the + character. For example:

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

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

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

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

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

3.8.7. Resource types

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

3.8.8. 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.

You can extend compound types to include new attributes. For example, you can extend the predefined actor **car** to include new data fields, events, scenarios and so forth. Because extension modifies the type being extended, all instances of the type get the newly added attributes.

You can pass the attributes of a compound type to a new compound type using simple inheritance. For example, you can create a new actor **my_car** that inherits the attributes of **car**. You can then add new attributes. Simple inheritance does not modify the original type.

You can also pass the attributes of a compound type to a new compound type using conditional inheritance. With conditional inheritance, 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**. Like simple inheritance, conditional inheritance does not modify the original type.

3.9. 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]

Operator type	Operators	Description
If then else	x ? y : z	Select an expression
Arithmetic	+ - * / %	Perform arithmetic operations
Type check	is(<type>)	Check whether an object is a specified type
Type cast	<path-to-object>.as(<type-name>)	Cast an object to the specified type

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.

3.10. 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.

3.11. 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.
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.

Term	Definition
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.
path expression	Comprises steps connected by dots. Each step can be an identifier, 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 any scenario invocation within another scenario. 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.

Term	Definition
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.
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. Predefined AV types

Summary: This topic describes the predefined types for AV.

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

4.1. 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()
```

4.2. 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)
```

4.3. Predefined car actor

The **car** actor has a predefined scenario **drive()** as well as various predefined fields and events. See drive (page 126) for more information on this predefined scenario.

4.4. Predefined car actor fields

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

Field name	Description
is_dut: bool	Constrained by soft constraint to **false**.
physical: car_physical	A struct field. See below for full description.
policy: car_policy	A struct field. See below for full description.
state: car_state	A struct field. See below for full description.
category: car_category	The car_category type is defined as **sedan**, **truck**, **bus**, **van**, **semi_trailer**, **trailer**, **four_wheel_drive**.
model: string	Initially an empty string.
color: car_color	The car_color type is defined as **white**, **black**, **red**, **green**, **blue**, **yellow**, **brown**, **pink**, **grey**.
av_control: av_control	Constrained by soft constraint to **manual**
length: distance	Car length.
width: distance	Car width.
driving_style	Relevant only for non DUT cars, the style type is defined as **aggressive** or **normal**.
info: list of string	This field contains the most recent informational message emitted by the **car** actor and passed by the simulator. **Note:** do not constrain this field.
passing_by_info: pass_by_info	A list of struct fields. See below for full description.
lane_shifts: int	s

Constraints set when category == truck

- keep(soft policy.max_speed == 120kph)
- keep(soft physical.max_acceleration == 1.5mpsps)
- keep(soft physical.min_acceleration == -3mpsps)

- keep(soft length == 15meter)
- keep(soft width == 2meter)
- keep(soft length == 5meter)
- keep(soft width == 180centimeter)

4.4.1. Configuring the **car** profile

Four struct fields are defined in a **car** actor to hold various attributes:

- **physical**
- **policy**
- **passing_by_info**
- **state**

The **physical** and **policy** fields hold data related to a **car** actor's physical limitations and normal behavior (its *profile*). The **passing_by_info** and **state** fields hold data related to the car's current state, as updated by the simulator.

It is recommended to complete the profile of the DUT. By constraining the attributes of the **physical** and **policy** fields, you specify the expected limitations and behavior of the DUT.

A complete profile improves the planning of scenarios by fitting them to the DUT and saving extraneous computation time. Messages are issued if the DUT behaves differently than specified. You can control the level of severity.

You can modify the default constraints for the purpose of a particular scenario or test. You may also constrain these fields for cars other than the DUT. Constraining the attributes of cars other than the DUT is of less importance as the fields are set adequately by default.

These struct fields are described in detail below.

4.4.2. Predefined car.physical field

The **physical** struct field (type: car_physical) of a **car** actor holds a general description of a car's physical limitations. This description is used mainly for checking feasibility and for kinematic controlling. The fields in this struct, as well as the values set by soft constraint, are shown in the following table.

An error is issued if any car (either the DUT or an NPC) does not keep the declared physical limitations. (For example, the car drives at 300kph in simulation.) The severity of the issue is determined by the **physical.severity** field. Since this is a soft constraint, you can override it with a hard constraint.

Field name	Description	Soft constraint
severity: issue_severity	One of **ignore**, **info**, **warning**, **error_continue**, **error** (continue until end of cycle), **error_stop_now**.	error
min_speed: speed	Minimum speed, where negative speed means reverse	0kph
max_speed: speed	Maximum speed	200kph
min_acceleration: acceleration	Minimum acceleration, where negative acceleration means braking	-30mpsps
max_acceleration: acceleration	Maximum acceleration	10mpsps
power_up_time: time	The time it takes **dut** autoware to power up. The value for all cars will be taken from dut.car.	0second

Constraints on car.physical fields

- keep(soft physical.severity == error)
- keep(soft min_speed == 0kph)
- keep(soft max_speed == 200kph)
- keep(min_speed <= max_speed)
- keep(soft min_acceleration == -30mpsps)
- keep(soft max_acceleration == 10mpsps)
- keep(min_acceleration <= max_acceleration)
- keep(soft power_up_time == 0second)

4.4.3. Predefined car.policy field

The **policy** struct field (type: car_policy) of a **car** actor holds a general description of a car's behavior in normal circumstances. This description is used mainly for planning scenarios. There is no need to constrain all the fields, but better results are possible if the policy is at least partially described. The fields in this struct are shown in the following table.

An error or warning is issued if the car does not keep the declared policy limitations. (For example, the car can brake at 10mpsps, but by policy it does not brake at more than 4mpsps.) By default this field is set to error only for autonomous cars such as the DUT and cars in car_group. It is a warning otherwise. Since these are soft constraints, you can override them with hard constraints.

Field name	Description	Soft constraint
severity: issue_severity	One of **ignore**, **info**, **warning**, **error_continue**, **error** (continue until end of cycle), **error_stop_now**.	error for DUT and cars in **car_group**; otherwise, warning
min_speed: speed	Minimum speed, where negative speed means reverse	0kph
max_speed: speed	Maximum speed	150kph
max_legal_speed_percent: uint	Maximum speed as percentage of the road legal speed	100
arrival_speed: speed	Speed at goal position	Not less than min_speed and not greater than max_speed
min_acceleration: acceleration	Minimum acceleration, where negative acceleration means braking	-4mpsps
max_acceleration: acceleration	Maximum acceleration	2mpsps
average_acceleration_to_cruise: acceleration	Average acceleration until cruise speed for autonomous dut	max_acceleration * 3 / 4
unplanned_standing_time: time	How long may the car stand unplanned in the middle of the road	100millisecond

Field name	Description	Soft constraint
too_close_TTC: time	When to emit **too_close** warning	0millisecond
max_jerk: jerk	Define for reasons of comfort	MAX_INT*1nm_per_ms_cubed
may_jitter: bool	Define for reasons of comfort	For DUT, false
may_exceed_legal_speed: bool	Define for reasons of comfort	For DUT, false

Constraints on car.policy fields

- keep(soft (av_control == autonomous) => policy.severity == error)
- keep(soft (av_control != autonomous) => policy.severity == info)
- keep(soft min_speed == 0kph)
- keep(soft max_speed == 150kph)
- keep(soft max_legal_speed_percent == 100)
- keep(arrival_speed >= min_speed and arrival_speed <= max_speed)
- keep(soft min_acceleration == -4mpsps)
- keep(soft max_acceleration == 2mpsps)
- keep(soft average_acceleration_to_cruise == max_acceleration * 3 / 4)
- keep(soft unplanned_standing_time == 100millisecond)
- keep(soft too_close_TTC == 0millisecond)
- keep(soft max_jerk == MAX_INT*1nm_per_ms_cubed)
- keep(soft is_dut => may_jitter == false)
- keep(soft is_dut => may_exceed_legal_speed == false)

4.4.4. Predefined physical and policy constraints

The following constraints are defined for a **car** actor's physical and policy attributes:

```
keep(policy.max_speed <= physical.max_speed)
keep(policy.min_speed >= physical.min_speed)
keep(policy.max_acceleration <= physical.max_acceleration)
keep(policy.min_acceleration >= physical.min_acceleration)
```

4.4.5. Predefined car.passing_by_info field

The **passing_by_info** struct field of a **car** actor holds data received from the simulator, reflecting the relative position of nearby **car** actors. The fields in this struct are shown in the following table.

Note: do not directly constrain these fields because they carry information forwarded by the simulator.

Field name	Description
other_car: car	Identifies the other car near by
time_to_passing_by: time	How long until cars pass each other; when 0, the cars are passing
lateral_distance: distance	Lateral distance between cars

4.4.6. Predefined car.state field

The **state** struct field of a **car** actor holds data received from the simulator, reflecting the current state of a **car** actor. The fields in this struct are shown in the following table.

Note: do not directly constrain these fields because they carry information forwarded by the simulator.

Field name	Description
speed speed	Current car speed
acceleration: acceleration	Current car acceleration
jerk: jerk	Current car jerk
road_position: road_position	Current road position in road coordinates
road_speed: road_speed	Current road speed in road coordinates

Field name	Description
road_acceleration: road_acceleration	Current road acceleration in road coordinates
global_position: global_position	Current position in global coordinates
global_speed: global_speed	Current speed in global coordinates
location: location	
max_acceleration_until_now: acceleration	Max acceleration
max_break_until_now: acceleration	Max brake applied
max_lateral_speed: speed	Max traced lateral speed
avg_lateral_speed: speed	Average traced lateral speed
lateral_speed_calc_count: int	

4.4.7. Predefined car events

```
event collision(other_car : car) # car collision with another car
event lane_shift(agent: car, from: location, to: location, start_time: time)
event car_passing_by(info : pass_by_info)
```

4.4.8. Predefined car external methods

- time_to_passing_by_dut(time : time): bool
- create_location():location
- trace_lateral_speed()
- on_created(desc: av_actor_description)
- get_passing_by_info(other_car: car): pass_by_info
- get_min_speed(): speed
- get_max_speed(): speed
- get_max_legal_speed_percent(): uint
- get_arrival_speed(): speed
- get_min_acceleration(): acceleration

- get_max_acceleration(): acceleration
- get_average_acceleration_to_cruise(): acceleration
- get_unplanned_standing_time(): int
- get_max_jerk(): jerk
- get_may_jitter(): bool
- get_may_exceed_legal_speed(): bool

4.5. Predefined AV 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_control	manual, autonomous
av_gen_mode	hard_mode, soft_mode, gen_only
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
driving_style	aggressive, normal
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

Type name	Values
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

5. Inheritance

Summary: This topic explains how to use extension, unconditional inheritance and conditional inheritance

5.1. Introduction

M-SDL defines four extensible types subject to inheritance:

- Enumerated types
- Structs
- Actors
- Scenarios / Modifiers

There are four different inheritance-related mechanisms that you can apply:

- Extension
- Unconditional inheritance
- Conditional inheritance
- The **in** modifier

The applicability of these mechanisms to the various extensible classes are summarized in the following table.

Extensible type	Extension	Unconditional inheritance	Conditional inheritance	in modifier
Enumerated type	yes	no	no	no
Struct	yes	yes	yes	no
Actor	yes	yes	yes	no
Scenario / modifier	yes	no	yes	yes

5.2. Extension

Extending a type allows you to add features to a type while retaining its name. All instances of a type are endowed with the union of features declared in all that type's extensions. Features in an extension cannot shadow previously declared features, though some features, like function declarations, allow overrides.

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

Extending is particularly helpful when somebody else already created a library of inter-related actors / scenarios, and you want to add some attributes, constraints, scenarios or other features to accommodate project-specific needs.

Extension, like all other inheritance mechanisms, applies at compile-time. M-SDL is statically typed, so each object's type is known and unchanging during execution.

5.3. Unconditional inheritance

M-SDL implements single inheritance between extensible classes. This works like inheritance in most OO programming languages: a new subtype is declared, endowed with all the features of the parent type (*supertype*). Both types are accessible. Modifications to the supertype are automatically applied to the subtype. Changes to the subtype do not affect the supertype.

Unconditional inheritance is expressed by the following syntax:

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

Example: In this example, the subtype **junction** inherits from the supertype **road_element**.

```
struct junction: road_element:
    ...
```

5.4. Conditional Inheritance

Conditional inheritance enables the creation of subtypes that depend upon a value of a Boolean or enumerated field. The condition always sets a single field to a specific value. The field value (*determinant*) is a constant literal that is fixed during execution.

Conditional inheritance is specified by declaring the field and value that defines the dynamic subtype:

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

For example, the actor **vehicle** has the following attributes:

```
actor vehicle:
    category: vehicle_category # truck, car, motorcycle
    emergency_vehicle: bool
```

Assuming the above declaration, you can define:

```
actor car: vehicle(category: car):
    ...

actor police_car: car(emergency_vehicle: true):
    ...
```

Conditional inheritance has two advantages over unconditional inheritance:

- An object can inherit features from multiple orthogonal conditional types.
- A generateable field can be declared as a type, but be assigned a dynamic subtype by the generator.

These capabilities are discussed below.

5.4.1. Relations between conditional and unconditional Inheritance

The rule governing the relationship between these two kinds of inheritance is as follows:

Rule 1: A conditional type cannot be inherited unconditionally.

Inheritance relationships form a tree whose trunk is the predefined M-SDL types. Main limbs are inherited unconditionally. The final branches of the unconditional inheritance tree can be the roots of conditionally inherited sub-trees.

5.4.2. Relations between conditional subtypes

Any pair of conditional subtypes (types conditionally inherited from the same supertype) have one of the following relationships:

- One of the types can be a supertype of the other; or
- Both types have a common supertype. (They are orthogonal.)

5.4.3. Type membership

Consider a field declared as type **car**. It may be assigned an object whose **emergency_vehicle** field is set to **true**.

Because the field is declared as type **car,** its value must be an instance of **car**, and only features of **car** are accessible. However, M-SDL allows type-membership checking, using the **is()** operator.

Using type checking, you can access features under an active subtype, even though it is not the declared type. Such active but undeclared subtypes (**police_car** in this example) are called *latent subtypes*. This is summarized by the following two rules:

- **Rule 2:** The declared type determines an object's type membership.
- **Rule 3:** Dynamic type check allows access to latent subtype features.

5.4.4. Preventing type reconvergence

Type reconvergence occurs when two orthogonal subtypes are combined. This means the resulting type has multiple supertypes. This is not allowed in a single-inheritance scheme.

Consider the following example. In addition to the above **vehicle** declaration, add the following:

```
actor truck: vehicle(category: truck):
    ...

actor fire_truck: truck(emergency_vehicle: true):
    ...
```

It might be possible to declare a type called, for example, **first_responder**, that includes both **police_car** and **fire_truck**. That hypothetical type would inherit from both, with a determinant that is a Boolean expression:

```
# this is not allowed in M-SDL semantics
emergency_vehicle == true and (category == car or category == truck)
```

M-SDL semantics disallows such complex determinants, in order to prevent type reconvergence. This is expressed in the following rule:

Rule 4: Each field can occur at most once in a conditional type determinant.

Note that it is possible to type check an object dynamically in order to determine if it's an emergency vehicle.

5.4.5. Type checking with the is() 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 that accepts a Boolean expression.

5.4.6. Type casting with the as() operator

Syntax:

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

Example:

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

as() performs a cast, declaring the type of the object to be the specified type-name. 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 / execution).

5.5. Scenario and modifier inheritance

The rules for scenario and modifier inheritance are the same; “scenario” is used below to refer to both.

Scenario inheritance rules are compounded because scenarios are features of their actors, but they are also types by themselves. The inheritance process is described by the following steps:

1. An actor subtype (conditional or unconditional) inherits its supertype scenarios.
2. It then can extend the inherited scenarios, modifying their behavior.
3. It can also declare new scenarios that can inherit conditionally from any of its scenarios (that is, other scenarios of the type or any of its supertypes).

When extending a scenario, it is possible to extend its behavior by adding a **do <scenario invocation>**, which is performed serially. The inherited behavior can be overridden by using **do only <scenario invocation>**.

The following examples illustrate some options:

```
# drive scenario defined under car
scenario car.drive:

# police_car inherits drive() because it is a conditional subtype of car.
extend police_car.drive:

# the scram() scenario conditionally inherits from police_car.drive().
scenario police_car.scram: drive(emergency_lights: on):
```

5.6. The in modifier

While **in** is described as a modifier in MSDL, it has properties that affect the type system.

Syntax:

```
in <path-expression> with:
    <members>
```

The **in** path designates a scenario instance to which the members are applied. Some possible members, like **on**, **cover**, **synchronize** and **until** are changing the underlying type of the scenario instance, essentially creating a prototype. (In the JavaScript sense, a prototype is a subtype that's specific to an instance).

The following rule applies to the use of the **in** modifier:

Rule 5: The application of **in** is static. Although the modifier may appear in some temporal context (such as the invocation of a scenario), the effect of it is static and global. The affected scenario instance will be endowed with the **in** properties throughout its existence.

This is implemented in the type system by creating a latent subtype of the affected scenario type. That latent subtype will have the **in** members. The particular instance affected by the **in** will be statically typed as the latent subtype.

6. Statements

Summary: This topic describes M-SDL statements.

6.1. actor

Purpose

Declare an active object with activities (behaviors)

Category

Statement

Syntax

```
actor <actor-name>[: <base-actor-type>[(<condition>)]] [:
    <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.

<base-actor-type>

The name of a previously defined actor type. The new type inherits all members of the previously defined type as well as its associated scenarios.

<condition>

Has the form **<field-name>:<value>**. This syntax lets you add members to an actor only if the field has the value specified.

<member>+

Is a list of one or more of the following:

- field declarations
- **keep** constraints
- **cover** definitions
- **event** declarations
- 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)
```

You can use the following extension mechanisms with actors:

- **extend**
- Unconditional inheritance
- Conditional inheritance

Example actor declaration

```
type vehicle_category: [car, truck, motorcycle]
type track_kind: [single_track, multi_track]


actor my_vehicle:
  category: vehicle_category
  emergency_vehicle: bool
  track_kind: track_kind
```

Example unconditional inheritance

```
actor my_car : car:
    keep(category==four_wheel_drive)
```

Example conditional inheritance

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

6.2. 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> : []
```

Enumerated types can be extended. You cannot use inheritance, either unconditional or conditional, with enumerated types.

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, timid]
```

6.3. extend

Purpose

Add to an existing type or subtype of an enumerated type, struct, actor, scenario or modifier

Category

Statement

Syntax

```
extend <type-name> :
    <member>+
```

Syntax parameters

<type-name>

Is the name of a previously defined enumerated type, struct, actor, scenario or modifier.

<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)
```

Example extend modifier

In the following example, the modifier **map.path_curve_with_sign()** is extended to include a third path modifier, **path_length()**. Below is the original modifier declaration:

```
modifier map.path_curve_with_sign:
    path1: path
    max_radius: distance
    min_radius: distance
    side1: av_side
    sign1: sign_type

    path_curve(path1, max_radius, min_radius, side: side1)
    path_has_sign(path1, sign1)
```

Now here is the extension:

```
extend map.path_curve_with_sign:
    num_of_lanes: int

    path_min_driving_lanes(path1, num_of_lanes )
```

6.4. 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:

These are the search rules for **import**:

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 directory where the importing file resides.
3. Else look in the current directory.
4. Else look in directories specified by the SDL_PATH environment variable.
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
```

6.5. 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
- 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. Modifiers can be extended.

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

You can use the following extension mechanisms with modifiers:

- **extend**
- Conditional inheritance
- the **in** modifier

Note: Unconditional inheritance is not allowed for modifiers.

Example 1

```
modifier car.speed_and_lane:
    s: speed
    l: lane_type

    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)
```

6.6. scenario

Purpose

Declare an ordered set of behaviors by an actor

Category

Statement

Syntax

```
scenario <name>[: <base-scenario-type>(<condition>)] [:
    <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.

<base-scenario-type>

Is the name of a scenario previously defined with <actor-name>. The new type inherits the behavior of the previously defined type.

<condition>

Has the form **<field-name>:<value>**. This syntax lets you modify the behavior of the scenario only if the field has the value specified.

Note: Unconditional inheritance is not allowed for scenarios.

<member>+

Is a list of one or more of the following:

- field declarations

- **keep** constraints
- **cover** definitions
- **event** declarations
- 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.

You can use the following extension mechanisms with scenarios:

- **extend**
- Conditional inheritance
- the **in** modifier

Example scenario declaration

```
# 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 scenario inheritance

```
type vehicle_category: [car, truck, motorcycle]
type emergency_vehicle_kind: [fire, police, ambulance]
type light_kind: [parking, low_beam, high_beam, emergency]

actor vehicle:
    category: vehicle_category
    emergency_vehicle: bool

actor emergency_vehicle: vehicle(emergency_vehicle:true):
    vehicle_kind: emergency_vehicle_kind

scenario vehicle.drive:
    lights: light_kind


scenario emergency_vehicle.drive_emergency: drive(lights: emergency)
    siren: bool with: keep(it == true)
```

See Example scenarios (page 8) for more examples of how to create and reuse scenarios.

6.7. struct

Purpose

Define a compound data structure

Category

Statement

Syntax

```
struct <type-name>[: <base-struct-type>[(<condition>)]] [:
    <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. The new type inherits all members of the previously defined type.

<condition>

Has the form **<field-name>:<value>**. This syntax lets you add members to a struct only if the field has the value specified.

<member>+

Is a list of one or more of the following:

- field declarations
- **keep** constraints
- **cover** definitions

- **event** declarations
- 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.

You can use the following extension mechanisms with actors:

- **extend**
- Unconditional inheritance
- Conditional inheritance

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 unconditional inheritance

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

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

Example struct conditional inheritance

This example creates a new struct type **snow_storm_data** from the base type **storm_data** and adds a field for **snow_depth**.

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

struct storm_data:
    storm: storm_type
    wind: speed

struct snow_storm_data: storm_data(storm: snow):
    snow_depth: distance
```

7. Struct, actor or scenario members

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

7.1. Coverage and Performance Metrics

Metrics collected during test execution are used to answer two critical questions: *how well was the DUT (Device Under Test) tested*, and *how well did the DUT perform within these tests*. The first question is answered by the *coverage grade*, the multi-dimensional representation of all situations encountered during testing. The second question is answered by *performance grade*, the collection of Key Performance Indicators, normalized within their context.

Together, these metrics provide insight to the following questions

Coverage:

- What is the current coverage grade (overall and specifically for a given scenario)?
- What are the main coverage holes for a scenario? How do they cluster, in other words, are there big uncovered areas?

Performance:

- What were the values for a specific KPI (overall and specifically for a given scenario)? Do those cluster in some interesting way?
- How well does the DUT perform on specific KPI grades (overall and for a scenario)? Where is this worse / better than the previous SW release?
- How may runs actually failed with a DUT error, in other words, with a grade below the threshold? How do they cluster?
- What is the trend in all of these relative to the previous week? Which metrics improved and which degraded?

Both coverage and performance metrics defined in M-SDL typically implement a verification plan, specifying goals and thresholds. The verification plan is a result of an engineering effort driven by requirements such as AV performance, ODD, safety standards and so on.

7.1.1. 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>*

See cover() and record() parameters (page 74) below.

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 metric group, along with **record** data sampled by the same event. 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.state.location, car1.state.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 < 0cm or rel_d_slow_end > 6000cm))
```

Cross coverage: 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], event: change_lane_start,
    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.state.speed, @start) with:
    cover(it, unit:kph)
```

7.1.2. record()

Purpose

Define a performance metrics and other data collection point. Any scalar or string value can be captured by record().

Category

Struct, actor, or scenario member

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

Syntax

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

Syntax parameters

<exp>

Is an expression using objects in the enclosing construct. The expression must be of scalar or string type. The value of the record metric is the value of the expression when the record group event occurs.

<param>*

See cover() and record() parameters (page 74) below.

Description

record() is used to capture performance indicators and other data items that are not part of the coverage model, like the name and version strings identifying the DUT.

The purpose of performance evaluation is to see how well the AV performed in specific conditions as they occur within a test. Performance is evaluated along multiple dimensions, like safety, ride comfort and so on.

Performance metrics can provide pass/fail indication: sampled values that cross a specified threshold will raise an error message, indicating that the DUT (Device Under Test) performance was outside the acceptable range.

KPIs or Key Performance Indicators are the raw metrics measured to see how well the AV performed. There can be safety-related KPIs (such as min-Time-To-Collision or min-TTC, measured in seconds), comfort-related KPIs (such as max-deceleration, measured in meter/second2), and so on.

Often, raw KPI values need to be interpreted in the context of a specific scenario. For example, it may be acceptable to cross the max-deceleration threshold if emergency braking is required. For this purpose, raw KPIs are converted to performance grades. A performance grade (also called a *normalized KPI*) is a context-dependent number between 0 (“really bad”) and 1 (“excellent”) that is attributed to some aspect of the DUT behavior. This grade is computed using a user-defined grading formula, which converts one or more raw KPIs into a grade in a context-dependent way.

Example 1

The following example shows **record()** used to capture time-to-collision into the metric group associated with the end of a change-lane maneuver.

```
extend 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(car1), @change_lane.end)

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

Cross record - Combining metrics

You can combine record and coverage metrics of two or more previously defined items, by specifying the items as a list. This creates a Cartesian product of the specified metrics, showing every combination of values of the items. Only items that belong to the same metrics group (same sample event) can be crossed.

Example 2

The following example creates a cross record: a Cartesian product of the time-to-collision record item and the DUT velocity cover item, both sampled at the start of the change_lane phase. This is considered a cross-record because it includes at least one record item.

```
record([ttc_at_end_of_change_lane, dut_v_cls], event: start,
    text: "Cross record of TTC and absolute DUT velocity")
```

7.1.3. cover() and record() parameters

Cover and record members accept a comma-separated list of zero or more of the following:

- name: <name> specifies a name for the cover or record 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.

  Notes:

  - You must specify a unit for cover items that have a physical type.
  - If the cover or record item(s) are not members of the **end** metric group, a sampling event must be specified.

- 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 *metric group*. Note that both coverage and performance items can be collected in the same metric group.

  - You can sample a field value on one event and cover it on another. This way you can, for example, capture the speed of a car when changing lanes, but associate the cover item with the scenario end metric group.
  - The cover event must be local (an event defined in the enclosing struct/actor/scenario). No dotted path expressions are allowed. If necessary, you can define a local event to be derived from some path expression and use that for coverage, for example:

    ```
    event change_lane_start is @change_lane.start
    ```

- text: <string> is explanatory text, enclosed in double quotes, about this metric point.
- ignore: <item-bool-exp> defines values that are to be completely ignored. The expression is a Boolean expression that can contain only the 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).
- disable: <bool> must be either the literal **true** or **false**. **true** completely disables a metric group. This parameter is used with **override** (see below), to disable an existing metric item. (Default : false).
- buckets: <list of bucket boundaries>. Provides a way to declare different-sized buckets, by specifying bucket boundaries. A list of N values defines N-1 buckets. Each value must be >= the previous one, else this is an error. For example:

```
cover(speed, unit: kph, buckets: [1, 2, 6.5, 10])
```

Will create the following buckets

```
1..2, 2..6.5, 6.5..10
```

This option replaces both **every** and **range**, if specified.

7.1.4. override

The **override** option is used to override some parameter in an already defined **cover()** or **record()** member.

Syntax

```
cover | record(override:<name> [, <param>* ])
```

Description

The override feature works as follows:

- **override** must be the first parameter in a **cover()** or **record()** modifier.
- <name> must be the name of an existing cover or record item in the event metric group.
- Any other specified parameter in the <param> list overrides the corresponding original parameter. Parameters not provided in the override retain their original values. (If no event parameter is specified, the default **event: end** is used.)
- Multiple overrides are allowed. The last value provided for each parameter prevails.
- This automatically percolates to any cross coverage using this item.

Example

```
# Original definition
cover(speed_diff, units: kph, range: [1..20], every: 5)

# New definition overrides the every: 5 and adds ignor
e
cover(override: speed_diff, every: 4, ignore: speed_diff in [10..13]kph):
```

Restrictions

- The name of the item cannot be changed. If the original item’s <name> parameter was not specified then the item name is automatically generated. For example, **x.y** becomes **x_y**. This is the name you need to use with **override: <name>**.
- **override**:<exp> is not allowed. You cannot override the expression as originally defined.
- Override can only be used in the same type where the cover or record modifier was originally defined. It cannot be used to override a cover or record modifier in a subtype.

7.2. 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:

```
scenario dut.scenario1:
    car2: car
    path1: path
    distance1: distance

    on (map.abs_distance_between_locations(dut.car.location,
          car2.location) < 2m):
      emit dut.near_collision(other_car: car2, distance: distance1)

    do serial:
        car2.drive(path1)
```

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)
```

7.3. external method declaration

Purpose

Declare a procedure written in a foreign language

Category

Struct, actor, or scenario member

Syntax

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

Syntax parameters

<msdl-method-name>

Is a name that is unique in the current M-SDL 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>* )
```

where <param> is a language-specific list of parameters.

external.cpp allows you to declare methods implemented in C++. The bind expression has the following syntax:

```
external.cpp([<c++-method-name>, ]<shared-object-name>)
```

<c++-method-name> specifies the name for the C++ method. The name must be unique within the current context. When not set, the C++ method name is the same as <msdl-method-name>.

<shared-object-name> specifies the shared object that contains the implementation of the C++ method. The name should include only the file name; the full library path is detected according to the operation system conventions. For example, in Linux, libraries are searched for in paths defined by the environment variable LD_LIBRARY_PATH.

Description

The following declare but do not define an external method:

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

Both **empty** and **undefined** let you declare a method without specifying any actions. If an **empty** and **undefined** method is called:

- **empty** returns the default type value, if defined.
- **undefined** causes a runtime error.

The following specify an extension to a previously declared method:

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

In addition, each foreign language includes an interface for invoking M-SDL methods.

Example

```
extend car:
    def calculate_dist_to_other_car(other_car: car) is external.cpp("calculate_dist_t
o_other_car", "libexample.so")

extend top.main:
    on @c.slow.start:
        call dut.car.calculate_dist_to_other_car(c.car1)

    set_map(name: "$FTX_QA/odr_maps/hooder.xodr")
    do c: cut_in_and_slow()
```

7.4. 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 the form:

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

or

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

<member> is either:

- 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 constraint in the form

  ```
  keep(<constraint-type> <constraint-exp>)
  ```

  where <constraint-type> is either **soft** or **default**. **default** is allowed only for fields declared within scenarios.

  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.

Note: If the field you are declaring is of type struct or actor, and you want to constrain one or more of its fields to a single value or range of values, you can pass the fields as parameters using the following syntax:

```
with(<field-name>: [<constraint-type>] <value>, ...)
```

For example:

```
field1: some_struct with(x: default 1m, y:2):
    cover it.x
    cover it.y
```

Is the same as:

```
field1: some_struct with:
    keep(default x == 1m)
    keep(y == 2)
    cover it.x
    cover it.y
```

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

    # the list will have between 2 and 10 items
    # the first item is first_car
    keep(soft cars.size() <= 10)
    keep(soft cars.size() >=  2)
    keep(cars[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.state.speed, @get_ahead.end) with:
        cover(it, text: "Speed of car1 at get_ahead end (in kph)", unit: kph,
          range: [10..130], every: 10)
```

7.5. keep()

Purpose

Define a constraint

Category

Struct, actor, or scenario member

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

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 physical or numeric type to a range of values, constrain an enumerated type to a list of values, or constrain 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)
```

You can use the **in** constraint operator to constrain a physical or numeric parameter to a range of values. In the example below, the legal value for the field **dist** is any value between 2 and 4, inclusive. The legal values for the field **i** are -1, 0, and 1.

You can also use the **in** constraint operator to constrain an enumerated type to a list of values or to constrain a list to be a subset of another list. In the example below, the legal values for the field **ms** are **assertive** and **timid**. The final constraint, **keep(it in list1)** has the effect of adding the three values specified to **list1**. The order of elements in **list1** is not determined.

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

struct misc:
    dist:distance with:
        keep(it in [2..4]m)
    i: int with:
        keep(it in [-1..1])

    ms: my_driving_style with:
        keep(it in [assertive, timid])

    list1: list of uint with:
        keep(it == [0,1,2])
    list2: list of uint with:
        keep(it == [3,4,5])
        keep(it in list1)
```

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

    # the list will have between 2 and 10 items
    # the first item is first_car
    keep(soft cars.size() <= 10)
    keep(soft cars.size() >=  2)
    keep(cars[0] == first_car) # list indexing
```

Read only constraints

The meaning of **read_only(**<exp>**)** within a constraint expression is that the value of <exp> is read, but is not changed. This simplifies the generation problem by making some constraints unidirectional.

```
current_speed: speed with:
    keep(it <= read_only(max_speed))
```

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)
```

8. Scenario members

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

8.1. Scenario modifier invocation

Purpose

Constrain or modify the behavior of a scenario

Category

Scenario member

Syntax

```
[<label>: ]<scenario-modifier>(<params>)
```

Syntax parameters

<label> is an optional label for the invocation. If you have multiple invocations of the same modifier within the same scenario phase, labels are automatically created unless you specify them.

<scenario-modifier> is the name of the modifier you want to invoke.

<params> is a comma-separated list of parameters enclosed in parentheses. The parentheses are required.

Description

You can invoke scenario modifiers in the following contexts:

- Within a scenario declaration outside of the **do** behavioral definition.
- Within the **with:** block of a scenario invocation.
- Within the body of an **in** scenario modifier.
- Within a scenario modifier declaration.

Example map modifier:

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

```
extend top.main:
    set_map("hooder.xodr")
    do a: cut_in_and_slow() with:
        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**.

Note: this example uses labels (**p1**, **p2**) to distinguish the two **position()** modifiers.

```
do parallel:
    dut.car.drive(path) with:
        s1: speed([50..120]kph)
    car1.drive(path, adjust: true) with:
        p1: position(distance: [5..100]m, behind: dut.car, at: start)
        p2: position(distance: [5..15]m, ahead_of: dut.car, at: end)
```

8.2. do (behavior definition)

Purpose

Define the behavior of a scenario

Category

Scenario member

Syntax

```
do [only] <scenario-invocation>
```

Syntax parameters

do only replaces all previously defined behavior with the current definition. Use **do only** to override previously defined or inherited behavior as well as any extensions to that behavior.

See Scenario invocation (page 98) 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 102) 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 100) 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)
```

9. Scenario invocation

Summary: This topic describes how to invoke a scenario.

A scenario can only be invoked within an operator scenario or within a **do** clause of a scenario declaration.

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 100) 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 22) 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 parameters the form [<field-name>:] [**default**] <value> where:

- **default** is the equivalent of adding a default **keep()** constraint.
- <value> is a single value or a range. A unit is required if the type is physical.

The list must be comma-separated and enclosed by parentheses. It can be name-based (<field-name>: [**default**] <value>,…) or, in some special cases, order-based ([**default**] <value>,…). In order-based lists, the first value is assigned to the first field in the scenario, and so on.

In the list, a name-based parameter can follow an order-based parameter, but not vice-versa. Thus, **turn(x:3, y: 5)**, **turn(3, y: 5)**, and **turn(3, 5)** are legal and assign the same values, but **turn(x:3, 5)** is not allowed.

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

<with-block>

<with-block> is a list of one or more **keep()** constraints or scenario modifiers, where the members are listed on separate lines as a block or on the same line as **with:** and separated by semi-colons. For example, the following code:

```
ss2: some_scenario(z: 4) with:
    keep(ss2.ss.x==3)
    keep(ss2.ss.y==5)
```

Is the same as:

```
ss2: some_scenario(z: 4) with: keep(ss2.ss.x==3);  keep(ss2.ss.y==5)
```

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

Example

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)
```

9.1. 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**.

The following rules apply when accessing implicit labels:

- **label()** is limited to accessing invocations within the current scenario. To access an invocation declared in the body of a nested scenario, use the following:

  ```
  label(<path-in-current-scenario>).<user-provided-label-in-invoked>
  ```

  Or:

  ```
  label(<path-in-current-scenario).label(<path-in-invoked-scenario>)
  ```

- User-declared labels cannot be used within **label()**. When combining implicit and user defined labels, **label()** should wrap implicit labels only, as in the following example:

```
<user-defined1>.label(<implicit1>).<user-defined2>.label(<implicit2>)
```

- Implicit labels use the first identifier in any invocation. For example, the label for **dut.car.drive()** is **dut**, accessed by **label(dut)**

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)
```

10. 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**.

10.1. 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 **keep()** constraints or scenario modifiers, where the members are listed on separate lines as a block or on the same line as **with:** and separated by semi-colons.

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:
    keep(i1.car1.color != i2.car1.color)
```

10.2. 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")
```

10.3. 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()**.

Example

```
scenario top.report_match_success:
    do log_info("MATCH: intercept_1 and intercept_2 successfully matched")

scenario top.report_match_failure:
    do log_info("NO MATCH: intercept_1 and intercept_2 no match found")

extend top.main:
    car1: car
    path1: path

    do serial:
        i1: intercept_1()
        match():
            i2: intercept_2()
        then:
            report_match_success()
        else:
            report_match_failure()
```

10.4. 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.

Example

```
scenario top.report_match_success:
    do log_info("MATCH: intercept_1 and intercept_2 successfully matched")

scenario top.report_match_failure:
    do log_info("NO MATCH: intercept_1 and intercept_2 no match found")

extend top.main:
    car1: car
    path1: path

    do serial:
        i1: intercept_1()
        multi_match():
            i2: intercept_2()
        then:
            report_match_success()
        else:
            report_match_failure()
```

10.5. 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 **keep()** constraints or scenario modifiers, where the members are listed on separate lines as a block or on the same line as **with:** and separated by semi-colons.

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** and **end_to_end** parameters are the time offsets of the secondary scenarios relative to the primary one.

The **overlap** parameter is one of the following:

- **any_mix** specifies that there is no constraint on the amount of overlap between operands. This is the default.
- **full** specifies that the secondary scenarios fully overlap the primary one: start_to_start <= 0, end_to_end >= 0.
- **inside** specifies that the secondary scenarios are fully overlapped by the primary scenario: start_to_start >= 0, end_to_end =< 0.
- **equal** specifies that the primary and secondary scenarios start and end at the same point in time.
- **initial** specifies that the secondary scenarios cover at least the start of the primary scenario: start_to_start <= 0, end_to_end can be anything.
- **final** specifies that the secondary scenarios cover at least the end of the primary scenario: end_to_end >= 0, start_to_start can be anything.

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()
```

10.6. 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 **keep()** constraints or scenario modifiers, where the members are listed on separate lines as a block or on the same line as **with:** and separated by semi-colons.

Example

```
one_of:
    i1: intercept_1()
    i2: intercept_2()
with:
    keep(i1.car1.color != i2.car1.color)
```

10.7. 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 **keep()** constraints or scenario modifiers, where the members are listed on separate lines as a block or on the same line as **with:** and separated by semi-colons.

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)
```

10.8. repeat

Purpose

Run multiple scenarios in sequence repeatedly

Category

Operator scenario

Syntax

```
repeat([<count>]):
    <scenario>+
```

Syntax parameters

<scenario>+

Is a list of one or more scenarios.

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

```
do repeat(3):
    i1: intercept_1()
    i2: intercept_2()
```

10.9. 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 **keep()** constraints or scenario modifiers, where the members are listed on separate lines as a block or on the same line as **with:** and separated by semi-colons.

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()
```

10.10. 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.

11. 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.

11.1. 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.

11.2. 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)
```

11.3. 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)
```

12. Zero-time scenarios

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

12.1. 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 car:
    def calculate_dist_to_other_car(other_car: car) is external.cpp("calculate_dist_t
o_other_car", "libexample.so")

extend top.main:
    on @c.slow.start:
        call dut.car.calculate_dist_to_other_car(c.car1)

    set_map(name: "$FTX_QA/odr_maps/hooder.xodr")
    do c: cut_in_and_slow()
```

12.2. 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.

12.3. 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

```
on @foo:
    end()

do cut_in()
```

12.4. 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.

12.5. Zero-time messaging scenarios

Purpose

Print message to STDOUT.

Category

Built-in scenario

Syntax

```
log(<string>)
log_info(<string>)
log_debug(<string>)
log_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
log()	Used to report major events and messages
log_info()	More detailed reporting
log_debug()	Verbose information that may be useful for debug
log_trace()	Most detailed information used to trace execution

Example

```
scenario top.report_match_success:
    do log_info("MATCH: intercept_1 and intercept_2 successfully matched")

scenario top.report_match_failure:
    do log_info("NO MATCH: intercept_1 and intercept_2 no match found")

extend top.main:
    car1: car
    path1: path

    do serial:
        i1: intercept_1()
        match():
            i2: intercept_2()
        then:
            report_match_success()
        else:
            report_match_failure()
```

13. 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)
```

13.1. drive

Purpose

Describe a continuous segment of movement by a car actor.

Category

Movement scenario (**car** actor)

Syntax

```
drive( [duration: <time>,][path: ]<path>[, exactly: <bool>][, adjust: <bool>])[<with-b
lock>]
```

Parameters

<time> is a single time value or a range with a time unit, such as [100..120]s.

<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 **keep()** constraints or scenario modifiers, where the members are listed on separate lines as a block or on the same line as **with:** and separated by semi-colons. For example, the following code:

```
car1.drive(path1) with:
    speed(30kph)
    acceleration(5kphps)
```

is the same as

```
car1.drive(path1) with: speed(30kph); acceleration(5kphps)
```

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:
        avoid_collisions(false)
```

14. 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.

15. 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.

15.1. 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.

<with-block> is a list of one or more **keep()** constraints or scenario modifiers, where the members are listed on separate lines as a block or on the same line as **with:** and separated by semi-colons. For example, the following code:

```
in ci.ga.car1 with: speed([50..75]kph); lane(1)
```

Is the same as:

```
in ci.ga.car1 with:
    speed([50..75]kph)
    lane(1)
```

Note: cover() definitions are not allowed in scenario modifiers, including **in**.

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.

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

do cut_in()
```

*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 extension to **top.main**.

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:
    my_car: car
    my_path: path

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

extend top.main:
    drv_attr: drive_attributes

    in enclosing.nested.start_up with:
        speed(outer.drv_attr.my_speed)

    do enclosing: dut.enclosing_scenario()
```

15.2. 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 call to an external method, a zero-time scenario such as **end()**, **fail()** or a zero-time messaging scenario.

Example

```
scenario top.scenario1:
    path: path
    car1: car

    event near_collision

    on @near_collision:
        log_info("Near collision occurred.")

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

15.3. 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)
```

15.4. trace()

Purpose

Define a trace action.

Category

Scenario modifier

Syntax

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

Syntax parameters

<exp>

Is an expression defining the value you want to be traced. Typically the expression is the name of a field or an expression referring to multiple fields. The expression is evaluated in the context of the current scenario, but it can also refer to any global field.

<param>*

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

- unit: <unit> specifies a unit for a physical quantity such as time, distance, speed. The expression's value is converted into the specified unit, and that value is used as the trace value.

  Note: You must specify a unit for trace expressions that have a physical type. Conversely, do not specify a unit for all other items.

- name: <name> specifies a name for the trace expression. By default, <name> is the same as <exp>, with any sequence of non-alphanumerics replaced by an underscore. This name is used in all targets (the graphical timeline, the CSV file, and the log file). It is also used for filtering.

For example, if you specify **trace(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 **trace(speed1 - speed2, name: speed_diff)**.

- title: <string> defines the explanatory text for the expression to be used in the log and as the headers in the CSV file. The default title is the text of the <exp>.

Description

Tracing an item is useful because you can see exactly when the value of an expression changes during the scenario execution.

Note: Defining a trace with the **trace()** scenario modifier only identifies the items to be traced. To activate the trace, you must issue interactive **trace** commands in an M-SDL tool such as Foretify.

Example trace definitions

```
extend dut.cut_in_and_slow:  # Add some tracing to this scenario
    trace(car1.state.speed, unit: kph)
    trace(dut.car.state.speed, unit:  kph)
    trace(car1.state.speed - dut.car.state.speed, unit: kph, name: speed_diff,
          title: "speed diff to ego")
    trace(side) # Will show the side as enum (constant value in this case)

extend top.main:

    do c: cut_in_and_slow()
```

15.5. 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(@e1)


# Example 2
on @e1:
    end()

do serial:
    phase1: car1.drive(path1) with:
        speed(40kph)

    phase2: car1.drive(path1) with:
        speed(80kph)
```

16. Movement-related scenario modifiers

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

Movement-related scenario modifiers specify or constrain attributes of movement scenarios such as **car.drive**. They must appear either 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

All these modifiers have an optional **gen_mode** parameter that controls how modifier constraints are enforced. By default, modifier constraints are always enforced during the initial planning of the scenario and during execution. In some cases, enforcement might result in a constraint contradiction either during planning or at runtime. For example, if scenario execution deviates from the original plan due to unexpected DUT behavior, enforcing movement modifier constraints during execution can cause the scenario to exit before completion. Using the **gen_mode** parameter, you can relax enforcement of movement modifier constraints in order to complete scenario planning and execution.

The **gen_mode** parameter has the following possible values:

- **hard_mode** - Modifier constraints are enforced both during planning and execution. This is the default.
- **soft_mode** - Modifier constraints are enforced during planning whenever they are not contradicted by another modifier constraint. They are ignored if contradicted. In this mode, modifier constraints are ignored during execution.
- **gen_only** - Modifier constraints are enforced during planning but ignored during execution.

Example:

```
    car1,drive() with:
        speed([30..50]kph, gen_mode: soft_mode)
```

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()
```

16.1. 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.
```

16.2. avoid_collisions

Purpose

Allow or disallow an actor to collide with another object.

Category

Scenario modifier

Syntax

```
avoid_collision(<bool>)
```

Parameters

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

Description

By default, all actors avoid collisions (**avoid_collisions(true)**). This means that the runtime mechanism for collision avoidance is on for the **drive()** scenario specified by the modifier. When set to **false**, the actor moves regardless of surrounding traffic and may collide.

Example

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

16.3. 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)
```

16.4. 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)
```

16.5. 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()
```

16.6. 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()
```

16.7. 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()
```

16.8. 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: <av-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 lane 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)
```

16.9. 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)
```

16.10. 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 position is maintained throughout the current period unless **time** is specified along with **ahead_of** or **behind_of**. In that case, the physical distance between the actors may vary during the time period. See **Description** below for more details.

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.

When **time** is specified along with **ahead_of** or **behind_of**, the physical distance is calculated using the speed of the car that is behind (irrespective of whether it's the scenario's actor or the referenced car) and the location at that moment of the car that is ahead. The speed of the car that is ahead is not taken into the calculation. If <event> is **all**, the physical distance at any point in time refers to the speed of car that is behind at that moment and the location of the car that is ahead at that same moment (meaning that the physical distance may vary during the time period). The following two examples are equivalent; in both cases the physical distance is calculated according to the speed of **car2**.

```
do parallel:
    car1.drive(path) with:
        speed(speed: 30kph, at: end)
    car2.drive(path) with:
        speed(speed: 40kph, at: end)
        position(time: 3second, behind: car1, at: end)

# is identical to:

do parallel:
    car1.drive(path) with:
        speed(speed: 30kph, at: end)
        position(time: 3second, ahead_of: car2, at: end)
    car2.drive(path) with:
        speed(speed: 40kph, at: end)
```

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)
```

16.11. 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)
```

17. 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.

17.1. 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)
```

17.2. 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)
```

17.3. 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)
```

17.4. 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: <tolerance>])
```

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)
```

17.5. 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)
```

17.6. 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)
```

17.7. 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)
```

17.8. 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_length: ]<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)
```

17.9. 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
```

17.10. 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
```

17.11. 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
```

17.12. 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_type: <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)
```

17.13. 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_before: <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)
```

17.14. 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>,]
    [more_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)
```

17.15. 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)
```

17.16. 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)
```

17.17. 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)
```

17.18. 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)
```

17.19. 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.

Description

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

Example

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

18. Change log

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

18.1. Version 20.07

- Added a **gen_mode** parameter to the movement-related scenario modifiers. See Movement-related scenario modifiers (page 138).
- Added a chapter on inheritance. See Inheritance (page 46).
- Created a new description of metrics to include both **cover()** and **record()**. See Coverage and Performance Metrics (page 69).
- Moved the Predefined AV Types section to a new chapter. See Predefined AV types (page 35).
- Clarified that the **on** scenario modifier can call an external method. See on qualified event (page 131).
- Added the **trace()** scenario modifier. See trace() (page 134).
- Updated the list of generic numeric types. See Generic numeric types (page 24).
- Clarified the optional parameters for the map-related modifiers as well as which parameters can be passed by order. See Chapter 15 (page 153).
- Updated the descriptions of the **overlap** parameter of **mix()***. See mix (page 108).
- Changed the names of the messaging scenarios. See Zero-time messaging scenarios (page 124).
- Replaced **like** inheritance and **when** subtypes with simple and conditional inheritance. See actor (page 53), scenario (page 63) and struct (page 66).
- Rewrote the introduction. See Introduction (page 5).
- Added an example of constraining a field of an enumerated type to a list of values. See keep (page 87).
- Described the predefined struct fields of the **car actor**, including **car.physical**, **car.policy**, **car.state**, and **car.passing_by_info**, as well as predefined events and soft constraints. See Predefined car actor fields (page 36).
- Noted that user-defined identifiers beginning with a digit are not supported. See User-defined identifiers, constants and keywords (page 16).
- Clarified the scoping rules for accessing objects within **it**. See Predefined identifiers (page 16).

- Clarified the description of the effect of using a **time** parameter with **ahead_of** or **behind_of** in the **position()** modifier. See position (page 148).
- Documented the syntax for floating point numbers (reals). See Generic numeric types (page 24).
- Noted that enumerated types can be extended. See extend (page 57).
- Added the **duration** parameter to **drive()**. See drive (page 126).
- Updated the syntax description for external methods to show that **is** <bind-exp> is allowed. See external method declaration (page 80).
- Modified the syntax for the **repeat** operator to show that **with:** blocks are not allowed. See repeat (page 114).
- Clarified the syntax and context for invoking scenario modifiers. See Scenario modifier invocation (page 94).
- Documented **read_only** constraints. See Read only constraints (page 91).
- Changed **collides()** to **avoid_collisions()**. See avoid_collisions (page 141).
- Specified that scenario modifiers can be extended. See extend (page 57) and modifier (page 61).
- Noted that the keyword **default** can be used when passing parameters in scenario invocations to specify default **keep()** constraints on field values. See Scenario invocation (page 98).

18.2. Version 0.9.1

The following changes appear in version 0.9.1:

- Expanded the description of list elements. See List types (page 26).
- Updated the syntax for **emit** to show that parameters must be passed by name, not position. See emit (page 118).
- Updated the description of **weather_type** and **time_of_day**. Added **road_type**. See Predefined AV types (page 44).
- Documented the map modifiers **path_same_origin()**, **path_over_highway_junction()** and **path_over_speed_limit_change()**. See Map modifiers (page 153).
- 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 126).
- Clarified the difference between soft and default constraints. See Soft constraints and

default constraints (page 93).

- Added a definition of *basic clock* and replaced all references to **top.clk** with basic clock. See Terminology (page 31).
- 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 98).
- Added **me** and **actor** to the list of predefined identifiers and added an example. See User-defined identifiers, constants and keywords (page 16).
- Documented the new scheme for implicit labels, which uses **label()**. See Automatic labels (page 100).
- Changed the **no_collides()** movement scenario modifier to **collides()**. **Note:** in 20.07 changed again to **avoid_collisions()**.
- Added a definition of "phase" as an informal term. See Terminology (page 31) and parallel (page 112).
- 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 129).
- 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 138).
- Added a physical unit for speed, **meter_per_second** or **mps**. See Physical types (page 25).
- Documented the default file extension as **.sdl** and the use of the SDL_PATH environment variable. See M-SDL file structure (page 21).
- Documented the **name** option of **cover()**. See cover (page 69).
- Documented the **lateral()** movement modifier. See lateral (page 147).
- Clarified the syntax for declaring and calling an external method. See external method declaration (page 80).
- 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 126), Operator scenarios (page 102), Scenario invocation (page 98), and Scenario modifier invocation (page 94).
- Added the physical types **temperature** and **weight**. See Physical types (page 25).
- 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 13).

- 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. **Note: when** subtypes were replaced in 20.07 with conditional inheritance.
- Updated the list of keywords. See User-defined identifiers, constants and keywords (page 16).
- 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 63) and Scenario invocation (page 98).
- Clarified the description and example of global actors. See Actor hierarchy and name resolution (page 22).
- Added a definition of "path expression". See Terminology (page 31).
- Added **sample()** to examples that show how to sample a field at a specified event. See field (page 83).
- Added a note that enclosing parentheses for boolean expressions are optional for the **if** operator. See if (page 103).
- Clarified that identifiers beginning with an underscore character are not allowed. See User-defined identifiers, constants and keywords (page 16).
- Clarified that physical expressions, such as **start_speed-1kph** are allowed in ranges. See Physical types (page 25).

18.3. Version 0.9

This version includes minor edits.

18.4. 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.