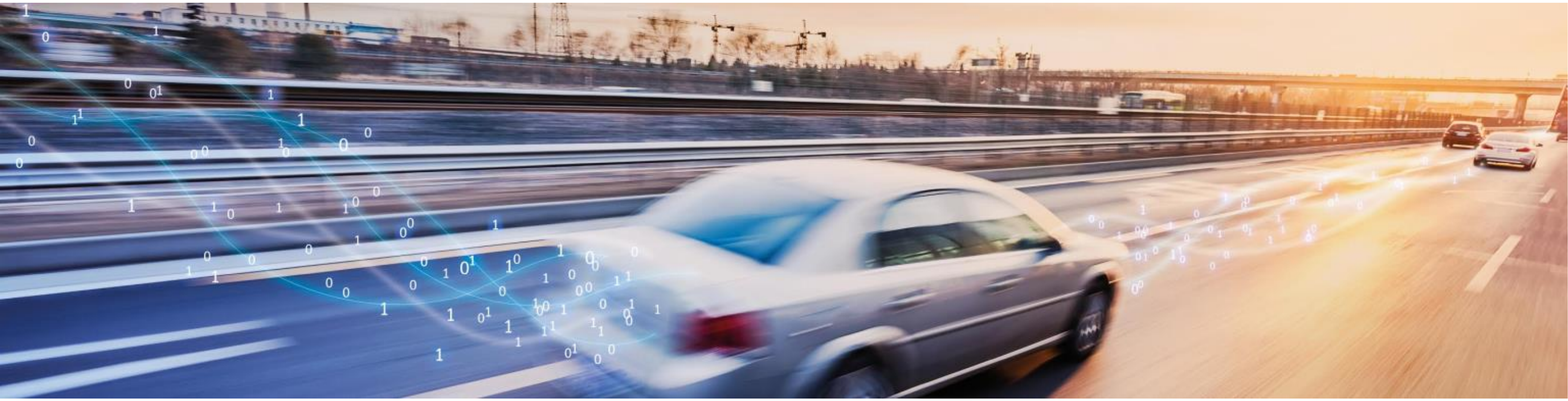
ASAM OpenSCENARIO 2.0 Concepts

Language Concepts

Pierre R. Mai
PMSF IT Consulting

2020-03-26



Association for Standardization of Automation and Measuring Systems

Guiding Principles

Design Guidelines for the Language

Language Concepts Guiding Principles

What we tried to keep in mind and should carry forward into 2.0 standardization

- **Readability**

  The quality of being easy to be read and understand by a target audience, which includes not only domain experts, programmers and engineers, but also safety engineers, regulators, and potentially even the general public.

  While this has clear implications for suitable surface syntax of the language, it also requires the semantics of the language to be well-defined, regular, and focused on the problem domain, and not technical implementation artifacts.

- **Expressivity**

  The ability of the language to allow the concise and direct expression of domain subject matter. This interacts with readability and the declarative nature of the language, but also enables for example the ability to query databases of scenarios using language semantics rather than only through meta-databased queries.

- **Composability**

  The quality of being able to use standard language building blocks to incrementally build up more complex behavior in well-defined ways. This requires clear rules for composition and composition operators, that afford some level of predictable behavior post composition, while also allowing for interesting emergent behavior to occur, to ensure coverage of overall realistic traffic behavior.

- **Portability**

  The quality of being able to use the language and scenarios written in it across many execution platforms (including simulation platforms as well as real-world test tracks), without undue adjustments needing to be made. This includes the provision of suitable fallback mechanisms to deal with necessary differences between platforms.

Language Concepts Guiding Principles

What we tried to keep in mind and should carry forward into 2.0 standardization

- **Support for Reuse**

  The quality of being able to reuse scenarios and parts of scenarios across multiple levels of abstraction, platforms, and use cases without undue adjustments needing to be made.

  This comprises suitable library concepts and mechanisms in the language, as well as the way core language features are structured to foster reuse.

- **Extendibility**

  The quality of allowing for easy extension of the underlying language mechanisms and domain model towards new requirements, including new traffic participants, their models and properties, compositions, and constraints.

- **Migration from OpenSCENARIO 1.0**

  The language should support a migration path from OpenSCENARIO 1.0. This includes the ability to transform OpenSCENARIO 1.0 scenarios into OpenSCENARIO 2.0 scenarios, without necessarily requiring direct backward compatibility.

Overall Structure

Necessary Mechanics of the Language

Overall Structure

Minimalistic set of things required for the basic operation of the language

- Current surface syntax (for examples): Python-like
- Seperate modules, one per file
- Import of other modules by name, library concept for namespacing and location to be developed
- Remainder of module consists of type definitions and extensions
- Generic complement of scalar types, plus:
  - Physical types, including units
  - List types (i.e. ordered collections of one member type)
  - String type
  - Compound types:
    Structs, Actors, Scenarios, Modifiers
- Generic set of expressions (arithmetic, relational, logic, …)
- Parameters and Constraints

Type Definitions

Types in the Language, Inheritance, Extension

Compound Type Definitions

Kinds of Compound Types

- **Structs**
  define a set of related data, similar to a *class* in other object-oriented languages.

- **Actors**
  define an agent that is capable of action and whose state changes over time.
  They are used to model physical entities such as cars, pedestrians and their environment. More abstract objects such as traffic and weather are also modeled using actors. Actor states are modeled using fields that are updated to reflect a ground-truth as provided by a simulator or other reporting mechanism (e.g. telemetry). Actor behavior is modeled using scenarios.

- **Scenarios**
  define behavior attributed to a particular actor. Scenarios define behavior by activating other scenarios, built up from a library of basic scenarios describing built-in actor behavior, such as moving, accelerating, turning and so on.

- **Modifiers**
  define changes to the behavior of the scenario invocations to which they are applied.

Compound Types Definitions

How to Define Compound Types

- All compound types define their own namespace
- Compound type defintions give the type a name, an inheritance relationship, and a set of member definitions:

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

- Members can comprise (at a minimum):
  - Field Declarations (incl. Parameters)
  - Constraints
  - Cover Defintions
  - Event Declarations
  - External Method Declarations

- Scenarios are declared in the context of an actor. While they define a namespace for their features, the parent actor namespace is accessible too:

```
scenario <actor>.<scenario-name>:
    <members>
```

- Scenarios have all the features of structs and actors, in addition they can invoke scenarios and modifiers.
- Invoked scenarios can be built-in or external, user-defined ones.
- A special kind of built-in scenarios are *operators*, that create temporal relationship between the scenarios.

```
scenario traffic.two_car_drive: # Scenario header
    path: path # a route on the map
    car1: car # an instance of car
    car2: car
    do parallel: # operator - members execute concurrently
        car1.drive(path) # drive() is a scenario of car
        car2.drive(path)
```

Compound Type Definitions

Inheritance and Extensions

- **Simple (unconditional) inheritance**

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

- **Type extension**

```
extend <type-name>:
    <members>
```

```
actor car:
    car_length: distance   # or any other feature
extend car:
    car_weight: weight
    keep(car_weight < 2500kg)
```

- **Conditional inheritance**

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

```
actor vehicle:                           # the base type
    category: vehicle_category           # truck, car, motorcycle etc.
    emergency_vehicle: bool

actor truck: vehicle(category: truck): # conditional, only for truck
    num_of_trailers: uint with: keep(it in [0..2])
```

Parameters and Constraints

Controlled Abstraction and Concretization

Parameters and Constraints

Controlled Abstraction and Concretization of Scenarios

- The rationale for **parameters** is to enable the definition and use of **abstract** OpenSCENARIO entities (such as scenarios, actors, models).
- An **abstract entity**, in the context of parameter-zation, is an entity with *at least one piece of information* that is necessary to invoke or instantiate the entity is *not yet fixed to one concrete value*.
- Any such piece of information which is left open in the definition of the entity is called a **parameter** of the entity.
- In contrast to an abstract entity, a **concrete entity** is an entity with all information, that is necessary to invoke or instantiate it, unambiguously defined.

```
actor my_car:
    width: distance # width is an attribute (a piece of information) which
                    # is required to instantiate an entity of type my_car

    c: my_car       # c is defined as an abstract car with parameter width
    wide_c: my_car with:     # wide_c is defined as a concrete car
        keep(width == 3meter)
```

- Given a scenario with n parameters, the n-dimensional space of all possible/useful/legal n-tuples of concrete parameter values is the **parameter space** of the scenario.
- Such parameter spaces are expressed by **parameter type** definitions and **constraints**.
- Special cases of constraints are **parameter ranges**.

```
p1: speed                   # parameter type
p2: speed
keep(p1 > 10kph)            # a simple constraint
keep(p2 in [30..50]kph)     # a parameter range
keep(p1 < p2)               # a constraint involving 2 parameters
```

Parameters and Constraints

Controlled Abstraction and Concretization of Scenarios

- Besides the parameter space definition through parameter types and constraints, the choice of concrete values of parameters may also be influenced by **parameter probability distributions**.

```
keep(soft v_desired == normal(30, 5))
```

- Depending on the constraints it may or may not be possible to choose concrete values for all parameters before the start of the top-level scenario execution.

- In any case, all parameters of an entity must be chosen at the latest when the respective entity gets instantiated/invoked.

- Together these concepts support:
  - The definition of **logical scenarios** in the sense of the **Pegasus** project, which represent a multitude of (similar) concrete scenarios using a parametrized presentation.
  - **Ruling out useless parameter values** and combinations thereof when generating concrete scenarios from an abstract (parameterized) definition.
  - Applying **stochastic methods** for choosing parameter values, e.g. to account for real-world scenario likelihoods when creating test plans.
  - **Referencing** static content of the **road network** without prior knowledge of the concrete road network that will be used for a scenario execution.
  - **Modular re-use** of scenarios.

Scenario Building Blocks

The Parts of a Scenario and how to Compose

Scenario Building Blocks

What are Scenarios Composed of?

- **Actors**
  The acting and reacting entities/objects of a scenario
- **Actions**
  The basic operations that actors are able to perform. Actions are atomic in the sense that they are not further decomposable into smaller parts, i.e. they are atomic scenario invocations.
- **Atomic Phases**
  Instantiations of actions of certain actors enriched by modifiers, consisting of:
  - an ***Actor*** or ***Group of Actors*** performing it
  - an ***Action*** (e.g. drive) that is to be performed
  - a set of ***Modifiers*** that direct **in what way** the action is to be performed
  - references to ***Events / Conditions*** that define **when** actions should occur
- **Parametrized Invocation of User-Defined Scenarios**
  allowing for reusability of scenarios e.g. defining an overtake scenario and using them in different contexts
- **Temporal Composition Operators**
  (as e.g. serial, parallel, etc ⇒ see Section 5.5.4) to create a new phase by composing
  - atomic phases,
  - invocations of other scenarios, or
  - arbitrary compositions of atomic/complex phases and scenario invocations
- **Constraints**
- **External Methods**
  with a well-defined interface (defined inputs and outputs) that may be written in any other programming language

Modifiers

Influencing the Behavior of Particular Scenario Invocations

- Modifiers are used to influence (i.e. modify) the behavior of particular scenario invocations, including actions as well as composite scenarios.
- They can be seen as structured specifications for intended behavior, which are interpreted by the scenario; they do not consume simulated time by themselves. More precisely, they are stateless; there are no state variables associated with modifier instances.
- Some modifiers can only be embedded in specific scenarios. In the following example, a user-defined modifier can be applied to drive() only, as indicated by the of keyword in the header:

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

- Once defined, the modifier can be used with various values, for example:

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

Temporal Composition Operators

How to Compose Scenarios using Temporal Composition

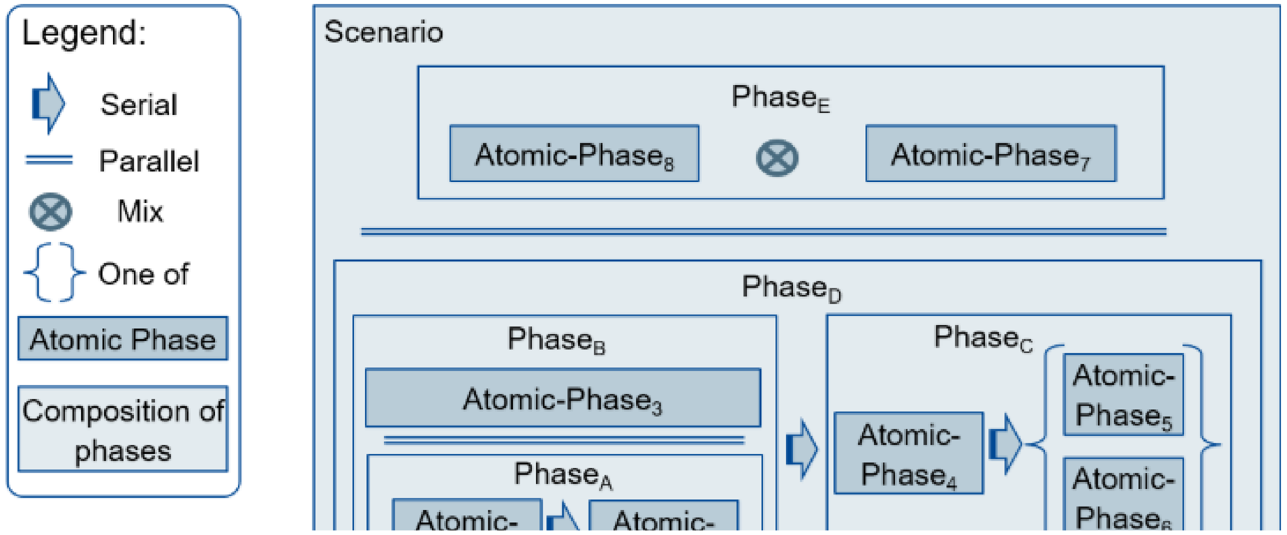
- Scenarios can be composed out of other (sub-)scenarios using temporal composition operators, which operate on atomic phases
- Operators comprise serial, parallel, mixing, selection (one of) and repetition of atomic phases
- Synchronization can be achieved using events

Example for mixing phases with synchronization

```
do mix():
    P1: Accelerate()
    P2: Change_Lane() with:
        synchronize(@drive&
```

Example for starting a phase on an

```
serial:
  wait @lane_ahead_free() #
  accelerate()
```



Example for until event

```
Drive_fast_until_it_rains: car1.drive() with:
    speed([115..120]kph)        # car1 should drive 115..120kph
    until(@rain_starts)         # until it starts raining
                                # (event that is emitted by another phase)
```

Pierre R. Mai
Owner / Director, PMSF IT Consulting

Phone: +49 8161 976 96 - 11
Email: pmai@pmsf.de