efmi

Functional Mock-up Interface for embedded systems

eFMI® Standard

Version 1.0.0 Beta 1 (candidate-draft), June 4, 2024

Contents

Preamble

.1. Abstract

The eFMI (FMI for embedded systems) standard specified in this document aims to extend the scope of FMI (https://fmi-standard.org) from simulation towards software development. The eFMI standard is intended as exchange format for workflows and tool chains from *physical models to embedded software*. It is defined as a layered approach built upon the *FMI for Co-Simulation* standard (any version). An eFMI component, that is an eFMU (Functional Mock-Up Unit for embedded systems), can be packed in different formats. Especially, an eFMU can be packed as FMU and can then be simulated with any FMI compliant tool (https://fmi-standard.org/tools) to perform Software-in-the-loop (SiL) testing. Code generation for an embedded device requires however dedicated tool support for eFMI.

This effort is motivated by the fact that especially the development of advanced control functions and diagnosis functions can benefit from physical models. As of today the realization of such model-based functions incorporating physical models, in the following refered to as physics-based functions, is very involved. The expertise from the physical modeling domains, control design and numerics for real time applications are required as well as implementation knowledge in terms of rules & regulations for embedded software have to be taken into account in order to supply an industry grade function on an embedded device.

The eFMI standard describes a container format that will allow to exchange models in a variety of different types of model representations:

$u(t_i)$ → sampled data system → $y(t_i)$

$$x_{i+1} = f_x(x_i, u_i)$$
$$y_i = f_y(x_i, u_i)$$
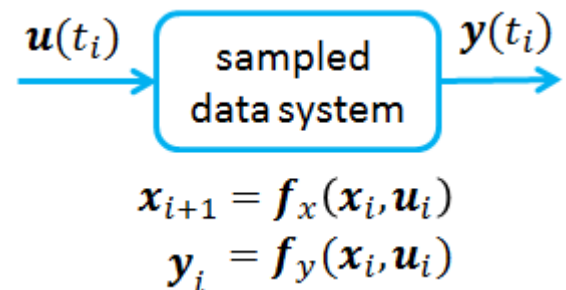
- The *Algorithm Code* representation describes the mathematical model in a *target and implementation independent* fashion as input/output, sampled data block with one fixed or variable sample time using the standardized intermediate language *GALEC* (Guarded Algorithmic Language for Embedded Control) developed for this purpose. GALEC is based on a small subset of *Modelica functions* together with changes and extensions as needed for *embbeded real-time systems*. GALEC code can be *scrambled* to provide a certain degree of Intellectual Property protection. *Physical modeling tools* should be able to generate this representation with reasonable effort.

- The *Production Code* representations allow to ship *C* or *C++* code within the same container, either as nearly target-independent generic code and/or as highly optimized target specific code. Contrary to FMI, there is no standardized API (getX, setX, doStep, ...), but a description of the actual code interface to allow the code to be integrated into existing software architectures with minimal calling overhead. When an eFMI is packed as FMU, an FMU wrapper is added to a selected code representation. *Software development tools* should be able to provide the

transformation from an Algorithm Code to one or more Production Code representations with reasonable effort.

- The *Binary Code* representations provide target specific *executable codes.* These code representations naturally provide the best Intellectual Property protection.

- The *Behavioral Model* representation provides *references results* for different scenarios to allow *automatic tests* of the Production and Binary Code representations. In the future this representation might be extended to include the original model from which the eFMI representations are derived, or computable scenarios might be added in form of FMUs.

By means of one global content XML description of all parts of an eFMU and by one XML manifest file for every eFMI representation shipped in an eFMU, a highly flexible and extensible mechanism is provided that allow to integrate eFMUs into arbitrary software architectures being deployed to any kinds of execution environment, including for example AUTOSAR or adaptive AUTOSAR.

.2. Overview

This document specifies the eFMI (FMI for embedded systems standard) with references to the FMI (Functional Mock-Up Interface) standard (https://fmi-standard.org/)

In section Section .3 the development of the eFMI standard and its intended usage is motivated.

The technical key concepts with reference to the current FMI standard are explained in section Chapter 1 for the better understanding of the later sections.

Thereafter the eFMI standard is specified starting with the description of the overall container structure of an eFMU (Functional Mock-Up Unit for embedded systems) in section Chapter 2.

The following sections Chapter 4, Chapter 3, Chapter 5, Chapter 6 are dedicated to the different types of model representations supported by eFMI. Each description consists of an introductory section followed by the specifications of the corresponding meta data and language:

- The *Behavioral Model* representation provides reference results to allow automatic verification of the Production and Binary Code representations.

- The *Algorithm Code* representation describes the mathematical model of *discrete-time*, sampled data, input/output blocks in a *target and implementation independent* fashion with the standardized intermediate language *GALEC* (Guarded Algorithmic Language for Embedded Control - a small subset of the Modelica language (https://www.modelica.org/modelicalanguage) with extensions as needed for embbeded systems).

- The *Production Code* representation defines one or more mappings of an Algorithm Code representation to C or C++ Code (for example 32-bit and/or 64-bit representation of floating point numbers, generic ANSI C-Code and/or code specialized to a particular target environment like AUTOSAR and/or specific target processors).

- The *Binary Code* representation provides one or more target specific *executable codes* for one production code representation.

In the following image an overview of the eFMI representations is given, together with examples for potential tool chains:

eFMI representations and examples for tool chains
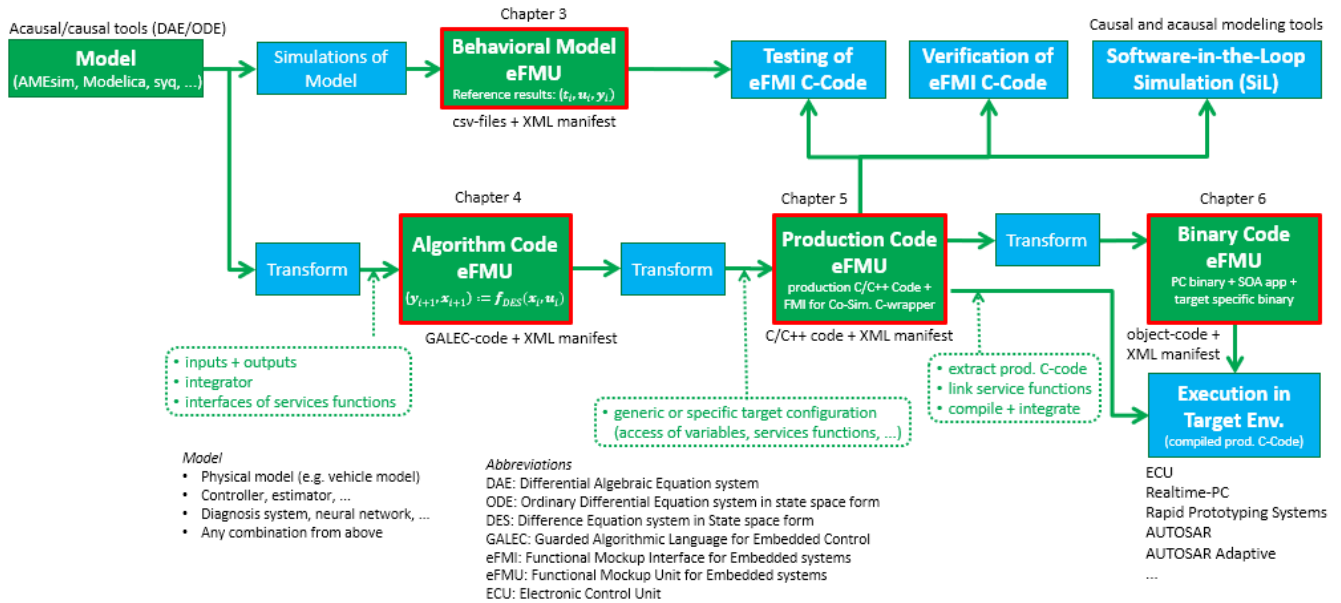
This standard document is accompanied by the following open source codes and files to allow tools to more easily support the eFMI standard:

- *XML schema files* for all xml manifest files defined in this document.

- An *eFMI compliance checker* in form of a Python library, to check compliance of eFMUs (Functional Mock-Up Units for embedded systems) with this specification.

- The *eFMI_TestCases Modelica package* providing > 20 dedicated Modelica models and variants of them to test eFMI tool chains.

- The *eFMI Modelica package* providing all eFMI builtin-functions as Modelica functions with a Modelica implementation, in order that Modelica models can use these functions.

- *ReferenceResults* for the models of the *eFMI_TestCases* library in form of > 50 csv files.

- eFMUs for the *eFMI_TestCases* library generated with various tools.

.3. Introduction

The goal of the eFMI standard (FMI for embedded Systems) is to enhance Production Code of embedded control systems by physics-based models in an automated way. This shall improve the performance of the underlying systems, reduce the maintenance costs and increase the productivity of software development for embedded systems.

Embedded software is commonly used on ECUs (Electronic Control Units) to control or monitor a system. In these cases it is beneficial to incorporate knowledge of the system behavior into the function. Physical models aim to describe the behaviour of the system for a given range of operation. These models are well described by differential- and algebraic equations or can be approximated by projection on a neural network.

Physical models can be utilized to achieve a significantly better performance of the system in applications such as:

- *observers/virtual sensors* (e.g. extended and unscented Kalman filters, moving horizon

estimation),

- *model-based diagnosis* (e.g. signal based fault detectors, linear/nonlinear residual generators),

- *feedback and feedforward controllers* (e.g. linear controllers with gain scheduling, nonlinear inverse models, nonlinear dynamic inversion, feedback linearization, linear/nonlinear model-predictive control),

- *neural networks* to approximate physical models and/or the above applications.

These types of functions are typically hand-coded software implemented and tested in an elaborate and time-consuming fashion. The eFMI standard aims to provide model exchange capabilities that allow to transfer physical models created in dedicated modeling and simulation tools to embedded code generating tools for ECU software. This enables an end to end workflow from physical modeling to the deployment of the software function on an embedded device.

The eFMI standard is an open standard based on the FMI standard (Functional Mock-Up Interface, https://fmi-standard.org/). eFMI components are able to interoperate with software components according to the automotive embedded system standards AUTOSAR (https://www.autosar.org/standards/classic-platform/) and Adaptive AUTOSAR (https://www.autosar.org/standards/adaptive-platform/). Generated code shall refer to typical safety measures and coding guidelines, e.g. in the Automotive industry the ISO 26262 and MISRA-C 2012 for Autocode (https://www.misra.org.uk/Activities/MISRAAutocode/tabid/72/Default.aspx).

Different types of model representation shall allow to separate the concerns of deriving a proper computation algorithm and its compliant implementation for an embedded device. The container architecture and rich meta information, extending the FMI model description, support the integration in existing development processes and tool chains.

Chapter 1. General concepts

This section describes the general concepts of the eFMI standard

The goal of the standard is to extend the existing FMI standard to the embedded domain. The FMI standard is focused on simulation of models and model parts, on few standardized execution platforms (Windows, Linux) with well known tool chains. With this context in mind, the FMI standard does not consider any constraints with respect to resource consumption or run time characteristics of the model.

In contrast there is a considerable diversity of embedded platforms, each with their own constraints with respect to runtime performance, memory limits or available compiler support. Given these additional constraints the goal of the FMI standard "Compile once, run everywhere" is neither feasible nor desirable.

A further aspect is the use of models not only for the sake of simulation but in a broad application range, from advanced control strategies like model predictive control to model based diagnosis. The eFMI standard must consider these aspects and is therefore designed as an extension to the FMI standard as described in the following.

1.1. Comparing FMI with eFMI

A major enhancement of the eFMI standard in comparison to the FMI standard is the introduction of different abstraction levels. The FMI standard is based on an executable C Code with an interface of fixed and well defined functions (like getX, setX and doStep). This approach is well suited for the purpose of simulation on a standardized platform (either Windows or Linux).

However, such an approach is not very suitable for (deeply) embedded code due to the following reasons:

- Support of a diverse number of execution targets.

- Support of a diverse number of compilers.

- Integration of the code into existing code structures (in the following we will call this the "Software context") with minimum overhead in data passing and function calling.

For this reason one fixed C Code (or one fixed executable) representing the implementation is not sufficient. Instead the eFMI supports the concept of **several C Code implementations** (or also binary implementations), each with a description of the interface of the C Code. These descriptions are defined in so-called **manifest** files and are bundled with the corresponding code files into a Production **code container**. More details on these manifest files can be found in the section on Production Code manifests (Section 5.1). Here you will also find examples demonstrating the influence of the software context onto the generated code and manifest descriptions.

An FMU represents exactly one model (implemented by the C Code or executable). The same shall be true also for an eFMU despite of the fact that it may contain any number of C Code implementations, and additionally, it shall be easily possible to add further implementations (e.g. for different targets or software contexts) into the eFMU at any time.

This requirement is enabled by adding a higher level abstraction to the eFMU, namely the "Algorithm Code".

The **Algorithm Code** contains an abstracted description of the function(s) to be computed, and serves as the input to generate the C Code implementations. The functions are described in a pseudo programming language (influenced by *Modelica functions*), and the meta data is also given in a manifest file. The Algorithm Code is a solution to a causalization of this system by specifying

- Causalization: the input/output behaviour of the system.

- Discretization: discretization of differential equations (use of solver, time discretization).

The Algorithm Code is organized in code containers in the eFMU, similar to the Production Code container. For more details on the organization of these containers to form a valid eFMU, please see the section on container architecture (Chapter 2).

The following table summarizes the differences between FMI and eFMI.

Topic	FMI	eFMI
Goal	(co-) simulation	efficient ECU implementation
Execution platform	standardized (Windows (.dll), Linux)	diverse: different ECUs, different compilers
Reuse	"as is" in "all" simulation environments	highly limited (therefore several implementations possible)
Interface	fixed based on standardized API (getX, setX, doStep, ...)	Only fixed on ideal algorithmic level (GALEC program of Algorithm Code container) but not C production code (derived Production and Binary Code containers); actual interface of production code described in Production Code manifest.
Implementation	one implementation (source code) compiled for different standard platforms	any number of implementations compiled for arbitrary many targets (vendor and target instruction set architecture (ISA) dependent)
Abstraction level	C Code level	Abstract model representation algorithm (Algorithm Code) in addition to (derived) C Code implementation (Production Code)

1.2. FMI compliance

An important fact is that despite the broadened scope of the eFMI, an eFMU can be packed into an

FMU. This is achieved by taking a distinguished Production Code level implementation and wrapping this to an FMI compliant interface with corresponding model description file. Surely this Production Code level implementation must be target independent and suitable for simulation targets like Windows or Linux.

1.3. Functions in eFMI

In the following different kinds of functions considered in the eFMI standard are described. It is mentioned for which model representation a certain function kind is available. Differences between the kind of functions and consequences and requirements for e.g. transformation tools are also covered.

1.3.1. Block methods

(Available in Algorithm Code and Production Code model representation)

The Algorithm and Production Code model representation is mathematically defined as a sampled input/output block with one (potentially varying) sample period for the whole block. All variables of the block have a defined type and all statements of the block are sorted and explicitely solved for a particular variable. Three block *methods* are defined, so functions that operate on the same memory `self` that is exchanged between the function calls. Especially, methods are provided to initialize the `self` memory with function **Startup** and to perform one step at the actual sample instant with method **DoStep**.

The block methods are defined in the Algorithm Code representation. A Production Code generator translates these methods to C-functions. It is also possible to define Production Code interface functions directly in C, without providing an Algorithm Code representation.

On Production Code level the block methods are highly integrated in the environment provided by the embedded control unit (ECU). For example, if the ECU provides input signals at certain addresses in memory or the parameters are part of an overall global C-struct. Consequently the actual implementation/interface of the methods is at liberty of the Production Code generating tool.

1.3.2. Built-in functions

(Available in Algorithm Code and Production Code model representation)

Built-in functions are functions with well defined syntax and semantics in the eFMI standard. This includes elementary functions such as **sin**, **cos**, **log**, **exp**, but also functions to solve linear equation systems in various ways, for example

```
1   x := solveLinearEquations(A, b);
```

to solve the linear equation system $A*x = b$ with regular A matrix for x.

Built-in functions can be used in Algorithm Code or Production Code. All built-in functions that are supported by the eFMI standard are defined in Section 3.2.6. The names of the built-in functions are reserved and must not be declared by the user.

A tool that transforms Algorithm Code into Production Code doesn't need additional information for those functions, because their syntax and semantics are clearly defined thus the tool knows how to handle it.

1.3.3. Local functions

(Available at Algorithm Code and Production Code level)

In Algorithm Code, local functions can be defined together with the physics-based model that underlies the eFMU. A local function is formally defined with the GALEC language, see section Section 3.2. A Production Code generator generates a C-function from this definition. Alternatively, a local function can be provided as C Code, together with a GALEC wrapper that defines how the call of the GALEC function is mapped to C (the syntax and semantics is identical to the Modelica external function interface). The declaration of the logical function interface must be provided in the corresponding manifest file.

Example of a local function implemented with the GALEC language:

```
1  function add
2    input  Real u1;
3    input  Real u2;
4    output Real y;
5  algorithm
6    y := u1 + u2;
7  end add
```

Example of a local function wrapper with the GALEC language around a C-function:

```
1  /* GALEC function wrapper */
2  function dot  /* scalar product */
3    input  Real v1[:];
4    input  Real v2[size(v1,1)];
5    output Real y;
6    external "C" y = dot(size(v1,1), v1, v2)
7  end dot
8
9  /* C Code signature */
10 float_t dot(const int32_t n, float_t const v1[], float_t const v2[]);
```

Chapter 2. eFMU container architecture

An eFMU can be packed in different formats. The basic structure of the eFMU specific part is always:

```
<eFMU root directory>  // depends on the package format
    // Directories for eFMU model representations (tool specific)
    schemas           // directory with the used eFMI schemas
    __content.xml     // defines the eFMU folder structure
```

The only required names are the file name `__content.xml` and the directory name `schemas` at the root of the eFMU folder. All other directory and file names are defined by the eFMU generation tool. The used directory and file names are stored in the `__content.xml` file and can therefore be deduced by reading this file.

The following eFMU package formats are defined:

1. The `<eFMU root directory>` is a standard directory in the file system.
   [This is useful to hold an eFMU in a text-based version control system, such as github, gitlab or svn.]

2. The `<eFMU root directory>` of (1) is zipped with the efmu-content, especially `__content.xml`, at the root of the zip-file. The zip-file has the extension `.efmu`.
   [This packaging is useful to ship or distribute an eFMU.]

3. The `<eFMU root directory>` of (1) is path `extra/org.efmi-standard` inside a standard FMU (Functional Mock-Up Unit) of any FMU type and any FMU version. The path is defined according to the FMI 3.0 specification. With attribute `activeFMU` inside the `__content.xml` file it is defined which of the Algorithm, Production or Binary code representations is used as basis of the FMU.
   [This package format is useful to ship or distribute an eFMU for Software-in-the-Loop simulation with any suitable FMU tool.]

Note, Algorithm Code, Production Code and Binary Code representations can optionally store associated FMUs. For example Algorithm Code can store a Model-in-the-Loop FMU and Production Code can store one or more Software-in-the-Loop FMUs for different targets. In order to execute these FMUs directly, an eFMI tool is needed. Otherwise, one of the stored FMUs can be selected for package format (3) in order that any FMI-tool can simulate this specific FMU.

Example:

An eFMU could be stored as zip-file with extension `.fmu` having the following internal structure:

```
modelDescription.xml     // required FMI file
// optional FMI directories and files
extra                    // extra directoy of FMI 2.0 and 3.0
    org.efmi-standard    // eFMU root directory
        // tool specific directories, e.g. AlgorithmCode
        schemas          // directory with the used eFMI schemas
        __content.xml    // defines the eFMU folder structure
```

An eFMU may contain any number of additional subdirectories below the `<eFMU root directory>`, each subdirectory being a container of some model representation type. An eFMU must contain exactly one Algorithm Code Model container; of any other model representation type arbitrary many containers can be given, including the special cases of none. Each container can be structured arbitrarily into further subdirectories and files, but must have a dedicated manifest file at its root.

2.1. Content description (efmiContainerManifest.xsd)

The `__content.xml` file is the registry for all model representations in the eFMU container. It has the following schema definition:
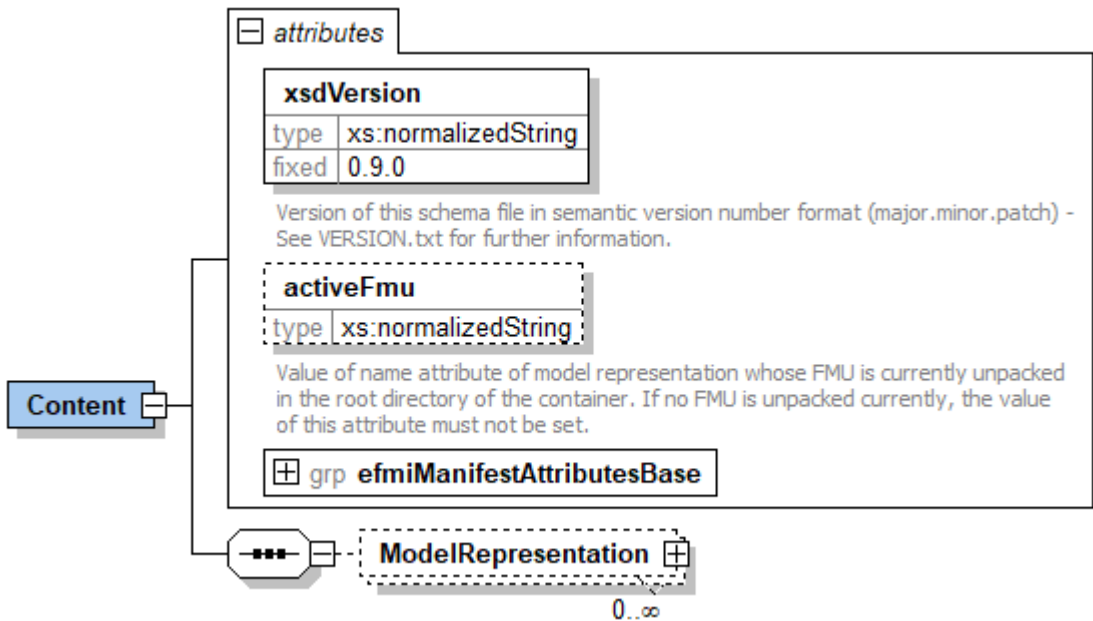
Name	Description
xsdVersion	The version of the XSD Schema of the `__content.xml` in *Semantic Versioning 2.0.0* format (https://semver.org).
activeFMU	Value of name attribute of model representation whose FMU is currently unpacked in the root directory of the FMU. If no FMU is unpacked currently, the value of this attribute must not be set.
efmiManifest AttributesBa se	A group of attributes that is identical for all manifest files. For details see [ManifestAttributesBase].

Each model representation that is a part in the eFMU container must have a corresponding entry in the `__content.xml` file with the following information:

Name	Description
name	Unique name of the container, also defining its root directory name.
kind	The type of the model representation. The allowed values are `AlgorithmCode`, `ProductionCode`, `BinaryCode`, `BehavioralModel`.
manifest	Name of the container's manifest file. The manifest is located in the container's root directory, cf. "name" attribute.
checksum	SHA-1 hash — according to the *Secure Hash Standard (SHS), FIPS PUB 180-4* — of the binary content — i.e., content as is without any modifications like operating environment specific line-ending or encoding localizations — of the manifest file of the model representation. The given checksum must be correct; if it is not the SHA-1 hash of the manifest file of the model representation, the eFMU is invalid. A checksum of the whole content of the model representation container is not required, because the files belonging to the container and their respective checksums are listed in its manifest.
manifestRefId	The unique GUID of the manifest file (= corresponding attribute of ManifestReference). References a manifest using the Manifest elements `id` attribute. This information has been added for technical purposes only to speedup resolving references between manifest files via the `manifestRefId` outlined below. Otherwise, following an inter-manifest reference (via a `manifestRefId` used in the source manifest) would demand to read other manifest files until a manifest with the desired `id` is found).

The following is an example of such a content file:

```xml
 1  <?xml version="1.1" encoding="utf-8"?>
 2  <Content xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 3          xsi:noNamespaceSchemaLocation="schemas/efmiContainerManifest.xsd"
 4          xsdVersion ="0.9.0"
 5          efmiVersion="1.0.0"
 6          id         ="{92b7edbe-e77d-419a-8457-bf8d452a98f6}"
 7          name       ="MyModel"
 8          generationDateAndTime="2021-02-27T15:43:25Z"
 9  >
10      <ModelRepresentation kind         ="ProductionCode"
11                           name         ="TLGeneratedCode_v1"
12                           manifest     ="mark.xml"
13                           checksum     ="e29810938a2a535dc8f6f9b8f51c5febe834ee01"
14                           manifestRefId="63f8c810-f008-47f0-a4b6-7a243f83e46b" />
15      <ModelRepresentation kind         ="AlgorithmCode"
16                           name         ="algoCode_v1"
17                           manifest     ="luke.xml"
18                           checksum     ="e29810938a2a535dc8f6f9b8f51c5febe834ee05"
19                           manifestRefId="63f8c810-f008-47f0-a4b6-7a243f85e46b" />
20      <ModelRepresentation kind         ="BinaryCode"
21                           name         ="binCode_v1"
22                           manifest     ="matthew.xml"
23                           checksum     ="e29810938a2a535dc8f6f9b8f51c5febe834ee08>"
24                           manifestRefId="63f8c810-f008-47f0-a4b6-7a243f85e47b" />
25  </content>
```

This `__content.xml` file describes therefore the following directory structure:

```
<eFMU root directory>
    TLGeneratedCode_v1
        mark.xml
    algoCode_v1
        luke.xml
    binCode_v1
        matthew.xml
    schemas              // directory with the used eFMI schemas
    __content.xml        // the xml-file of the example above
```

This example just demonstrates that the folder names of the model representations and the manifest file names are defined by the generating tool. Typically, more descriptive names would be used, such as:

```
<eFMU root directory>
    BehavioralModel
        manifest.xml
    AlgorithmCode
        manifest.xml
    ProductionCode_Generic_C_Float32
        manifest.xml
    ProductionCode_Generic_C_Float64
        manifest.xml
    ProductionCode_Autosar_Float32
        manifest.xml
```

```
    schemas
    __content.xml
```

2.2. Structure of Model Representations

Each model representation can have its own flexible structure. Its content and the structuring of information is described in the manifest file (for details on specific manifest files for the different kind of model representations refer to the corresponding sections). Which file in a model representation is its manifest file can be found as the reference entry in the `__content.xml` file. The manifest file must be located in the model representation's root folder.

eFMI allows for having model representations consisting of a manifest file only, hence information should not be doubled. For example, a tool generating directly a Production Code Model Representation must also generate an Algorithm Code Model Representation, because information relevant for Algorithm Code is stored only in the corresponding manifest file and not in the Production Code manifest.

2.3. Model Representation Manifests

The model representation manifests share the same guiding principles:

1. Entity names start with a capital letter

2. Attribute names start with a lower-case letter and use camelCase where needed.

3. Entities that serve as a group get the name of the grouped entities and an 's' as postfix.

4. Each entity that should be referred to has an attribute called `id`.

5. The type of an `id` attribute is an arbitrary string.

6. All `id` attribute values in a manifest file are unique.

7. References to other elements within or across manifest are established through attributes ending with "RefId". The value is the `id` of the referenced element.

8. For file references a string attribute is used and the value is interpreted as the relative path starting at the corresponding model representations root folder.

9. The context of a reference is specified in the definition of the manifest element and could be either within the same manifest (local context) or within the a referenced manifest (foreign context).

All manifests also share the principles outlined in the following sections:

2.3.1. Attributes of manifest files (efmiManifestAttributes.xsd)

The top-level element of a manifest file has the two attributes `xsdVersion` and `kind` that have a fixed value specific for the varying kinds of manifest. For example, for Algorithm Code manifests the two attributes must be:

xsdVersion	
type	xs:normalizedString
fixed	0.13.0

Version of this schema file in semantic version number format (major.minor.patch) - See VERSION.txt for further information.

kind	
type	ModelRepresentationKind
fixed	AlgorithmCode
enum	AlgorithmCode BehavioralModel ProductionCode BinaryCode

The attributes have the following meaning:

AName	Description
xsdVersion	The version of the XSD Schema of the manifest in *Semantic Versioning 2.0.0* format (https://semver.org).
kind	The type of the manifest. The allowed values are: `AlgorithmCode`, `ProductionCode`, `BinaryCode` and `BehavioralModel`.

Additionally, the top-level element of a manifest file has the following attributes (that are not specific to the manifest kind):

efmiVersion	
type	xs:normalizedString
fixed	1.0.0

efmi Version in semantic version number format

id	
type	efmiManifestIdentifierType

UUID of this manifest file.

name	
type	xs:normalizedString

Name of the block (e.g. name of the source block)

description	
type	xs:string

version	
type	xs:normalizedString

Version of the block (e.g. version of the source block)

generationDateAndTime	
type	xs:dateTime

Last modification date and time of manifest.

generationTool	
type	xs:normalizedString

Tool that was used to create or modify the Manifest and its files. Use "manual" if these files were created or modified manually.

copyright	
type	xs:normalizedString

Information on intellectual property copyright for this Manifest and its files, such as "© MyCompany 2020"

license	
type	xs:normalizedString

Information on intellectual property licensing for this Manifest and its files, such as "BSD license", "Proprietary", or "Public Domain"

The attributes have the following meaning:

Name	Description
efmiVersion	The version of the eFMI Standard in *Semantic Versioning 2.0.0* format (https://semver.org).
id	The UUID for this manifest file.
name	The name of the block (controller, diagnosis system etc.) as used in the modeling environment from which the manifest file was created, such as "Modelica.Mechanics.Rotational.Examples.CoupledClutches".
description	Optional string with a brief description of the block.
version	Optional version number of the block as used in the modeling environment from which the manifest file was created. *[Example: "1.0"]*.

Name	Description
generationDateAndTime	Date and time of last modification of the manifest or any contents within its container. Must be an `xs:dateTime` in Coordinated Universal Time (UTC), i.e, without time offset, only using arabic numerals, with hours, minutes and seconds but no smaller fractions, satisfying the pattern *YYYY-MM-DDTHH:MM:SSZ* whereas *Y* is the year, *M* the month, *D* the day, *H* the hour, *M* the minutes and *S* the seconds. This format is satisfying ISO 8601-1:2019. *[Example:* `generationDateAndTime="2009-12-08T14:33:22Z"` *]*
generationTool	Optional name of the tool that created the manifest file. If the files have been created manually use `generationTool="manual"`.
copyright	Optional information on the intellectual property copyright for the manifest and code files. *[Example: copyright = "© My Company 2020"].*
license	Optional information on the intellectual property licensing for the manifest and code files. *[Example: license = "BSD license <license text or link to license>" or "Proprietary" or "Public Domain"].*

Note, optional attributes defined in the `__content.xml` file, hold also for the manifest files in folders below this file, if not redefined in a manifest file. For example, if attribute `license` is defined in the `__content.xml`, but in no other manifest file of this eFMU, then the defined license holds for all directories and files below the `<eFMU root directory>`. If, say, a Production Code manifest defines a `license` attribute, then this license holds for all directories and folders in this Production Code model representation, independently what is defined in the `__content.xml` file.

2.3.2. Listing of relevant other manifest files (efmiManifestReferences.xsd)

The information about the eFMU is layered into several model representations (e.g. Algorithm Code, Production Code). In order to allow cross referencing between these model representations, the manifest files to be referenced need to be registered in a manifest file of a certain model representation. For this the `ManifestReference` tag is used with the following attributes

Name	Description
id	Unique id of the manifest reference entry. This id is used to establish cross manifest references.
manifestRefId	The unique GUID of the manifest. *[Note, the name of the associated model representation in the `__content.xml` file is not used, in order to decouple the manifest files from the container manifest.]*
checksum	The expected SHA-1 hash — according to the *Secure Hash Standard (SHS), FIPS PUB 180-4* — of the binary content — i.e., content as is without any modifications like operating environment specific line-ending or encoding localizations — of the referenced manifest file. If the actual SHA-1 hash of the referenced manifest, as listed in the `__content.xml`, and the expected one are not equivalent, this container is stale (i.e., the referenced container changed since the last update of this container).
origin	Boolean flag to indicate if that referenced model representation is the one that was used to derive the current model representation.

Example:

```
1  <ManifestReferences>
2    <ManifestReference id          ="ID_1"
3                       manifestRefId="{63f8c810-f008-47f0-a4b6-7a243f85e46b}"
4                       checksum     ="e29810938a2a535dc8f6f9b8f51c5febe834ee05"
5                       origin       = true />
6    <ManifestReference id          ="ID_2"
7                       manifestRefId="{63f8c810-f008-47f0-a4b1-7a243f85222b}"
8                       checksum     ="b4b84af148e587b95300d7a734302d1b911a6e58"
9                       origin       =false />
```

```
10  </ManifestReferences>
```

2.3.3. Listing of files belonging to the model representation (efmiFiles.xsd)

Each manifest contains a list of the files that are part of its model representation. These files are listed in a manifest as follows in the `Files` elements tag.

A `File` element has the following attributes:

Name	Description
id	id of the file reference entry. This is id is used to refer to the file reference within the manifests.
name	Name of the file
path	Directory part of path to the file (relative to root of model representation). Value has to start with `./` and end with `/`.
needsChecksum	Boolean flag indicating if the file is considered in the checksum calculation (the default value is `"true"`).
checksum	Optional SHA-1 hash — according to the *Secure Hash Standard (SHS), FIPS PUB 180-4* — of the binary content — i.e., content as is without any modifications like operating environment specific line-ending or encoding localizations — of the file. Must be given, if, and only if, `needsChecksum="true"`. A given checksum must be correct; if it is not the SHA-1 hash of the file, the eFMU is invalid.

Name	Description
role	The role of the file in the model representation. This attribute is an enumeration with the following valid values: - "Code": File containing code (Algorithm Code, Production Code or Binary Code). - "Manifest": The manifest file itself. - "FMU": One and only zip-file that is an FMU-container. Any version and any representation of an FMU can be used (for example FMI for ModelExchange, or FMI for CoSimulation, or FMU with a DLL, or an FMU with C-Code). This representation is useful to directly utilize the FMU in any FMI-compliant tool. - "FMUFolder": The content of an FMU (so the files after unzipping an FMU). Any version and any representation of an FMU can be used. This representation is useful when an eFMU is stored in a version control system, such as github, gitlab or svn. - "ReferenceData": File containing reference data (for example a csv file that stores reference values of variables). - "other": All other files (for example an AUTOSAR description file *.arxml). Note, a description of the file can be stored in attribute description. NOTE: The enumeration values have been selected such that each value may be used on an arbitrary level of abstraction, that is kind of model representation. In the future, more enumeration values might be added.
description	An optional description of the file (especially if role = \"other\").
ForeignFile	See below.

Example of a list of files:

```
1  <Files>
2      <File id="ID_1" name         ="model.c"
3                      path         ="./code/"
4                      needsChecksum="true"
5                      checksum     ="b4b84af148e587b95300d7a734302d1b912a6e58"
6                      role         ="Code"/>
7      <File id="ID_2" name         ="model.h"
8                      path         ="./code/"
9                      needsChecksum="true"
10                     checksum     ="b4b84af148e587b95300d7a734402d1b911a6e58"
11                     role         ="Code"/>
12     <File id="ID_3" name         ="misra.doc"
13                     path         ="./code/"
14                     needsChecksum="true"
15                     checksum     ="b4b84af148e587b95300d7a734302d1b914a6e58"
```

```
16                  role         ="other"/>
17      <File id="ID_4" name      ="model.arxml"
18                  path         ="./code/"
19                  needsChecksum="true"
20                  checksum     ="b4b84af148e587b95300d7a734302d1b911a7e58"
21                  role         ="other"/>
22      <File id="ID_5" name      ="model.doc"
23                  path         ="./description/"
24                  needsChecksum="false"
25                  role         ="other"/>
26  </Files>
```

2.3.4. Referencing

Referencing inside a model representation

Reference attributes pointing to entities in the same manifest must fulfill the naming convention that the attribute name consists of the original entity name and adding "RefId" as postfix. The value of the reference attribute must thereby be a valid id in the given context of the reference attribute, meaning that the id must exist in the context and be of the right type. For example a value of reference attribute `variableRefId` is an id number in the same manifest referencing a variable. In the Production Code Model Representation manifest file shown below, the DataReference with ID_100 references the variable T with ID_33 using the attribute `variableRefId`.

Referencing files

Files play a certain role in the eFMU model representation and are listed in a `Files` element of each manifest. Referencing files inside a model representation is done by using a `FileReference` element that comes along with `Files` and `File` element itself and not using a `fileRefId` attribute only. The reason to use a certain `FileReference` element is that the element comes along with a `kind` attribute of type string to allow for specifying the kind of a file in more detail.
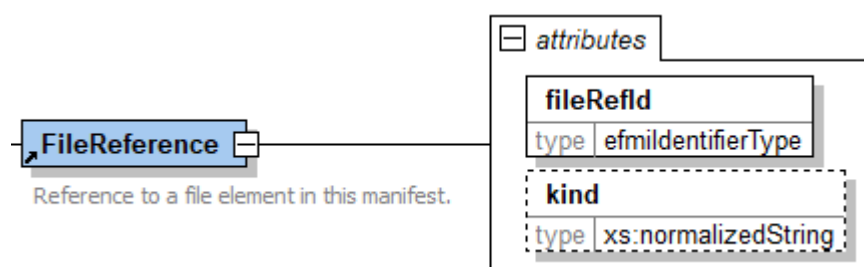
Name	Description
`fileRefId`	Reference to the id in the file overview
`kind`	Attribute for a more detailed specification of the kind of file used. The list of allowed values is not predescribed but should follow the guideline ????

```
1 <CodeFile id="ID_13" fileType="ProductionCode">
2     <FileReference fileRefId="ID_1" kind="SourceCode"/>
3 </CodeFile>
```

Note, that a `FileReference` attribute has no `id` attribute and therefore can't be referenced. This prevents transitive file referencing.

Referencing into other model representation - ForeignReference (efmiManifestReferences.xsd)

The eFMU describes one model on different levels of abstraction. Thereby the level of abstraction decreases in the following order

1. Behavioral Model
2. Algorithm Code
3. Production Code
4. Binary Code
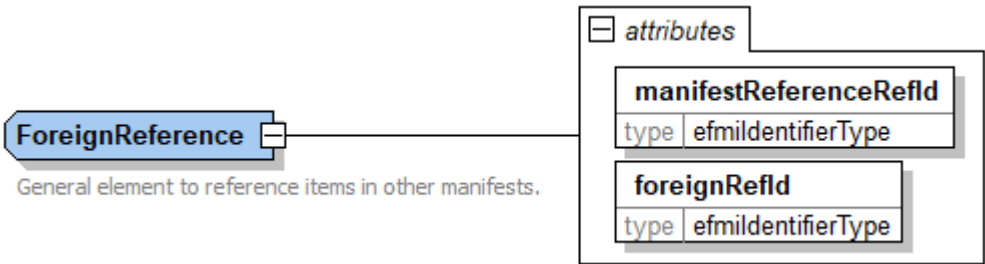
In order to establish cross referencing between these model representations, the "derived" model representation must include a `ManifestReference` to that model representation as described above. The consistency to the referenced one is ensured as follows:

The `manifestRefId` is used to retrieve the (current) model representation checksum of the entry in the `__content.xml` file. This (current) checksum can be compared with the (stored) checksum that is part of the ManifestReference to ensure consistency; if the checksums are not equivalent, the referencing container is stale (i.e., the referenced container changed since the last update of this container).

In order to cross reference into a referenced container's manifest, a `ForeignReference` element is present that has the following required two attributes:

Name	Description
`manifestReferenceRefId`	The (manifest local) id of a `ManifestReference`.
`foreignRefId`	The id inside the referenced manifest file.

Example:

```
1  <ManifestReferences>
2      <ManifestReference id          ="ID_1"
3                         manifestRefId="{63f8c810-f008-47f0-a4b6-7a243f85e46b}"
4                         checksum     ="e29810938a2a535dc8f6f9b8f51c5febe835ee05"
5                         origin       ="true"/>
6      ...
7  </ManifestReference>
```

```
 8   ...
 9        <Variable name      ="T"
10               id        ="ID_33"
11               typeDefRefId="ID_25"
12               pointer   ="false"
13               value     ="0.1"
14               const     ="false"
15               volatile  ="true"
16               static    ="false" />
17        <Variable name      ="_Clocks_interval"
18               id        ="ID_34"
19               typeDefRefId="ID_25"
20               pointer   ="false"
21               value     ="0.005"
22               const     ="false"
23               volatile  ="true"
24               static    ="false" />
25        <Variable name      ="gearRatio"
26               id        ="ID_35"
27               typeDefRefId="ID_25"
28               pointer   ="false"
29               value     ="105"
30               const     ="false"
31               volatile  ="true"
32               static    ="false" />
33   ...
34        <DataReferences>
35           <DataReference id="ID_100" variableRefId="ID_33" >
36              <ForeignVariableReference manifestReferenceRefId="ID_1"
    foreignRefId="ALG_ID_101"/>
37           </DataReference>
38           <DataReference id="ID_101" variableRefId="ID_34" >
39              <ForeignVariableReference manifestReferenceRefId="ID_1"
    foreignRefId="ALG_ID_100"/>
40           </DataReference>
41           <DataReference id="ID_102" variableRefId="ID_35" >
42              <ForeignVariableReference manifestReferenceRefId="ID_1"
    foreignRefId="ALG_ID_103"/>
43           </DataReference>
44   ...
```

In the example above (a cut-out of a Production Code Model Representation manifest file), the `manifestReferenceRefId` attribute (with value "ID_1") identifies the `ManifestReference` as the one that references the Algorithm Code Model Representation with the Manifest id "63f8c810-f008-47f0-a4b6-7a243f85e46b" in the eFMU container and the `foreignVariableRefId` attribute the element in that container with the given id (e.g. "ALG_ID_102").

It has to be checked, that the referenced ids actually are valid and are used for the objects of the right type.

Important restriction: The names of a variable can differ in the manifests of the Behavioral Model, the Algorithm Code, and the Production Code. But for input and output variables of the eFMI block, that are defined in the Algorithm Code manifest, the structure (e.g. scalar or vector or matrix) has to be preserved over the different model representations. It means, an output vector y in the Algorithm Code manifest corresponds to a vector with the same length in all other model representations.

Referencing Files in Foreign Model Representations (efmiFiles.xsd)

In cases where a file in another model representation is used without change in the current model representation, one should use `ForeignFile` elements in the `Files` list.
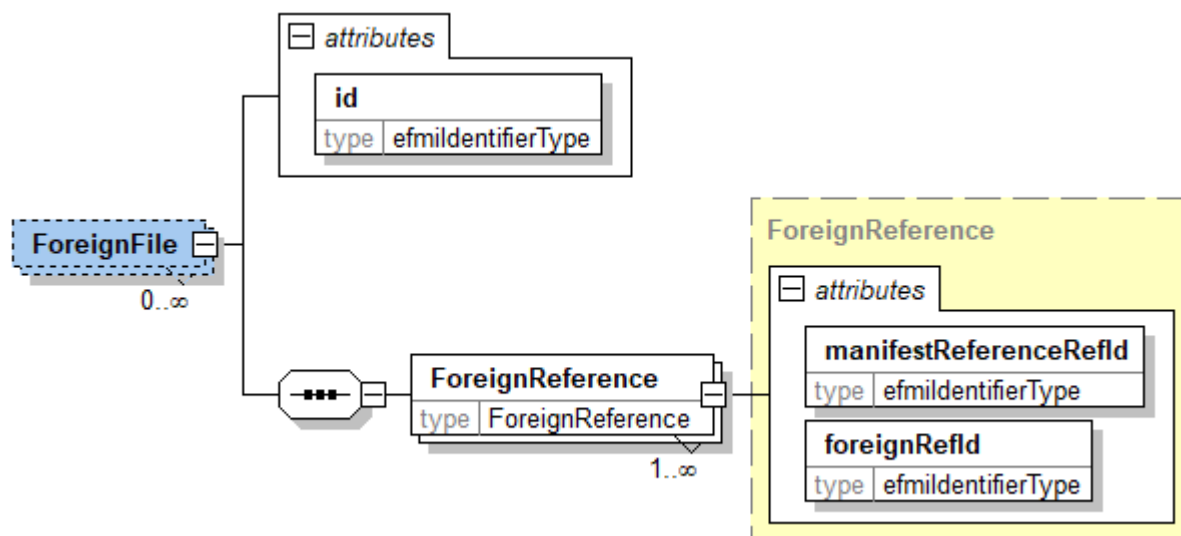
Name	Description
id	The (manifest local) id.
ForeignReference	Identifying the foreign manifest and the file inside the manifest.

Example:

```
 1  <ManifestReferences>
 2      <ManifestReference
 3          id="ID_0"
 4          manifestRefId="{63f8c810-f008-47f0-a4b6-7a243f85e46b}"
    checksum="e29810938a2a535dc8f6f9b8f51c5febe835ee05"
 5          origin="true"/>
 6      ...
 7  </ManifestReference>
 8  ...
 9  <Files>
10      <File id="ID_1" name="model.c" path="./code"
11          needsChecksum="true" checksum="e29810938a2a535dc8f6f9b8f51c5febe835ee05" role="Code"/>
12      <File id="ID_2" name="model.h" path="./code"
13          needsChecksum="true" checksum="e29810938a2a535dc8f6f9b8f51c6febe835ee05" role="Code"/>
14      <File id="ID_3" name="misra.doc" path="./code"
15          needsChecksum="true" checksum="e29810938a2a535dc8f6f9b8f51c5febe835ee06" role=
    "other"/>
16      <File id="ID_4" name="model.arxml" path="./code"
17          needsChecksum="true" checksum="e29810938a2a535dc8f6f9b8f51c5febe835ee06" role=
    "other"/>
18      <File id="ID_5" name="model.doc" path="./description"
19          needsChecksum="false" role="other"/>
20      <ForeignFile id="ID_6">
21          <ForeignFileReference manifestReferenceRefId="ID_0"
22                              foreignRefId          ="ID_26" />
```
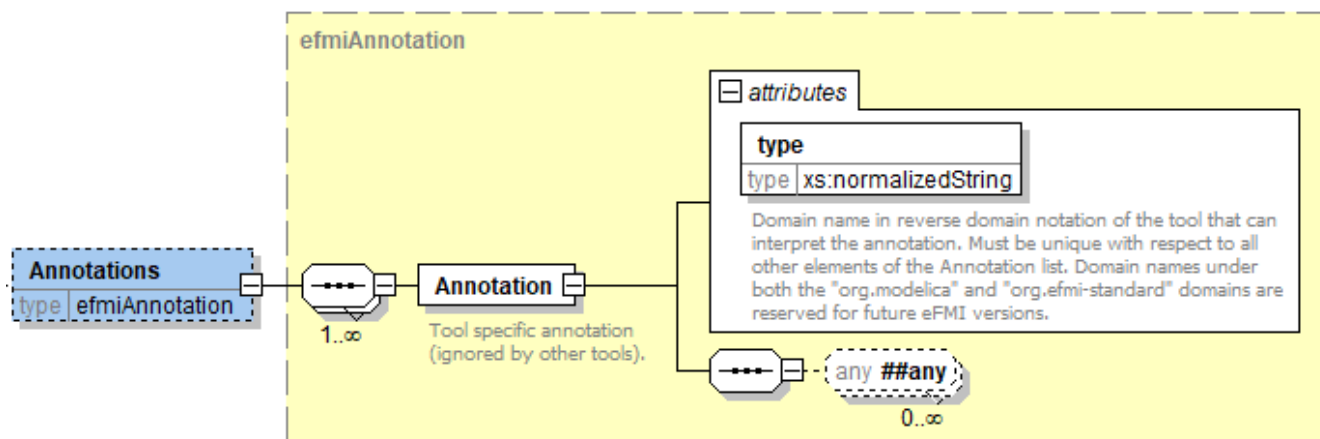
```
23      </ForeignFile>
24  </Files>
```

Annotations

Optional, non-standardized, vendor-specific information — that a vendor might want to store and other vendors can ignore — are defined via `<Annotation>` elements in an optional `<Annotations>` list. The XSD is defined in `efmiAnnotation.xsd` and aligned to the corresponding elements of the *FMI 3.0* standard:

The only difference to *FMI 3.0* is, that eFMI adds the additional reserved value `"org.efmi-standard"` for the `type` attribute of `<Annotation>` elements (besides the values already reserved in *FMI 3.0*).

2.3.5. Checksum calculation

The calculation of checksums is based on SHA-1 according to the *Secure Hash Standard (SHS), FIPS PUB 180-4*. Checksums are computed based on binary content — i.e., content as is without any modifications like operating environment specific line-ending or encoding localizations.

The checksum is the mean to ensure integrity across different containers in an eFMU. These different container relate to each other and may be changed independent of each other. In order to ensure / check the integrity, with each change of a container, its checksum is updated in the reference entry in the `__content.xml` file.

For containers, that reference information from other containers or depend on them, also the checksum of these referenced containers is locally stored in that manifest. The comparison of these checksums is now an appropriate mean to check the consistency within the eFMU.

The calculation of checksums is done on the files that are listed in the manifest of the container (for which the `needsChecksum` attribute has the value "true") and the checksum is stored in the `checksum` attribute of the corresponding "File" list entry of the "Files" elememt of each manifest file.

The overall checksum of a model representation is the checksum of the manifest file, where all checksums of files of the model representation has been stored. Since the paths of the files are part of the manifest file itself it is ensured that a change of names, structure or content of the concerned files will result in a different checksum and allows for detecting changes, e.g. a model representation

has been changed in the container, but has been taken as input for transformation tools before.

On the other hand, changes to files with attribute `needsChecksum="false"` (e.g. description files) will not affect the checksum as well as adding of files not listed in the manifest (listing in the manifest would also alter the checksum).

2.3.6. FMU File References

An eFMU container must be downward-compatible to an FMU container. Hence, it may have an FMU which is stored in the root directory of the container (above the "eFMU" directory). Such FMU needs to be associated with a certain model representation located in the eFMU container. In general, each model representation may have an optional FMU, especially a Production Code model representation.

The currently activated FMU needs to be specified in the `__content.xml` file by using the optional attribute `activeFmu`. If it is set, its value must correspond to the name of the associated model representation. If no FMU is unpacked currently, the value of this attribute must not be set.

The optional FMU of a model representation is specified within the manifest file of the model representation, where one and only one file in the list of files has the role attribute set to FMU. Its value must be a relative path inside the model representation to the FMU file.
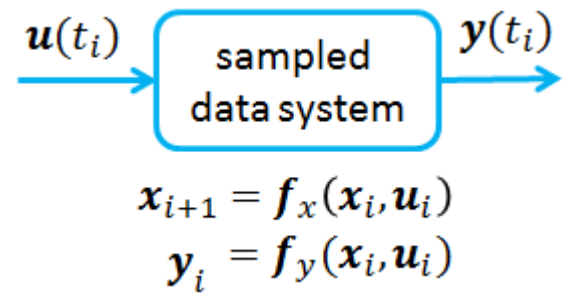
When the FMU of a model representation *M* is activated, the following steps are performed:

1. All files in the container's root except the "eFMU" directory are removed.
2. The FMU file referenced by *M* is unzipped to the container's root.
3. The value of the attribute `activeFmu` is set to the name of the model representation *M*.

Chapter 3. Algorithm Code Model Representation

$$u(t_i) \quad \boxed{\begin{array}{c} \text{sampled} \\ \text{data system} \end{array}} \quad y(t_i)$$

$$x_{i+1} = f_x(x_i, u_i)$$
$$y_i = f_y(x_i, u_i)$$

The Algorithm Code model is a portable and tool-independent *intermediate representation* for coupling physics-modeling tools with embedded Production Code generation. Mathematically, it is described as a sampled input/output block with one (potentially varying) sample period T_i for the whole block where inputs u_i and previous (block internal) states x_i are provided at sample time t_i and outputs y_i and new states x_{i+1} are computed and are latest used at sample time $t_{i+1} = t_i + T_i$ (see figure to the right). All variables of the block have a defined type and all statements of the block are sorted and explicitly solved for a particular variable. Functions are provided to execute the relevant parts of the block, especially to initialize it and to perform one step.

The purpose of the Algorithm Code model representation is to provide a well defined reusable basis for the Production Code generating tools. It can be seen as a target-independent Production Code on a *logical level* where the relationship to the original model is clearly visible (for example, the hierarchy of the original model is visible in the variable names). Depending on the embedded device the eFMU should be run on, a single Algorithm Code model representation can be used to generate multiple Production Code model representations and is therefore the last target independent model representation of the eFMU.

The Algorithm Code model representation consists

- of a *manifest file* in XML format in which all interface variables are defined (see Section 3.1),

- one *code file* with extension *.alg* that represents the executable part of the block consisting of a `block` with declarations, and mandatory definitions of the three methods `Startup`, `DoStep` and `Recalibrate`. These methods are defined in a target-independent way with the new language *GALEC* (Guarded Algorithmic Language for Embedded Control) which is based on the syntax of a *Modelica function* (https://www.modelica.org/modelicalanguage) with extensions as needed for embbeded systems (see section Section 3.2).

In the Algorithm Code specification and its examples the following coding conventions are used:

- *Types* — primitives and components — start with capital letters, and each successive word part starts capitalized. Examples: `Real, Boolean, Pid, GearBox, CrankShaftPid`.

- *Stateless functions* — including builtin functions — are defined with keyword `function`. The function names start with lower-case letters, and each successive word part starts capitalized. Examples: `sin, solveLinearEquations, computeCrankShaftPid`.

- *Stateful functions* are defined with keyword `method`. The method names start with capital letters,

and each successive word part starts capitalized. Examples: `Startup, Recalibrate, DoStep`.

- *Functions for scalars that are generalized to one and two dimensions* use the scalar function name with suffix `1D` and `2D` appended. Examples: `roundTowardsZero1D, interpolate2D`.

3.1. Algorithm Code manifest

$$u(t_i) \quad \boxed{\begin{array}{c} \text{sampled} \\ \text{data system} \end{array}} \quad y(t_i)$$

$$x_{i+1} = f_x(x_i, u_i)$$
$$y_i = f_y(x_i, u_i)$$
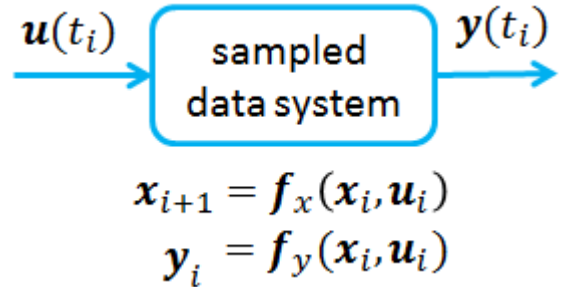
The *manifest file* of the Algorithm Code model representation is an instance of an XML schema definition and defines the variables and block methods that represent a sampled input/output block, see figure to the right.

3.1.1. Definition of an eFMU Algorithm Code (efmiAlgorithmCodeManifest.xsd)

This is the root-level schema file of the Algorithm Code model representation and contains the following definition:
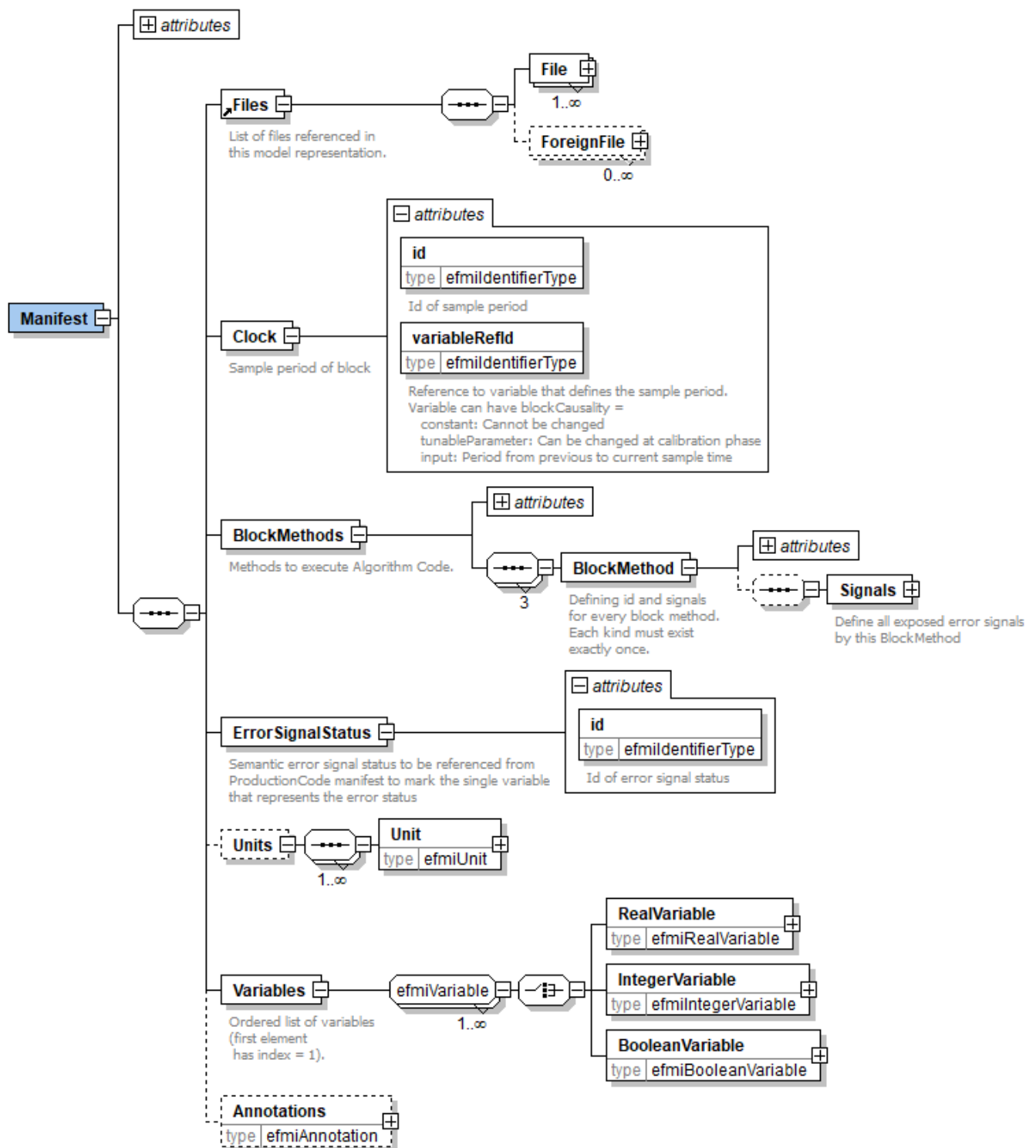
On the top level, the schema consists of the following elements (see figure above):

Element-Name	Description
attributes	The attributes of the top-level element are the same for all manifest kinds and are defined in section Section 2.3.1.
	Current kind-specific values: kind = "AlgorithmCode", xsdVersion (value is the current xsd version of the schema for the Algorithm Code model manifest).

Element-Name	Description
Files	List of files referenced in this model representation. There must be at least one file that contains the code of the BlockMethods. This element is the same for all manifest kinds and is defined in section Section 2.3.3.
Clock	A reference to the sample period defined by a block variable. For details see Section 3.1.2.
BlockMethods	The properties of the block methods DoStep, Recalibrate, and DoStep. For details see Section 3.1.3.
ErrorSignalStatus	Unique anchor that is used by derived model representations — like production code referring to this manifest — to mark error signal variables. For details see Section 3.1.4.
Units	An optional global list of unit definitions. These definitions are used in the XML element Variables. This element is nearly identical to the corresponding FMI 3.0 UnitDefinitions element. For details see Section 3.1.5.
Variables	A list of all variables that are accessible from the block methods defined in element BlockMethods. A variable might be a scalar or an array of an elementary type. Contrary to FMI 3.0, no target type variables (such a Float64) are defined here, but mathematical variable types (such as RealVariable). The reason is that target specific types are defined for the Production Codes *[otherwise it would not be possible to define, for example, Float32 and Float64 Production Codes in the same eFMU]*. For details see Section 3.1.6.
Annotations	Additional data that a vendor might want to store and that other vendors might ignore. For details see Section 2.3.4.5.

3.1.2. Definition of *Clock*

Element Clock provides a reference to the block variable defining the sample period for which the block was designed; only fixed sample periods are permitted, i.e., the referenced variable must be a constant. According to the GALEC block life-cycle (cf. Section 3.2.3, §3), system integration must execute the block periodically with the defined fixed sample period.

Element-Name	Description
`id`	The id of the sample period of the block.
`variableRefId`	Reference to the variable in `<Variables>` that defines the sample period. This variable must have value `constant` for its `blockCausality` attribute. If the variabe has a unit, it must be seconds; otherwise the implicit unit is seconds. The restrictions — constant in seconds — are required to guarantee a well-defined, unique interpretation of Behavioral Model representations for testing (cf. §1.3 and §4 of Section 4.2).

When the production code of this block is integrated in the target system (for example as AUTOSAR Adaptive Platform component), then it is expected that the block is executed as periodic sampled data system with the sample period defined by the variable `variableRefId` refers to. It might be that also a slightly changed sample period in the target system may still result in reasonable performance; in that case, system integration must adapt the sample period respectively in generated production code *and* take special care to adjust the sampling points of test tooling using Behavioral Model containers referring to this Algorithm Code manifest (cf. §1.3 and §4 of Section 4.2).

3.1.3. Definition of BlockMethods

Element `BlockMethods` defines properties of the defined block methods. Exactly three `BlockMethod` elements must be defined.
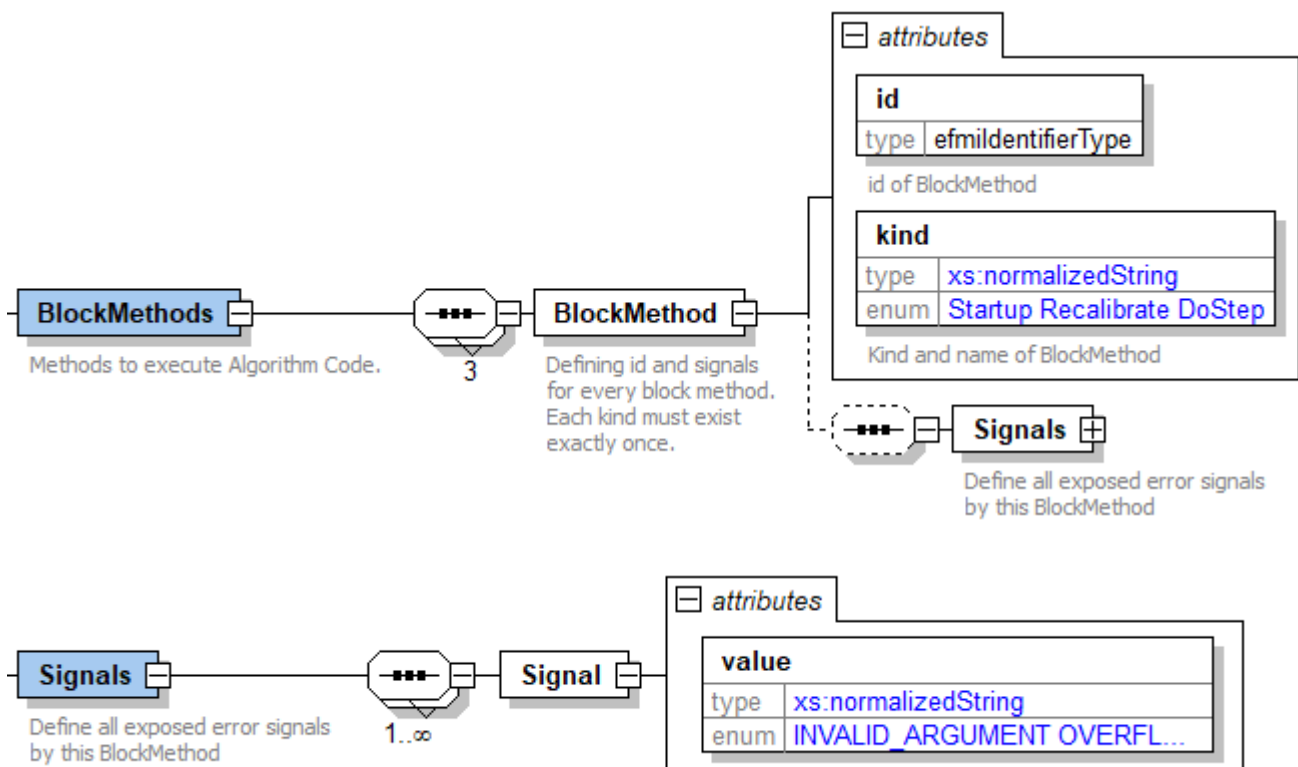
Name	Description
`fileRefId`	A reference to the file (defined in `<Files><File>`, cf. Section Section 2.3.3) in which the code of the block methods is stored.
`id`	The ID of the block method

Name	Description
`kind`	The kind of the block method (this is also the name of the method). Currently possible values are `Startup, DoStep, Recalibrate`.
`Signals`	The error signals exposed by the respective block method. The `value` attribute of each `<Signal>` defines one possible pre-defined error signal that can be signaled. Cf. Section 3.2.5, §1.1 and §1.6, for permitted values and their semantic.

3.1.4. Definition of *ErrorSignalStatus*

This element defines an unique anchor that is used by derived model representations — like production code referring to this manifest — to mark error signal variables. Such hold the error signal status of a block-interface method after its execution and therefore can be used to check if the method's execution succeeded or encountered any errors (cf. Section 3.2.5, §1.6 and Section 3.2.3, §3 regarding GALEC error signals and block-interface methods).

ErrorSignalStatus — Semantic error signal status to be referenced from ProductionCode manifest to mark the single variable that represents the error status

attributes — id — type efmiIdentifierType — Id of error signal status
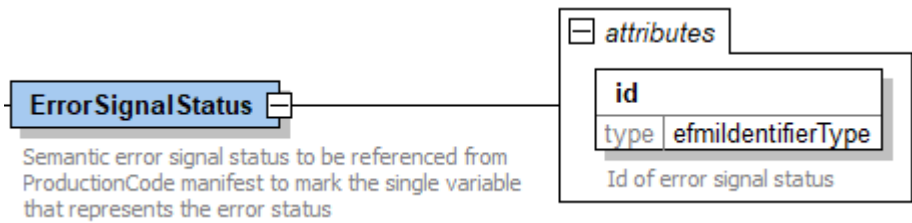
The `<ErrorSignalStatus>` consists only of attribute `id` that defines the unqiue ID of the anchor.

> ***E-1:*** Behavioral Model representations for example have to use the `<ErrorSignalStatus>` to conclude how to access the error signal status of the implementation of a block-interface method of tested production code. This is required, because testing error signals is mandatory for Behavioral Models (cf. §2.3 and §4 of Section 4.2).

3.1.5. Definition of Units

Element `Units` defines the units that are used by the `Variables` element.

This element is identical to element `UnitDefinitions` of FMI 3.0 with the only exception that there is an additional attribute `id` to identify a unit uniquely in the Algorithm Code manifest and that there is no support for display units (element `DisplayUnit`):

eFMI® Standard 1.0.0 Beta 1 (candidate-draft)| www.efmi-standard.org | CC BY-SA 4.0

efmiUnit

attributes

id
| type | efmiIdentifierType |

id of unit

name
| type | xs:normalizedString |

Name of Unit element, e.g. "N.m", "Nm", "N*m", "%/s".
"name" must be unique will respect to all other elements of the Units list.

BaseUnit ⊞

BaseUnit_value = factor*Unit_value + offset

Units

Unit
| type | efmiUnit |

1..∞

BaseUnit_value = factor*Unit_value + offset

attributes

kg
type	xs:int
default	0

Exponent of SI base unit "kg"

m
type	xs:int
default	0

Exponent of SI base unit "m"

s
type	xs:int
default	0

Exponent of SI base unit "s"

A
type	xs:int
default	0

Exponent of SI base unit "A"

K
type	xs:int
default	0

Exponent of SI base unit "K"

mol
type	xs:int
default	0

Exponent of SI base unit "mol"

cd
type	xs:int
default	0

Exponent of SI base unit "cd"

rad
type	xs:int
default	0

Exponent of SI derived unit "rad"

factor
type	xs:double
default	1

offset
type	xs:double
default	0

3.1.6. Definition of Variables

The `Variables` element consists of an ordered list of all variables used as *model states* of the methods defined in element `BlockMethods`, so the values of these variables can be directly accessed and changed in the respective method using the `name` of the variable prepended with the instance name `self` (for example `self.previous_x` if the variable has name `previous_x`). Variables that are defined with `blockCausality = input` are set from the environment at the beginning of a sampling period. Variables that are defined with `blockCausality = output` are used at the end of the sampling period by the environment in an appropriate way. Variables that are defined locally in a block method are not listed in the `Variables` element.

Variables are defined as (hereby one variable is defined according to schema group `efmiVariable` in file `efmiVariable.xsd`):
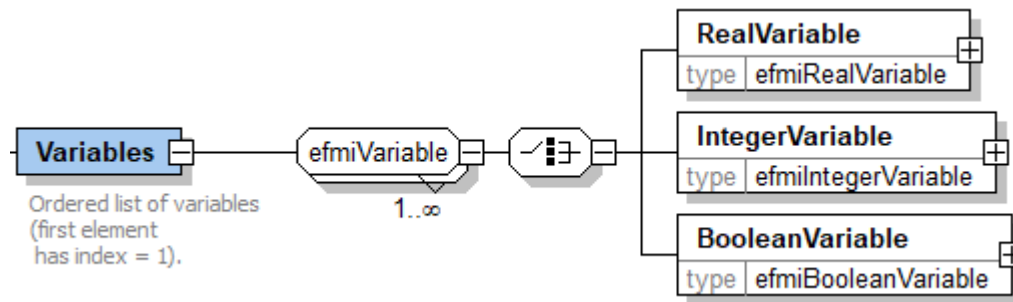
The schema definition contains basically the same information as element `ModelVariables` in FMI 3.0, but using mathematical instead of target types and having the following deviations:

- There is no `String` type.

- A type might have `Dimensions` where the size of a dimension is an Integer *literal* (a dimension cannot depend on a structural parameter as in FMI 3.0).

- The variable attributes `causality`, `variability` and `initial` of FMI 3.0 are replaced with the new attribute `blockCausality` (see below).

- The following FMI 3.0 attributes are **not** present:
    - `valueReference`
    - `canHandleMultipleSetPerTimeInstant`
    - `clockReference`
    - `clockElementIndex`
    - `intermediateUpdate`
    - `declaredType`
    - `quantity`
    - `displayUnit`
    - `unbounded`
    - `derivative`
    - `reinit`

Variable Base (attributes + elements)

All variable kinds (so `RealVariable, IntegerVariable, BooleanVariable`) have the following base attributes/elements:

Name	Description
id	The *unique* identification of the variable with respect to the AlgorithmCode manifest file (can be referenced from other manifest files).
name	The full, *unique name* of the variable. Every variable is uniquely identified within an eFMI AlgorithmCode instance by this name.
description	An optional description string describing the meaning of the variable.

Name	Description
blockCausality	Enumeration that defines the causality, variability and initialization of the variable. Allowed values of this enumeration: • `"input"`: The variable value is set by the environment at the start of a sampling period. • `"output"`: The variable value can be used by the environment once it is computed. • `"tunableParameter"`: Independent parameter that is constant during a call to `DoStep()` and can be calibrated. • `"calculatedParameter"`: A data value that is constant during a call to `DoStep()` and is computed during initialization or when tunable parameters change. • `"constant"`: The value of the variable defined with the `start` attribute never changes. • `"state"`: Local state variable that is initialized in `Startup` and is calculated from other variables. The value of this variable is kept between method calls.

Name	Description
start	Initial value of the variable as defined by default initialization.
	The given `xs:token` value can encode either a scalar value or a multi-dimensional value where each element value is separated by an XML whitespace character. In the latter case, the array elements are given in *row-major* order, that is the elements of the last index are given in sequence.
	[For example, a table T[4,3,2] (first dimension 4 entries, second dimension 3 entries, third dimension 2 entries) is mapped into the following sequence of values: *T[1,1,1],* *T[1,1,2],* *T[1,2,1],* *T[1,2,2],* *T[1,3,1],* *T[1,3,2],* *T[2,1,1],* *T[2,1,2],* *T[2,3,1],* *...]*
	If the variable is a scalar, the string must encode a scalar value. If the variable is a multi-dimensional array, the string can either: (1) encode a scalar value, meaning that each element of the multi-dimensional array has the respective scalar value as start value or (2) encode a multi-dimensional value, meaning that the start values of the elements of the multi-dimensional array are the respective encoded multi-dimensional value.
	Encoded values must be of the variable's type and each must satisfy its `min` and `max` value (if `min` and/or `max` elements are defined).
Dimensions	If the variable is an array, then the fixed dimensions of the array are defined by this element. For every dimension, the `number` defines the number of the dimension (must be consecutive numbers 1, 2, ...) and `size` defines the fixed size of the dimension (must be >= 1).
Annotations	Additional data of the variable, e.g., for the dialog menu or the graphical layout. For details see Section 2.3.4.5.

In FMI 3.0 the attributes `causality, variability, initial` are defined, which combinations are allowed and why the allowed combinations are needed for an offline simulation program with events. However, for eFMI most of the combinations cannot occur. For simplicity, eFMI uses therefore only the attribute `blockCausality`. In the following table the mapping of `blockCausality` to the FMI 3.0 attributes is defined:

eFMI	FMI 3.0		
blockCausality	**causality**	**variability**	**initial**

eFMI	FMI 3.0		
input	input	discrete	--- (no initial)
output	output	discrete	exact
tunableParameter	parameter	tunable	exact
dependentParameter	calculatedParameter	tunable	calculated
constant	local	constant	exact
state	local	discrete	exact

RealVariable-specific attributes

The following `RealVariable` specific attributes are defined:

```
unitRefId
type | efmiIdentifierType
```
Unit of variable (reference to Units list)

```
relativeQuantity
type    | xs:boolean
default | false
```
Defines if BaseUnit-based unit conversions have to consider the base-unit's offset (relativeQuantity=false) or not (relativeQuantity=true).

```
min
type | xs:normalizedString
```

```
max
type | xs:normalizedString
```
max >= min required

```
nominal
type | xs:normalizedString
```
nominal > 0.0 required

Attribute-Name	Description
unitRefId	Identifier of the unit of the variable defined in list `Units.Unit` (Section 3.1.5). The value of the variable is with respect to this unit.
relativeQuantity	Defines if BaseUnit-based unit conversions have to consider the base-unit's offset (relativeQuantity=false) or not (relativeQuantity=true). *[For example, 10 degree Celsius = 10 Kelvin if `relativeQuantity = "true"` and not 283.15 Kelvin.]*
min	Minimum value of variable (variable value ≥ `min`). If not defined, the minimum is the largest negative number that can be represented on the machine. If the variable is a multi-dimensional array, `min` is a scalar value that holds for all array elements.
max	Maximum value of variable (variable value ≤ `max`). If not defined, the maximum is the largest positive number that can be represented on the machine. If the variable is a multi-dimensional array, `max` is a scalar value that holds for all array elements.

Attribute-Name	Description
nominal	Nominal value of variable. If the variable is a multi-dimensional array, `nominal` is a scalar value that holds for all array elements. If not defined and no other information about the nominal value is available, then nominal = 1 is assumed. *[The nominal value of a variable can be, for example, used to define tolerances or scaling values for numerical algorithms in which the variable is used.]*

Example:

```xml
1  <Units>
2    <Unit id="UnitID_1" name="s"/>
3  </Units>
4
5  <Variables>
6    <RealVariable id="ID_1" name="Ti" unitRefId="UnitID_1" blockCausality="tunableParameter"
   start="0.1"/>
7    <RealVariable id="ID_A" name="A" blockCausality="constant" start="1.1 1.2 2.1 2.2">
8        <Dimensions>
9            <Dimension number="1", size="4"/>
10       </Dimensions>
11   </RealVariable>
12   <RealVariable id="ID_2" name="previous(I.x)"  blockCausality="state" start="0.0" min="0.0" />
13 </Variables>
```

IntegerVariable-specific attributes

The following `IntegerVariable` specific attributes are defined:


```
min
type | xs:int

max
type | xs:int

max >= min required
```
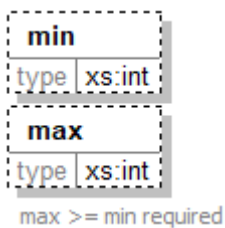
Attribute-Name	Description
min	Minimum value of variable (variable value ≥ `min`). If not defined, the minimum is the largest negative number that can be represented on the machine. If the variable is a multi-dimensional array, `min` is a scalar value that holds for all array elements.
max	Maximum value of variable (variable value ≤ `max`). If not defined, the maximum is the largest positive number that can be represented on the machine. If the variable is a multi-dimensional array, `max` is a scalar value that holds for all array elements.

Examples:

```
1  <Variables>
2    <IntegerVariable id="ID_11" name="numberOfCylinders" blockCausality="tunableParameter"
   start="6" min="0" />
3    <IntegerVariable id="ID_12" name="pivots" start="0">
4        <Dimensions>
5            <Dimension number="1" size="8"/>
6        </Dimensions>
7    </IntegerVariable>
8  </Variables>
```

BooleanVariable-specific attributes

The `BooleanVariable` element has no additional attributes.

3.2. Guarded Language for Embedded Control (GALEC)

$$x_{i+1} = f_x(x_i, u_i)$$
$$y_i = f_y(x_i, u_i)$$
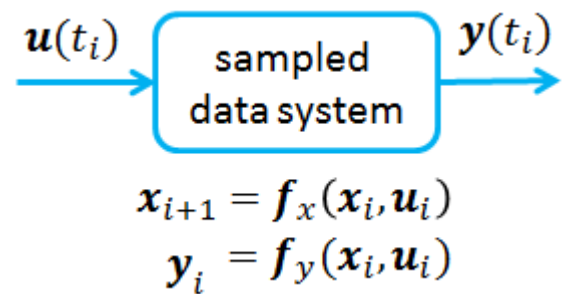
The algorithm that defines an input/output, sampled data block is defined with the new language **GALEC** (Guarded Algorithmic Language for Embedded Control) that is specified in this sub-section. GALEC is based on a small subset of the *Modelica Language* (especially on *Modelica functions*, *Modelica External Function Interface*, and on *Synchronous Language Elements*) of the Modelica Specification 3.4 (https://www.modelica.org/documents/ModelicaSpec34.pdf) together with changes and extensions as needed for **embbeded real-time systems**. GALEC has the following features that are **not** present in the Modelica Language:

- The language is designed so that only algorithms can be defined that have an upper-bound on the number of operations for each *control-cycle* to satisfy hard *real-time constraints* (for example, there are no *while* loops). Furthermore, all needed memory, especially of arrays and operations on arrays, is known statically.

- The language is designed for *computational safety*. For example it can be statically guaranteed that *out-of-bounds and otherwise illegal memory accesses for all possible executions* cannot occur at run-time.

- The language is designed for *traceability* so that GALEC code can be understood in terms of the original model and vice versa.

- The language has a restricted set of *methods* to efficiently pass the block state between functions.

- A set of *built-in functions* is defined so that physical models and their solvers can be reasonably mapped to GALEC code. For example, there are built-in functions for *interpolation* and for the solution of *linear equation systems*.

- The language is designed to handle *erroneous situations in a safe way*. For example, it is possible to determine at the end of the algorithm whether the computed outputs can be used for further processing, or whether it is necessary to switch to a backup code, for example, if operations produced *qNaN* (quiet-Not-a-Number) values. Furthermore, min/max values defined in the declaration of variables are used to implicitly limit the variable values at the start and at the end of the `DoStep` method. This is different to the Modelica language that raises assertions if min/max definitions are violated.

The GALEC code of a block is stored in a file with extension `*.alg` and is a self-contained file that can be parsed and interpreted without inspecting the Algorithm Code manifest file. For examples of GALEC programs, see Section 3.2.7.

3.2.1. Language-design Overview

GALEC code generation is subject to many, often contradicting, requirements imposed by physics and mathematics (physics-modeling domain), embedded real-time system-control (Production Code domain) and development processes for certified systems (embedded development domain):

(a) An algorithmic source-language for embedded real-time

GALEC code has to take into account that further embedded code generation typically must satisfy hard *real-time constraints*. Generated algorithmic solutions must have an *upper-bound of algorithmic steps* executed each *control-cycle*, such that termination within a statically fixed number of computational steps can be guaranteed. To derive such upper-bounds for actual GALEC code is subject of the *termination-analysis*, which checks that functions of GALEC code are transitively non-recursive and loops always have a statically fixed maximal number of iterations. To transform equation-based models to such solutions may not always be possible. To that end, GALEC code generators are free to reject valid models of their modeling-language as not being suitable for GALEC code generation.

Another important concern of embedded applications is *computational safety*, requiring for example that *programs are free of out-of-bounds or otherwise illegal memory accesses for all possible executions*; and that control-flows for error detection and handling always shortcut normal program execution [1]. To that end, a *dimensionality-analysis* is enforced, which statically defines the sizes of multi-dimensions w.r.t. function call contexts; considering all possible call contexts is required to support generic functions working on arbitrary sized multi-dimensions. The dimensions derived are used to statically ensure that all multi-dimensional accesses always will be within bounds throughout later program executions. Dimensionality and termination-analysis are closely linked; bounded loops can conveniently iterate multi-dimensions whose statically known dimensions in turn define respective upper iteration bounds. Since iteration bounds can depend on the sizes of any multi-dimension, other iteration indices or integer expressions combining such, GALEC code supports advanced iteration schemes that are still guaranteed to be well-defined.

(b) An algorithmic target-language for simulation of physics-models

GALEC code generators have to rearrange original physics-model equations to derive an algorithmic solution. The more comprehensive, complex and mathematically challenging a controller design is — and therefore interesting for modeling its physics — the more rigorous such transformations are typically. Particularly later real-time constraints as described in (a) often

require radical transformations to handle algebraic loops and enable equation-system optimisations like symbolic processing, tearing and index reduction. GALEC code generators are therefore encouraged to apply whichever *mathematical and logical equation-system transformations* they consider required to yield an equivalent algorithmic solution.

Besides the requirement to achieve an algorithmic solution in terms of expression- and assignment-sequences that compute the next state of the simulated control-cycle, no further transformation has to be performed. GALEC provides means to *compute with structured-data as common in physics-modeling languages*, particularly *higher-level matrix-operations*. And a library of *builtin functions* supports common mathematical tasks like solving a linear system of equations. The exact implementation of all these mathematical-abstractions is the responsibility of Production Code generators, leaving opportunity for later target-machine specific optimization. To that end, GALEC code generators are highly encouraged to leverage on the provided mathematical-abstractions.

(c) An intermediate-language leaning towards algorithm-logics and mathematical-optimization, not algorithm-implementation and target-specific optimization

The emphasis in (b) has been on mathematical transformations only; otherwise GALEC code generators should *not apply transformations that curtail Production Code generators* in their code generation decisions, particularly regarding optimisations leveraging on target-specifics. Typical target-specific optimisations are for example data-structure changes to improve memory-layout for faster access-operations or optimisations of the trade-off between code-size and performance like loop-unrolling. Especially higher-level matrix-operations and builtin function calls are interesting for target-specific Production Code optimisations. Although it seems obvious *not to further reduce such mathematical abstractions*, it is non-trivial in practice.

The mathematical equation-system transformations described in (b) typically imply separation or reduction of existing and introduction of new multi-dimensional data-structures, influencing matrix-operation and builtin function calls in turn. For example, tearing may be used to reduce the required numerical integration, in turn yielding smaller but also more frequent matrix allocations for linear solving. Fortunately, such mathematical transformations most often also result in more efficient embedded code generated by Production Code generators; but that is hard to say in general. Of course, if required to achieve an algorithmic solution at all, such transformations have to be done. But otherwise, the resulting *decomposition of matrices accompanied by matrix-operation flattening* and therefore increase in code size may very well supersede the advantage.

On the other hand, GALEC code generators have the domain-knowledge for mathematical-optimisations that Production Code generators lack. An important case for trade-offs between mathematical and Production Code optimisations is scalarization to *eliminate controller-output irrelevant or redundant state-variables and equations*. Physics-models often contain simple equality-equations between the state-variables of two components; likewise, the components constituting a certain controller may be generalized for more advanced cases than their actual application context, leaving equation-parts unused. GALEC code generators are encouraged to eliminate such system parts, which typically results in multi-dimensions with unused elements like a 2x3 matrix of which only four entries are actually required to compute the outputs. Eliminating the unused entries means to change model structure, while shifting the matrix or changing its dimensionality is not an option because of traceability and a lack of knowledge regarding the final

matrix-layout Production Code will eventually apply.

As an alternative, GALEC code can *scalarize* such multi-dimensions, i.e., flatten the higher-level multi-dimensional entity to a set of scalars — and therefore dimension-less — otherwise equally typed entities. Unused scalars can then just be discarded. The drawback of scalarization is, that all expressions containing higher-level matrix-operations with scalarized multi-dimensions and loops referring to such must be expanded to respective sequences of scalar operations. Besides being in conflict with the requirement to not curtail Production Code from optimizing higher-level matrix-operations, the resulting code-size increase due to expansions may very well render the savings in elements futile.

(d) A language for algorithmic controller implementation

TODO: `Startup` and `DoStep` (with input parameters); eFMU state and method vs. function; `previous` and `derivative` state-variables.

(e) A language part of a *trustworthy tool-chain* from physics-models to embedded-code

GALEC code generators have to maintain *traceability*, such that embedded solutions derived from physics-based controller designs can be understood in terms of the original model; and vice versa, all parts of a controller-model can be traced to its embedded implementation. To link individual physics-equations to their respective algorithmic solution is very challenging in general, since equations are likely subject to rigorous transformations as described in (b). A common denominator between a physics-model and its transformed solution is however, that both simulate the same system. It therefore is a starting point for GALEC code to at least *refer to the states of the original physics-model components* whenever using or updating such. The premise is of course, that controllers are modeled as systems consisting of well-structured parts; only then GALEC code generators can, and are highly encouraged, to utilize original system-structure for traceability. To that end, GALEC does not only provide mathematical multi-dimensions as described in (b), but also *nested multi-dimensional components with matrix- and scalar-variables*; and in case of optimisations resulting in scalarization as described in (c), a *quotation-based notation can be used to denote scalarized elements as if their original multi-dimensions still exist*. GALEC code generators have to maintain *traceability*, such that embedded solutions derived from physics-based controller designs can be understood in terms of the original model; and vice versa, all parts of a controller-model can be traced to its embedded implementation. To link individual physics-equations to their respective algorithmic solution is very challenging in general, since equations are likely subject to rigorous transformations as described in (b). A common denominator between a physics-model and its transformed solution is however, that both simulate the same system. It therefore is a starting point for GALEC code to at least *refer to the states of the original physics-model components* whenever using or updating such. The premise is of course, that controllers are modeled as systems consisting of well-structured parts; only then GALEC code generators can, and are highly encouraged, to utilize original system-structure for traceability. To that end, GALEC does not only provide mathematical multi-dimensions as described in (b), but also *nested multi-dimensional components with matrix- and scalar-variables*; and in case of optimisations resulting in scalarization as described in (c), a *quotation-based notation can be used to denote scalarized elements as if their original multi-dimensions still exist*. For example, a scalarized real variable may have the name `'a.b[2].c[2,3]'`, linking it with original model structure for traceability although all output-relevant combinations of components `a` and `b` and matrix `c` are scalarized into individual variables.

(f) A portable and tool-independent language for standardized tool-integration and distribution of controller implementations

GALEC code is at the center of eFMUs, *linking physics-modeling with embedded-development tooling*. Although eFMUs are free to only contain target-specific source code, build scripts and resulting binaries, such eFMUs are just fancy containers for embedded solutions; and vice versa, a pure modeling eFMU without executable embedded-solutions misses the actual purpose of **e**FMI compared to the ordinary FMI standard. It is the GALEC code that brings both worlds together and exposes their relation to eFMU users. The latter does not only imply traceability as described in (e), but also to adhere to a *common specification of controller inputs, outputs, states and parameters and control-cycle functionality* — an abstract *controller usage interface*. In the spirit of the FMI standard, and to not preclude a potential future integration with it, this interface is given in terms of an FMI like *XML manifest declaring all entities and functionalities of interest for users of the eFMU*. The control-state defined in GALEC code — the state components with state variables, control-inputs and -outputs and their nesting — therefore always is linked to entities declared in the manifest; likewise, the initialization and control-cycle functions are exposed in the manifest to clearly declare the functionality an eFMU provides. GALEC code generators are required to derive respective manifests if asked for.

3.2.2. Notation Conventions

The concrete syntax of GALEC code is defined using *Extended Backus–Naur Form* (EBNF) according to ISO/IEC 14977 [https://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip]. The whole grammar is split into different sections, each defining a specific language construct — i.e., *syntactic concept* — of GALEC code like lexemes, references, expressions, statements etc. The EBNF-rules — i.e., *syntactic rules* — defining the syntactic concept a section is about can be amended with further *semantic rules* given in prose. Semantic rules constrain the applicability of the syntactic rules they refer to. They are in turn classified w.r.t. the different *semantic concepts* of GALEC code they contribute to like type-analysis, dimensionality-analysis, termination-analysis etc.

Due to the decision to structure the whole specification w.r.t. language constructs, semantic concepts cross-cut sections. Table TODO summarizes all semantic concepts, the semantic rules contributing to their definition and the section they are defined. The inevitable complexity of cross-dependencies, typical for any serious formal language, is further attenuated by using a consistent notation for semantic rules, enabling explicit linkage between defined rules, the semantic concepts they contribute to and further rules relevant for or later refining a definition. Likewise, syntactic rules are well-prepared for usage in semantic-rules, i.e., usage in prescriptive definitions given in prose.

Syntactic Rules, Terms and Relations

Each syntactic rule has a unique rule-number of the form **G-X_1.X_2**, where X_1 is the section the rule is part of and X_2 is its unique rule-number within that section; the actual EBNF rule follows separated by a colon. The *non-terminals defined by syntactic rules are human readable terms* that are well-suited for prose-text usage. Semantic rules denote such usage by writing the respective non-terminal in *italic*. For readability reasons, every non-terminal can be used in plural or singular form and its first letter can be capitalized when used at the beginning of a sentence. The meaning of a non-terminal within a semantic rule is defined by the following meta-rule:

M-1.1 (syntactic term / Meta-rules, terminology): Parts of semantic rules typeset in *italic* refer to non-terminals; they are called *syntactic term*. Let *N* be a non-terminal referred to in a semantic rule *S*; let *G* be the syntactic rule defining *N* (cf. **M-1.2** for uniqueness of syntactic rules). The semantic of *N* in *S* is: a code fragment *F* of a whole GALEC program *P*, where *F* is derived according to *G* throughout the derivation of *P* and satisfies all semantic rules amended to *G*.

M-1.1 requires that the syntactic rule a syntactic term refers to is unique; to that end we define:

M-1.2 (uniqueness of syntactic rules / Meta-rules): For every non-terminal *N* exists a single syntactic rule whose EBNF syntax-rule has *N* as meta-identifier (cf. ISO/IEC 14977).

M-1.1 has severe consequences. If, for example, the specification refers to *loop-iterator-declarations*, it is clear that this must be *names* declared by a *for-loop* regardless in which context the syntactic term *loop-iterator-declaration* is used; this implication is given because *loop-iterator-declaration* just derives to *name* and is only used by *bounded-iteration*[2] which in turn is only used by *for-loop*. Besides such implicit restrictions, further explicit restrictions about the *syntactic relation* between syntactic terms — i.e., that some term's own derivation must be in a well-defined relation to another term's derivation throughout the whole derivation — are used:

M-1.3 (syntactic relations / Meta-rules, terminology): Let N_1 and N_2 be syntactic terms.

N_1 is *contained* in N_2, if, and only if, N_1 is derived throughout the derivation of N_2; in this case N_2 is called a *container* of N_1 and we say N_2 *contains* N_1 and N_1 *is part of* N_2. If, and only if, N_2 contains N_1 and both refer to the same non-terminal *N*, N_1 is called a nested *N*. N_2 is the *closest container* of N_1, if, and only if, N_2 contains N_1 and for all N_3 containing N_1 and that refer to the same non-terminal as N_2 it holds that N_3 contains N_2.

N_1 is *preceding* N_2, if, and only if, neither is contained in the other and the left-most derivation of the closest container of N_1 and N_2 derives N_1 before N_2; in this case N_2 *follows* N_1. Instead of preceding also the term *before* is used; and instead of follows also the term *after*. If, and only if, either, N_1 follows N_2 or N_2 follows N_1, both are siblings. N_1 and N_2 are *different*, if, and only if, they are siblings or the one contains the other.

N_1 is *lexically-equivalent* to a sequence of characters *α*, written $N_1 =_{lexical} α$, if, and only if, N_1 derives to *α*. N_1 is *lexically-equivalent* to N_2, written $N_1 =_{lexical} N_2$, if, and only if, N_1 and N_2 derive to the same sequence of characters.

If, and only if, N_2 contains N_1 and throughout all possible derivations of the non-terminal N_2 refers to the non-terminal N_1 refers to can be derived at most once, we speak of the N_1 *of* N_2; obviously, N_2 is the closest container of N_1 in that case.

Let $d = β_1, ..., β_n$ be a single definition according to ISO/IEC 14977; $β_i$ with $1 ≤ i ≤ n$ is called the *i'th factor* of *d*. A $δ_z$ is called the $γ_2$-...-$γ_z$*'th factor* of $δ_1$, if, and only if, $∀i,j∈ℕ+; i = j - 1; 2 ≤ j ≤ z: δ_j$ is the $γ_j$*'th factor* of $δ_i$. Let *G* be the syntactic rule of N_2. We call N_1 the i_1-...-i_k*'th child* of N_2, if, and only if, N_1 has been derived for the i_1-...-i_k'th factor of *G* when deriving N_2; in this case N_2 is

called the parent of N_1. If, and only if, the i_1-...-i_k'th factor of G has been derived when deriving N_2, we say N_2 *has a i_1-...-i_k'th child*; otherwise it is *without i_1-...-i_k'th child*.

A syntactic term F is *without* a code fragment according to some non-terminal N, if, and only if, N is not derived throughout the derivation of F; in this case, we say F does *not contain* a N, it is *N-free*. Note, that F is a syntactic term — i.e., a code fragment derived according to the syntactic rule for the non-terminal F — whereas N is just a non-terminal referring to some syntactic rule; nevertheless, N will be highlighted *italic* in semantic rules as if it is a syntactic term, denoting that it is a non-existing code fragment.

E-1: The derivation of the following *block* fragment defines various syntactic relations (denoted by using capitals only). Note, that according to **M-1.1** syntactic relations are only defined for syntactically correct inputs, i.e., *blocks* (cf. **S-2.1**).

```
1  /*
2      For-loop CONTAINING another for-loop.
3      Thus, neither for-loop is BEFORE or AFTER the other.
4      Both for-loops are function-call-FREE:
5  */
6  for i in 1:size(A,1) loop
7      /*
8          If-statement PART OF a for-loop and CONTAINING a
9          DIFFERENT for-loop. The if-statement is WITHOUT a
10         function-call since it does NOT CONTAIN such:
11     */
12     if
13         /*
14             The 2ND CHILD of the if-statement is an expression:
15         */
16         mod(i,2) == 0
17     then
18         /*
19             NESTED for-loop, i.e., a for-loop CONTAINED in
20             another for-loop. The NESTED for-loop FOLLOWS its
21             CONTAINING if-statement's 2ND CHILD:
22         */
23         for j in 1:size(A,2) loop
24             /*
25                 Assignment α PRECEDING another assignment β, with
26                 which its 1ST CHILD is LEXICALLY-EQUIVALENT.
27                 The assignment is also BEFORE another assignment γ
28                 that is DIFFERENT to β; all three assignments are
29                 SIBLINGS:
30             */
31             A[i,j] := 1; /* α */
32         end for;
33     else
34         /*
35             Assignment β AFTER a PRECEDING assignment α with
36             LEXICALLY-EQUIVALENT 1ST CHILD:
37         */
38         A[i,j] := 0; /* β */
39     end if;
40 end for;
```

```
41  /*
42      Assignment y most likely not PART OF a for-loop,
43      but for sure with exactly one function-declaration CONTAINER
44      that trivially is its CLOSEST function-declaration CONTAINER:
45  */
46  A[size(A,1), size(A,2)] := -1; /* y */
```

E-2: Consider the syntactic rule **G-2.3**:

```
1  function-declaration =
2      ( "function" | "method" ),
3      name,
4      { parameter-declaration },
5      [ "protected", { local-variable-declaration } ],
6      "algorithm",
7      { statement },
8      "end",
9      name,
10     ";" ;
```

Its first factor is ("function" | "method"), its 1-2′th factor is "method", its 4′th factor is ["protected", { local-variable-declaration }], its 4-2′th factor is { local-variable-declaration } and its 4-2-1′th factor is local-variable-declaration. According to the presented syntactic rule, every *function-declaration* must have a 5′th child lexically-equivalent to "algorithm" even if it contains no *statements*; it can also be without 4-2′th child although it has a 4′th and 4-1′th child. It is important to note here, that if without 4-2′th child, a *function-declaration* cannot contain *local-variable-declarations*; the reason is because the 4-2′th factor is the only possibility to derive *local-variable-declaration* throughout any possible derivation of *function-declaration*. Likewise the 6′th factor is the only possibility to derive *statements* throughout the derivation of *function-declarations*. Finally, note the difference between without an *i*′th child vs. without a code fragment according to some non-terminal. *Local-variable-declaration* and *parameter-declaration* will always derive *variable-declaration* throughout their own derivation. Thus, *function-declarations* for example can be without 4-2′th child and still contain a *variable-declaration* if they have a 3′rd child, i.e., a *function-declaration* can be without 4-2′th child but still not *variable-declaration*-free.

Consider the following *function-declaration*:

```
1  function foo
2  protected
3  algorithm
4  end foo;
```

Its second and eight children are *names* lexically-equivalent to foo. It is without 1-2′th child because it has a 1-1′th child lexically-equivalent to "function". And although it has a 4′th child, it is without a *local-variable-declaration*.

Using syntactic relations, complicated constraints can be conveniently and precisely defined. For example, the usage of *references* in statically-evaluated expressions is restricted; on the one hand, they never must be used to access control-state-dependent — i.e., runtime — values, but on the other hand, they should be available to access runtime-independent values provided by the dimensionality- and termination-analysis like the dimensional-sizes of variables or the iteration-values of loop-iterator variables which are always statically-bound. A respective formal definition, based on syntactic relations only, is: every *reference* contained in a *constant-scalar-integer-expression* must either, be the 3'rd child of a *dimension-query* or have a unique *for-loop* container whose *loop-iterator-declaration* is lexically-equivalent to the *reference*. Although such constraints sound like common prose, they are completely formally well-defined by meta-rules **M-1.1** to **M-1.3** and the derivation semantics of EBNF as defined in Section 5 of ISO/IEC 14977.

It is important to note, that meta-rules, like **M-1.1** to **M-1.3**, are used by nearly all semantic rules and therefore not explicitly referenced by definitions even if relevant.

Semantic Rules

Likewise syntactic rules, also semantic rules have unique rule-numbers. The structure for semantic rule-numbers is **S-X_1.X_2**; again X_1 is the section the rule is part of and X_2 a unique rule-number within that section. The unique rule-number is followed by an informal rule name describing the rule-intention, a slash and finally one or more semantic concepts the rule contributes to, all wrapped in parenthesis. The actual definition follows separated by colon.

As an example consider the following semantic rule:

> **S-TODO (guarded multi-dimension access / Dimensionality-analysis):** For each dimensional-context of the *function-declaration* a *reference* R is part of (cf. **S-TODO**), the dimensional-bounds of the *computed-dimensions* of R must be within the dimensional-bounds of the declaration R refers to (cf. **S-TODO**).

The general definition of dimensional-bounds and what it means for one to be within another is given by meta-rule **M-TODO** to which — like for all common meta-rules — is not explicitly referred to.

Rationales, Limitations and Examples

Besides syntactic and semantic rules, sections also list rationales, limitations and examples. A rationale gives further reason why something is specified as it is, like usage-considerations, other specifications of interest or easy overlooked cases that are non-trivial to handle. A limitation clarifies a language constraint that might be relaxed in further iterations of the standard to support future use-cases, that is required to support further tooling working with GALEC code or that is very hard to ease in general for which reason it has been introduced. Examples are used to investigate the implications of the specification by demonstrating code fragments that are illegal GALEC code or that are valid but with a twist fostering understanding of the specification. All three — rationales, limitations and examples — can be part of semantic rules, in which case they are uniquely numbered within the rule they are part of. If more general, they can also be freestanding, in which case their unique number is constructed likewise syntactic and semantic rule numbers, only that rationales are prefixed by **R-**, limitations by **L-** and examples by **E-**. In any case, rationales and limitations have an

informal name describing their intention likewise semantic rules have. If freestanding, they also can be associated with semantic concepts, again separated by a slash like for semantic rules; if not freestanding and part of a semantic rule, they implicitly contribute to the same semantic concepts as the rule they are part of.

As an example consider the following non-freestanding rationales, example and limitation:

S-TODO (uniqueness of early loop exits / Termination-analysis): Let B_1 and B_2 be two different *early-loop-exits*. Their respective closest *for-loop* containers must be different; and their *loop-iterator-references* must refer to different *for-loops*.

R-1 (well-formedness): That *early-loop-exits* must be part of a *for-loop*, and the name-analysis of their *loop-iterator-references*, are already defined by **S-TODO**.

R-2 (MISRA C:2012 compliance): The rule is introduced to enforce compliance with MISRA C:2012, Rule 15.4.

E-1: The following *for-loop* is illegal due to multiple early loop exits for each of the nested loops:

```
 1 for i in 1:3 loop /* Outer loop. */
 2     for j in 1:3 loop /* Inner loop. */
 3         if b1 then
 4             break i; /* First break of outer and inner loop. */
 5         else
 6             break j; /* Illegal: Second break of inner loop. */
 7         end if;
 8     end for;
 9     if b3 then
10         break i; /* Illegal: Second break of outer loop. */
11     end if;
12 end for;
```

L-1 (relaxation of MISRA C:2012 compliance): To transform non-unique early loop exists to a unique form complying with MISRA C:2012 is not trivial. Production code generators may miss support for such transformations, to which end this rule has been introduced. On the other hand, it may unnecessarily constrain GALEC code generators, even forcing them to fail to generate an algorithmic solution. To shift the responsibility of compliance from GALEC code generators to Production Code generators, the rule can be disabled using the `consider-misra=false` flag throughout GALEC code generation.

Other specification parts can refer to enclosed rationales, limitations and examples by appending

their unique number separated by a colon to the number of the enclosing semantic rule; for example, one can refer to the limitation of above example by writing **S-TODO:L-1**.

3.2.3. Block-interface and life-cycle

This Section investigates the utilization of GALEC programs (i.e., *blocks*) that are due for deployment on an embedded target and its runtime environment.

§1: Embedded target, runtime environment, system integration, block instance & block-interface (terminology, system integration)

GALEC defines an operational interface for blocks — called block-interface — that must be preserved by Production Code generators when translating a block to code that is subject of embedded system integration. Embedded system integration is not just achieved by means of a block's interface; it must over and above adhere to the operational restrictions defined in *§1* to *3* (particularly the block life-cycle of *§3* must be satisfied).

A single block can be instanziated many times on an embedded target and its runtime environment; each instance is operationally isolated. There are no restrictions on the number or kind of block instances (in particular different blocks can be instanciated within the same runtime environment). Any interaction of the runtime environment with a block instance must be via its block-interface (even instances of the same block must interact via their block-interface).

§2: Block-interface variables & methods (runtime semantic, system integration)

The block-interface constitues of block-interface variables and block-interface methods.

The block-interface variables are:

- **Block inputs:** The sampling inputs provided by the runtime environment.
- **Block outputs:** The sampling results consumed by the runtime environment; they must never be written by the runtime environment.
- **Tunable parameters:** Parameters sporadically, and not necessarily each sampling, changed by the runtime environment.

Besides this block-interface variables, other block-variables exist, which are block internal and therefore cannot (and must not, cf. *§1*) be written or read by the runtime environment:

- **Dependent parameters:** The parameters derived from tunable parameters.
- **Block states:** The internal states.

All block-variables are persistently stored in block instances, such that their values survive block-interface method calls and therefore can be used in call sequences of such. Each block instance has its individual set of block-variables; changing some tunable parameter `t` of a block instance `b1` does not change `t` of another block instance `b2` of the same block.

The block-interface methods are:

- `Startup()`: Computes initial values for all block-variables.

- `Recalibrate()`: Updates the dependent parameters considering the currently set tunable parameters.
- `DoStep()`: Computes the block outputs and updates the block states for the given block inputs and the current tunable and dependent parameters for a single sampling.

L-1 (design-space of Production Code generation and system integration): Production Code generators and system integration are free to realize a GALEC block by any means they see fit as long as its operational semantic is satisfied. They can achieve a mutual agreement that block-interface functionality is not supported, like recalibration by means of `Recalibrate()` or reading block inputs from the runtime environment, given that the use-case and system integration scenario does not require such. In general however, Production Code generators must support the full block-interface and life-cycle to be eFMI specification conformant.

Examples of integration scenario specific design-space agreements are:

1. Not generate and call `Startup()`, but instead statically evaluate it and store start values in read only memory or only load them once when the runtime system boots.

2. Not generate a dedicated `DoStep()` function, but instead inline the implementation in the runtime environment.

3. Not generate `Recalibrate()`, transforming tunable and dependent parameters to become constants which can be constant-folded.

4. Store block-variables globally, leveraging on knowing that there is exactly one instance and not several (no need to support individual block-instances).

5. Not persist block inputs (cf. ***§3:R-1***, last paragraph), but instead provide new values for every input every sampling, for example as function arguments to `DoStep()`.

Particularly (3) is a common integration scenario, since recalibration typically is only performed during the development phase of an embedded system and no longer supported in production systems.

R-1 (block-variable initialization and Algorithm Code Container manifest start values): The start values of the variables in the manifest of an Algorithm Code Container are conceptually determined by calling `Startup()` on the target system and its runtime environment. A Production Code generator can for example (1) use these start values directly in the C-Code for static initialization (i.e., as precomputed values), hereby casting from the concrete manifest-variable type in which the `start`-values are stored to the best fitting concrete type of the target system, or (2) provide an implementation of `Startup()` to be called by the runtime environment during startup, or (3) use any other means to ensure block-variables have initial values according to `Startup()` (cf. ***§2:R-1***).

---

§3: Block life-cycle (runtime semantic, system integration)

The permitted interactions with block instances are defined by the following state machine,

eFMI® Standard 1.0.0 Beta 1 (candidate-draft)| www.efmi-standard.org | CC BY-SA 4.0

specifying a universal life-cycle for block instances, called block life-cycle (the do-actions of states refer to the block-interface methods defined in *§2*):
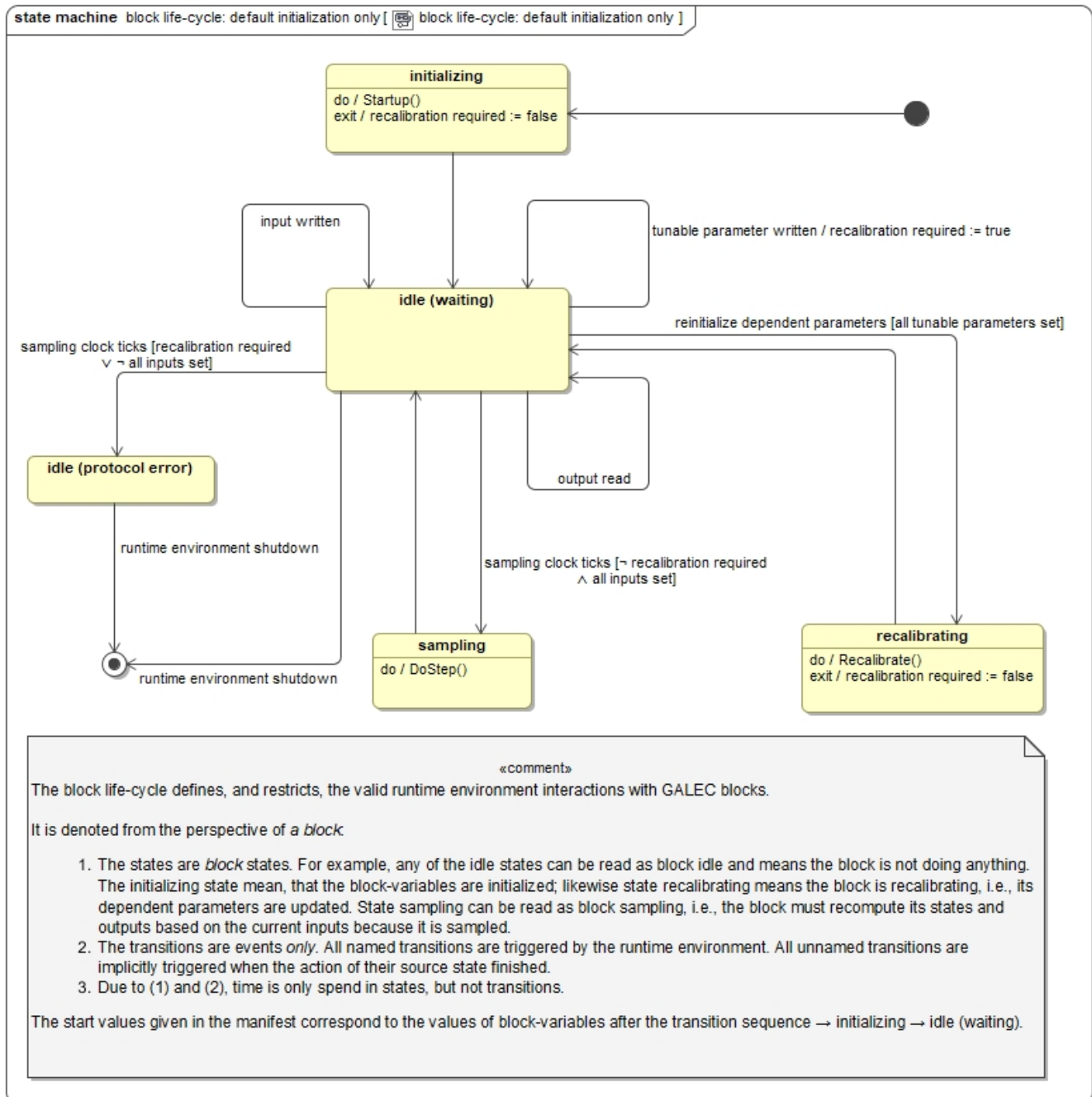
The block-interface methods of a single block instance must be called in sequence by the runtime environment; parallel execution of such is prohibited. The block-interface methods of separate block instances can be executed in parallel. The block-interface variables of a block instance must not be read or written by the runtime environment while any of its block-interface methods is in execution.

> **R-1 (block life-cycle implications for system integration):** The following discussion refers to the block life-cycle state machine. *Italic* refers to states or transitions of it; `monospace` refers to state actions, i.e., block-interface methods according to *§2*.
>
> The block life-cycle does not enforce the runtime environment to set inputs and tunable parameters (*input written* and *tuneable parameter written* transitions) separately in sequence or at most once before each *sampling*. It does not prohibit the runtime environment to read

block inputs or tunable parameters or execute `Recalibrate()` several times before a single *sampling*. This allows complex system integration scenarios where the runtime environment has to setup the next sampling depending on the state of a block instance.

The block life-cycle enforces however, that whenever a tunable parameter is changed via *tunable parameter written*, all dependent parameters must be recomputed via `Recalibrate()` before the next sampling (*recalibration required* conditional). Otherwise, a protocol error is given and the block behavior is undefined (*idle (protocol error)* state). Several tunable parameter changes can be bundled though; it is not required to switch to *recalibrating* after each individual new tunable parameter is set, but sufficient to do so once before the next *sampling*.

Likewise, the block life-cycle enforces that `DoStep()` is executed exactly once for each sampling (*sampling clock ticks* transition).

The block life-cycle also enforces that the new block inputs, to be used for the next *sampling*, must be ready before the execution of `DoStep()` starts (*all inputs set* condition of *sampling clock ticks* transition). It is however not enforced that every input must be assigned a new value each sampling. Since `Startup()` assigns all block-variables a well-defined value, including block inputs, following samplings will be well-defined even if an input is not set anew (assuming recalibration is done as described in the last but one paragraph). If a block-input is not updated before a sampling, it has the last value set. It is however very uncommon not to set all inputs each sampling; one reasonable scenario not to do so is if the block is super-sampled compared to some of its inputs (e.g., a sensor provides a new input value every 2ms, but the block is sampled every 1ms because of other faster changing inputs).

E-1: The following C99 pseudo-code snippets sketch typical system integration scenarios for blocks.

All examples share the following conventions. It is assumed that the Production Code generator mapped the block-interface methods `Startup()`, `Recalibrate()` and `DoStep()` to equally named C functions that expect the block-variables to operate on as argument, e.g., a struct pointer; to that end, `c` is a constant pointer to the static struct holding the block-variables (it encapsulates a single block instance). Prose text bracket by `[[` and `]]` denotes arbitrary C code implementing the respective action, but does not interact any further with the block-interface than denoted.

The most common integration scheme, with support for recalibration throughout samplings, is:

```
1 /*
2     Initialization:
3 */
4 Startup(c); /* Assigns every block-variable a value, particularly outputs. */
5 [[ process initial outputs of block ]]
6
7 /*
8     Sampling cycle:
9 */
```

```
10 while ([[ block not shutdown ]])
11 {
12     if ([[ recalibration desired ]])
13     {
14         [[ set new tunable parameters of block ]]
15         Recalibrate(c); /* Recompute dependent parameters of block. */
16     }
17     [[ set new inputs of block ]]
18     [[ wait until sampling clock ticks ]]
19     DoStep(c); /* Recompute internal states and outputs of block. */
20     [[ process new outputs of block ]]
21 }
```

A more simple integration scenario may not utilize recalibration throughout sampling, but only once immediately after initialization:

```
1  /*
2      Initialization:
3  */
4  Startup(c); /* Assigns every block-variable a value, particularly outputs. */
5  [[ process initial outputs of block ]]
6  [[ set new tunable parameters of block ]]
7  Recalibrate(c); /* Recompute dependent parameters of block. */
8
9  /*
10     Sampling cycle:
11 */
12 while ([[ block not shutdown ]])
13 {
14     [[ set new inputs of block ]]
15     [[ wait until sampling clock ticks ]]
16     DoStep(c); /* Recompute internal states and outputs of block. */
17     [[ process new outputs of block ]]
18 }
```

An even more simple integration scenario may not require recalibration at all, effectively transforming tunable and dependent parameters to constants since they can not change anymore after initialization:

```
1  /*
2      Initialization:
3  */
4  Startup(c); /* Assigns every block-variable a value, particularly outputs. */
5  [[ process initial outputs of block ]]
6
7  /*
8      Sampling cycle:
9  */
10 while ([[ block not shutdown ]])
11 {
12     [[ set new inputs of block ]]
13     [[ wait until sampling clock ticks ]]
14     DoStep(c); /* Recompute internal states and outputs of block. */
15     [[ process new outputs of block ]]
```

```
16 }
```

A Production Code generator can optimize this scenario, leveraging on enhanced constant-folding.

An advanced integration scenario might also require several different recalibartions and input modifications depending on the state of the runtime environment:

```
 1 /*
 2     Initialization:
 3 */
 4 Startup(c); /* Assigns every block-variable a value, particularly outputs. */
 5 [[ process initial outputs of block ]]
 6
 7 /*
 8     Sampling cycle:
 9 */
10 while ([[ block not shutdown ]])
11 {
12     /* Handle default setup: */
13     [[ set new inputs of block ]]
14
15     /* Handle first special case, modifying the default: */
16     v1 = [[ some value provided by the runtime environment ]];
17     t1 = [[ read tunable parameter t1 ]];
18     o1 = [[ read output o1 ]]; /* Previous sampling output, or initial if first
   sampling. */
19     if (o1 / t1 > v)
20     {
21         [[ set input i1 to v1 ]]
22         if (t1 > 2*v)
23         {
24             [[ set tunable parameter t1 ]]
25             Recalibrate(c);
26         }
27     }
28
29     /* Handle second special case (may amend the first case): */
30     v2 = [[ some value provided by the runtime environment ]];
31     t2 = [[ read tunable parameter t2 ]];
32     if (t2 < v2)
33     {
34         [[ set tunable parameter t2 ]]
35         Recalibrate(c); /* Recompute dependent parameters of block. */
36         [[ set input i2 to input i1 ]]
37     }
38
39     /* Everything is prepared for next sampling: */
40     [[ wait until sampling clock ticks ]]
41     DoStep(c); /* Recompute internal states and outputs of block. */
42     [[ process new outputs of block ]]
43 }
```

3.2.4. General Syntactic and Semantic Rules

Lexemes

G-1.1 — G-1.7 (white space characters):

```
1 character = ? any valid ISO/IEC 10646:2020 code point ? ;
2
3 white-space = { space | new-line-character | comment } - ( ) ;
4
5 space = " " | ? tabulator (ISO/IEC 10646:2020 code point 9) ? ;
6
7 new-line-character =
8     ? carriage return, line feed or carriage return followed by line feed
9       (ISO/IEC 10646:2020 code point 13 or 10 or 13 followed by 10) ? ;
10
11 comment =
12     "/*",
13     {
14         { character } - ( { character }, ( "/*" | "*/" ), { character } )
15       | comment
16     },
17     "*/" ;
```

G-1.8 — G-1.17 (constants):

```
1 boolean = "false" | "true" ;
2
3 digit = (* ? any ISO/IEC 10646:2020 code point in range [48, 57]: ? *)
4     "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
5
6 non-zero-digit = digit - ( "0" ) ;
7
8 integer = [ "-" ], ( "0" | positive-integer ) ;
9
10 positive-integer = non-zero-digit, { digit } ;
11
12 real = integer-places, decimal-places, [ exponent ] ;
13
14 integer-places = integer ;
15
16 decimal-places = ".", digit, { digit } ;
17
18 exponent = "e", ( "+" | "-" ), digit, { digit } ;
19
20 constant = boolean | integer | real ;
```

G-1.19 — G-1.26 (names):

```
 1 keyword =
 2        "block" | "protected" | "public" | "end"
 3      | "record"
 4      | "function" | "method" | "signals" | "algorithm"
 5      | "input" | "output"
 6      | "Boolean" | "Integer" | "Real"
 7      | "limit"
 8      | "if" | "signal" | "in" | "then" | "elseif" | "else"
 9      | "for" | "loop"
10      | "and" | "or" | "not" |
11      | "size"
12      | "self"
13      (* reserved for future extensions: *)
14      | "while" | "do" | "until"
15      | "break" | "return"
16      | "enumeration"
17      | "__", identifier ;
18
19 alphabetic-character =
20      (* ? any ISO/IEC 10646:2020 code point in ranges [65, 90] or [97, 122]: ? *)
21       "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
22      | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
23      | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M"
24      | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" ;
25
26 identifier = ( alphabetic-character, { alphabetic-character | "_" | digit } ) - ( keyword
   ) ;
27
28 quoted-identifier =
29      "'",
30      (
31        "previous", "(", scalarized-reference, ")"
32      | "derivative", "(", quoted-identifier-higher-order-derivative, ")"
33      | scalarized-reference
34      ),
35      "'" ;
36
37 quoted-identifier-higher-order-derivative =
38        scalarized-reference
39      | "derivative", "(", quoted-identifier-higher-order-derivative, ")" ;
40
41 scalarized-reference =
42      ( identifier | keyword ),
43      [ fixed-dimensions ],
44      { ".", ( identifier | keyword ), [ fixed-dimensions ] } ;
45
46 fixed-dimensions = "[", positive-integer, { ",", positive-integer }, "]" ;
47
48 name = identifier | quoted-identifier ;
```

S-1.1 (longest match / Meta-rules, lexical-structure): Given the following EBNF grammar: $G_{lexemes}$ = { *? all meta-identifiers of* **G-1.1 — G-1.26** *concatenated by* | *? }*. Let αβγδ be an arbitrary GALEC program *P*, with α being an arbitrarily long sequence of characters matched throughout

a left-most derivation of *P* according to $G_{lexemes}$, β and γ being arbitrary long but not empty sequences of characters, and δ being an arbitrary long sequence of characters. Let G_1 and G_2 be any two different rules of **G-1.1 — G-1.26** that can be applied next throughout the left-most derivation of *P* according to $G_{lexemes}$. Assume G_1 would match β and G_2 would match βγ; in that case G_1 is not applicable.

For every left-most derivation of any GALEC program *P* it must hold that the sequence of **G-1.1 — G-1.26** applications is the same as the sequence of **G-1.1 — G-1.26** applications for the left-most derivation of *P* by $G_{lexemes}$.

> **E-1:** Let α, β, γ and δ be as defined above, with β = i, γ = 4 and δ starts with *white-space*. The next rule applied within the set **G-1.1 — G-1.26** must be **G-1.21** (*identifier*).

S-1.2 (universality of white space / Meta-rules, lexical-structure): Except for rules **G-1.1 — G-1.26**, `{ white-space }` is implicitly preceding and following each syntactic-factor of a syntax-rule (cf. ISO/IEC 14977).

> **E-1:** The expanded rule of **G-TODO**, showing its implicit white-space, is:

```
 1 state-reference =
 2     { white-space },
 3     "self",
 4     { white-space },
 5     ".",
 6     { white-space },
 7     name,
 8     { white-space },
 9     [ { white-space }, computed-dimensions, { white-space } ],
10     { white-space },
11     {
12         { white-space },
13         ".",
14         { white-space },
15         name,
16         { white-space },
17         [ { white-space }, computed-dimensions, { white-space } ],
18         { white-space }
19     },
20     { white-space } ;
```

> **E-2:** According to **E-1**, the following is a valid *state-reference*:

```
 1 self.
 2 a . b [2 ] /* vector */
 3     . 'c[2].d[3]'
 4     .
```

```
5    m [
6 3 , /* matrix access */ 4
7 ]
```

E-3: The following is an illegal *quoted-identifier* due to *white-space* within its quotes:

```
1 'a . b [2 ] /* vector */
2    .
3    m [
4 3 , /* matrix access */ 4
5 ]'
```

S-1.3 (primitive names / Name-analysis, terminology): *Names*, *identifiers* and *quoted-identifiers* are primitive names. Let α be lexically-equivalent to a primitive name *N*; α is the name of *N*. Syntactic terms with a name are called named. Let α be the name of a named syntactic term *N*; we say *N* has name α and *N* is named α.

> **R-1:** The set of named syntactic terms can be easily extended by semantic rules by just defining a name for a syntactic term.

R-1.1 (scalarization and quoted identifiers / Traceability): *Quoted-identifiers* are provided to denote scalarized entities — typically multi-dimensional nested components of the original physics-model whose elements are flattened to individual scalar entities for further numeric optimisations throughout the generation of an algorithmic solution. By reusing the original multi-dimensional query for an element that is now an independent scalar as the scalar's name, traceability can be achieved.

The `previous(α)` and `derivative(α)` notations are intended to be used for support-variables holding the value a variable α had at the end of the last control-cycle or its derivative respectively. Many physics-modeling languages, like Modelica, provide such values implicitly by means of operators applicable to any variable. Since algorithmic solutions are discrete however, no continuous derivatives exist. And the meaning of previous, in terms of the last value before the current, depends on the applied discretization scheme. For backward discretization it indeed is the last control-cycle's value; for forward discretization however, it is the current value. For mixed schemes the meaning is unclear. Due to these issues, no specific operators are provided. Instead, algorithmic solutions have to explicitly introduce variables to hold values or compute derivates. The `previous(α)` and `derivative(α)` notations can be used to give the explicit variables introduced for the variables α that have been subject of such implicit operations convenient names, ultimately increasing traceability.

> **E-1:** The Modelica model

```
 1 model M
 2     model MI
 3         model MII
 4             Real x;
 5             Boolean y;
 6         equation
 7             ...
 8         end MII;
 9         MII b[3];
10     equation
11         ...
12     end MI;
13     MI a[2];
14 equation
15     ...
16 end M;
```

could be scalarized to

```
 1 Real 'a[1].b[1].x';
 2 Boolean 'a[1].b[1].y';
 3 Real 'a[1].b[2].x';
 4 Boolean 'a[1].b[2].y';
 5 Real 'a[1].b[3].x';
 6 Boolean 'a[1].b[3].y';
 7 Real 'a[2].b[1].x';
 8 Boolean 'a[2].b[1].y';
 9 Real 'a[2].b[2].x';
10 Boolean 'a[2].b[2].y';
11 Real 'a[2].b[3].x';
12 Boolean 'a[2].b[3].y';
```

Further numeric analyses could conclude that `a.b[2]` is an alias or irrelevant for the simulation for which reason it can be eliminated, reducing the set of individual scalar state variables to only

```
1 Real 'a[1].b[1].x';
2 Boolean 'a[1].b[1].y';
3 Real 'a[1].b[3].x';
4 Boolean 'a[1].b[3].y';
5 Real 'a[2].b[1].x';
6 Boolean 'a[2].b[1].y';
7 Real 'a[2].b[3].x';
8 Boolean 'a[2].b[3].y';
```

R-1.2 (reserved keywords): G-1.19 (*keyword*) reserves certain character sequences for future language extensions; the respective sequences are not used elsewhere in the grammar. The sequences `while`, `do` and `until` are reserved for a potential future introduction of non-bounded or more complicated loops, `return` and `break` for potential early function and loop

exit statements and `enumeration` for a potential extension with enumeration types. Such reservations do not imply by any means that the language indeed will be extended accordingly; they rather serve to preserve up-wards compatibility of code when respective language extensions are added. The `"__", identifier` alternative reserves names that might collide with internal compiler macros of further tooling; it is in the spirit of 6.11.9 of ISO/IEC 9899:TC3.

> **E-1:** `Boolean until;` is not a *local-variable-declaration* due to `until` being a reserved keyword.

Blocks and Declarations: Control-state and -cycle (memory and inter-functional flowchart)

G-2.1 — G-2.3 (blocks, state compartments and functions):

```
1 block =
2     "block",
3     name,
4     { state-entity-declaration } (* TODO: must be inputs, followed by outputs followed by
  parameters *),
5     "protected",
6     { state-compartment-declaration },
7     { state-entity-declaration },
8     { error-signal-declaration },
9     { function-declaration },
10     "public",
11     { function-declaration },
12     "end",
13     name,
14     ";" ;
15
16 error-signal-declaration = "signal", identifier, ";" ;
17
18 state-compartment-declaration =
19     "record",
20     name,
21     { state-entity-declaration },
22     "end",
23     name,
24     ";" ;
25
26 function-declaration =
27     ( "function" | "method" ),
28     name,
29     [ signal-interface ],
30     { parameter-declaration },
31     [ "protected", { local-variable-declaration } ],
32     "algorithm",
33     { statement },
34     "end",
35     name,
36     ";" ;
```

```
37
38 signal-interface = "signals", identifier, { ",", identifier }, ";" ;
```

G-2.4 — G-2.12 (state entity, parameter and local variable declarations):

```
 1 state-entity-declaration =
 2     [ "constant" | "parameter" ], (* TODO: Definition of terms and semantic of constants
   and tuneable and dependent parameters *)
 3     variable-declaration ;
 4
 5 parameter-declaration = data-flow-direction, variable-declaration ;
 6
 7 local-variable-declaration = variable-declaration ;
 8
 9 data-flow-direction = "input" | "output" ;
10
11 variable-declaration =
12     ( primitive-type | state-compartment-reference ),
13     name,
14     [ constant-dimensions ],
15     ";" ;
16
17 primitive-type = "Boolean" | "Integer" | "Real" ;
18
19 state-compartment-reference = name ;
20
21 constant-dimensions =
22     "[",
23     ( derived-dimension | constant-scalar-integer-expression ),
24     { ",", ( derived-dimension | constant-scalar-integer-expression ) },
25     "]" ;
26
27 derived-dimension = ":" ;
```

R-2.1 (unique start symbol): According to ISO/IEC 14977 and **S-1.2**, *block* is the only start symbol.

S-2.1 (consistent naming / Name-analysis): The 2nd and 12th child of *blocks* must be lexically-equivalent. The 2nd and 5th child of a *state-compartment-declaration* must be lexically-equivalent. The 2nd and 9th child of a *function-declaration* must be lexically-equivalent.

E-1: The following *block* fragment is illegal due to inconsistent state compartment and function names:

```
 1 record GearBox /* Illegal: GearBox and gearBox not lexically-equivalent. */
```

```
 2     Real w;
 3 end gearBox;   /* Illegal: GearBox and gearBox not lexically-equivalent. */
 4
 5 method UpdateGearBox /* Illegal: UpdateGearBox and 'UpdateGearBox' not lexically-
   equivalent. */
 6     input Real x;
 7 algorithm
 8     self.gearBox.w := (x / self.gearBox.w) * self.gearBox.w;
 9 end 'UpdateGearBox'; /* Illegal: UpdateGearBox and 'UpdateGearBox' not lexically-
   equivalent. */
```

S-2.2 (state compartments, components and variables and control-inputs and -outputs; input and output parameters; local variables / Type-analysis, terminology): A *state-entity-declaration* without *primitive-type* is called state component, otherwise it is called state variable. State components and variables are called state entities.

State-compartment-declarations are called state compartment; the state entities contained in a state compartment are called its local entities (thus, state compartments have local components and variables).

State entities whose *data-flow-direction* is lexically-equivalent to `input` are called control-input; state entities whose *data-flow-direction* is lexically-equivalent to `output` are called control-output. Control-inputs and -outputs must be state variables and not be part of state compartments (i.e., state components cannot be control-inputs or -outputs nor can such be local entities of any state compartment).

A *parameter-declaration* whose *data-flow-direction* is lexically-equivalent to `input` is called an input parameter; otherwise it is called an output parameter. Input and output parameters are called parameters.

Local-variable-declarations are called local variable.

E-1: The following valid *block* fragment defines various non-functional entities:

```
 1 /*
 2     State compartment that is the control-state (cf. S-2.8).
 3     It has two local state entities, one variable and one component.
 4 */
 5 block Controller
 6     record C
 7         Real    r;
 8         Integer i;
 9     end C;
10
11     Integer i;      /* State entity that is a state variable. */
12     C c;            /* State entity that is a state component. */
13
14     function f
15         output Real out_1[size(in, 1)]; /* Parameter that is an output parameter.
```

```
     */
16        input  Real in[:];              /* Parameter that is an input  parameter.
     */
17        output Real out_2[size(in, 1)]; /* Parameter that is an output parameter.
     */
18    protected
19        Integer s; /* Local variable. */
20    algorithm
21        s := 0;
22        for i in 1:size(in, 1) loop
23            s := s + in[i];
24        end for;
25        out_1 := in / s;
26        out_2 := s * in;
27    end f;
28 end Controller;
```

S-2.3 (stateless and stateful functions / Side-effect-analysis, terminology): *Function-declarations* are just called functions. Functions whose first child is lexically-equivalent to `method` are called stateful function; otherwise, they are called stateless function.

> **R-1 (state of stateful functions / Runtime-semantic):** The motivation to separate stateful functions from stateless is, that the latter cannot change the control-state by any means; only stateful functions can write state variables as long as they are not control-inputs (cf. **S-TODO.TODO (non-writeable control-inputs, input parameters and loop iterators; side-effect-freeness of stateless functions / Side-effect-analysis)**). There are no restrictions on reading state variables however, including control-inputs and -outputs; stateless functions therefore still can depend on the control-state. These restrictions on when control-state changes are permitted improve readability of GALEC code and enable the generation of Production Code leveraging on parallel computing (cf. **S-3.TODO:R-1 (isolated side-effects of stateful function calls and parallel computing / Runtime-semantic)**).

S-2.4 (names of state compartments and entities, functions, parameters and local variables / Name-analysis): State compartments, state entities, functions, parameters and local variables are named.

The name of a state compartment is the name of its 2nd child.

The name of a function is the name of its 2nd child.

The name of a state entity, parameter and local variable is the name of its *variable-declaration* where the name of a *variable-declaration* is the name of its 2nd child.

> **E-1:** The following valid *block* fragment defines various names:

```
 1 block Controller2
 2     Real 'derivative(shaft[2].x)';   /* Scalar named 'derivative(shaft[2].x)'. */
 3     GearBox 'shaft[2].gear'[3];      /* State component vector named
   'shaft[2].gear'. */
 4     Real w;                          /* Scalar named w. */
 5
 6     method 'shaft[2].gear.update'           /* Stateful function named
   'shaft[2].gear.update'. */
 7         input Real 'previous(shaft[2].y)';   /* Scalar input parameter named
   'previous(shaft[2].y)'. */
 8         input Integer index;                 /* Scalar input parameter named index.
   */
 9     protected
10         Real exp_y;                          /* Scalar local variable named exp_y.
   */
11     algorithm
12         exp_y := exp('previous(shaft[2].y)');
13         self.'shaft[2].gear'[index].w :=
14         exp_y^2 - self.'derivative(shaft[2].x)' * exp_y;
15     end 'shaft[2].gear.update';
16 end Controller2;
```

S-2.5 (unique declarations (Part I) / Name-analysis): *Blocks* must not contain two different functions or state compartments with equivalent names. Functions and state compartments must have different names. Parameters and local variables must not be named like functions or state compartments. Functions must not contain two different parameters or local variables with equivalent names. Parameters and local variables contained in the same function must have different names. Different local entities of a state compartment must have different names.

S-TODO incorporates and adds further unique declaration restrictions for iterators.

R-1 (MISRA C:2012 compliance): The restriction that parameters and local variables must not have function or state compartment names is introduced to avoid hiding of outer-scope declarations in accordance with MISRA C:2012, Rules 5.3, 5.8 and 5.9.

R-2 (separate name-space for state entities): State entities can have the name of a state compartment, function, parameter, local variable or iterator because, according to **S-TODO**, they can only be accessed using a *state-reference* which always starts with the unique sequence `self.`. Thus, the intention to refer to a state entity always is clearly denoted; state entities are within their own separate name-space. State entities not local to the same state compartment can have equivalent names because they are always differently accessed.

E-1: The following *block* is illegal due to hiding of outer-scope declarations and re-

declarations (for the definition of preceding and follows cf. *M-1.3*; for hiding of outer-scope declarations cf. the ISO/IEC 9899:TC3 and MISRA C:2012 standards):

```
1  /*
2      The comments in this example are just abbreviations for
3          /* Illegal: Equally named <C>. */
4      where <C> is the comment and refers to the relative locations of
5      equally named entities.
6  */
7
8  record efmu /* state compartment follows */
9  end efmu;
10
11 record efmu /* state compartment preceding */
12     C v; /* state entity follows */
13     Real v; /* state entity preceding */
14 end efmu;
15
16 record C /* function and local variable follow */
17 end C;
18
19 method DoStep /* function follows */
20 protected
21     Real v; /* local variable follows */
22     Real v; /* local variable preceding */
23 algorithm
24 end DoStep;
25
26 method DoStep /* function preceding */
27 protected
28     Integer f; /* function follows */
29 algorithm
30 end DoStep;
31
32 function C /* state compartment preceding and local variable follows */
33     output Real r; /* local variable follows */
34 protected
35     Boolean r[4]; /* parameter preceding */
36 algorithm
37 end C;
38
39 function f /* local variable preceding */
40 protected
41     Integer C; /* state compartment and function preceding */
42 algorithm
43 end f;
```

E-2: The following valid *block* has no re-declarations or hiding of outer-scope declarations:

```
1  block Controller3
2      C C; /* Type and name are lexically-equivalent. */
3      /*
```

```
 4          Name lexically-equivalent to self.C.r, parameter of
 5          function f and local variable of function DoStep:
 6     */
 7     Real r;
 8
 9     record C
10         /*
11             Name lexically-equivalent to self.r, parameter of
12             function f and local variable of function DoStep:
13         */
14         Real r;
15         Boolean DoStep; /* Name lexically-equivalent to function DoStep. */
16     end C;
17
18     method DoStep
19     protected
20         /*
21             Name lexically-equivalent to self.r, self.C.r and
22             parameter of function f:
23         */
24         Real r;
25     algorithm
26     end DoStep;
27
28     method Startup
29     algorithm
30     end Startup;
31
32     function f
33         /*
34             Name lexically-equivalent to self.r, self.C.r and
35             local variable of function DoStep:
36         */
37         output Real r;
38     algorithm
39     end f;
40 end Controller3;
```

S-2.6 (state compartment lookup / Name-analysis): Let *R* be a *state-compartment-reference*. There must exist a state compartment *D* named like the *name* of *R*; according to **S-2.5**, *D* must be unique. We say *R* refers to *D*.

S-2.7 (types of state entities, parameters and local variables / Type-analysis): The first child of a *variable-declaration D* defines its type. If, and only if, *D* contains a *primitive-type T*, the type of *D* is lexically-equivalent to *T*. In this case *D* has a variable type; otherwise, the type of *D* is the state compartment its *state-compartment-reference* refers to and *D* has a component type.

The type of a state entity, parameter and local variable is the type of its *variable-declaration*.

The type of parameters and local variables must not be a component type.

E-1: The following valid *block* fragment defines entities of various types. Note, that type and dimensionality (cf. **S-2.12**) are orthogonal characteristics; declarations can combine every type with any dimensionality.

```
 1 block Controller
 2    record GearBox
 3       Real w;        /* State variable of type Real. */
 4    end GearBox;
 5
 6    Boolean s;        /* State variable of type Boolean. */
 7    Real w[3];        /* State variable of type Real. */
 8    GearBox g[3];     /* State component of type GearBox. */
 9
10    function 'g.w.T_sum'
11       input Integer T[3, 3]; /* Input parameter of type Integer. */
12       output Real y;         /* Output parameter of type Real; */
13    protected
14       Real 'g.w'[3];         /* Local variable of type Real; */
15    algorithm
16       for i in 1:3 loop
17          'g.w'[i] := emfu.g[i].w;
18       end for;
19       y := (if efmu.s then 1 else -1) * sum(real(T) * 'g.w');
20    end 'g.w.T_sum';
21 end Controller;
```

E-2: The following function is illegal due to parameters and local variables with component types:

```
 1 method UpdateGearBox
 2    input Shaft s; /* Illegal: Input parameter has a component type. */
 3    input Integer i;
 4 protected
 5    GearBox g; /* Illegal: Local variable has a component type. */
 6 algorithm
 7    g := s.gear[i]; /* Illegal: Cf. S-TODO.TODO (type of references / Type-
   analysis):L-1. */
 8    g.w := (g.x / g.w) * g.w;
 9 end UpdateGearBox;
```

S-2.8 (state compartment composition graph, control-state and control-state extent / Termination-analysis): We define the following directed graph *G*. For every state compartment *C*, *G* contains a node labeled with the name of *C*. For every state component with type *T* and local to *C*, we add a directed edge from *C* to *T*. *G* is called the state compartment composition graph.

The state compartment composition graph must be cycle-free and it must contain a node *N*

labeled `efmu` from which all other nodes are reachable (and which therefore is its only root, i.e., the only node without incoming edges).

The state compartment named `efmu` is called the control-state.

Control-inputs and outputs must be local state entities of the control-state.

L-1 (unique, all-embracing, finite control-state extent / Runtime-semantic): According to ***S-2.5***, state compartments are unique for which reason the state compartment composition graph cannot contain two nodes with equivalent label. It can contain multiple edges between two nodes however, since for each state component of type n_t contained in state compartment n_s the state compartment composition graph will contain a separate edge from n_s to n_t. Nodes can also have several incoming edges from different nodes, since state components of equivalent type can be part of different state compartments. Considering all these constraints, the state compartment composition graph must be a directed, cycle-free graph with unique root (and not necessarily a directed tree).

TODO: transform state compartment composition graph to tree defining control-state extent. Argue why that one is unique, all-embracing and finite and why that is good-for/required-by embedded code. Define that in the context of runtime-semantic the term control-state always refers to the control-state extent.

The control-state must be unique; and considering the restrictions of the state compartment composition graph, it must comprise all state entities defined, i.e., be all-embracing (reachability) and finite (cycle-free).

E-1: The following state compartments are illegal because they have a cyclic composition, miss the `efmu` root and have other roots:

```
 1 record C1 /* Illegal: Part of C1, C2, C3 cycle. */
 2    C2 c;
 3 end C1;
 4
 5 record C2 /* Illegal: Part of C1, C2, C3 cycle. */
 6    C3 c;
 7 end C2;
 8
 9 record C3 /* Illegal: Part of C1, C2, C3 cycle. */
10    C1 c;
11 end C3;
12
13 record C /* Illegal: Non-efmu root. */
14    C2 c;
15 end C;
16
17 /* Illegal: The control-state (efmu root) is missing. */
```

E-2: The following state compartments are illegal because the control-state is not a root:

```
 1 record C1
 2 end C1;
 3
 4 record C2
 5 end C2;
 6
 7 /*
 8     Illegal: Control-state is not a root (C1 not reachable from
 9             efmu in state compartment composition graph):
10 */
11 record efmu
12     C2 c;
13 end efmu;
```

E-3: The following state compartments are illegal because there are control-inputs and -outputs that are not local state entities of the control-state or are state components (cf. **S-2.2**):

```
 1 record C
 2     input Real i;   /* Illegal: Control-input not local to the control-state. */
 3     output Real o; /* Illegal: Control-output not local to the control-state. */
 4 end C;
 5
 6 record efmu
 7     C c;
 8     input Real i_1;
 9     output Real o_1;
10     input C i_2;    /* Illegal: Control-input is a state component. */
11     output C o_2;   /* Illegal: Control-output is a state component. */
12 end efmu;
```

S-2.10 (locally and transitively called functions, static function call graph and recursion-freeness / Termination-analysis): Let C_f be the set of names of the *function-calls* contained in a function f; C_f is called the local function call set of f and we say for each function f_c whose name is in C_f that it is locally called by f and that f locally calls f_c.

We define the following directed graph G. For every function f (including builtin functions), G contains a node labeled with the name of f. For every function f_c locally called by a function f, we add a directed edge from f to f_c. G is called the static function call graph.

Let n be a node of the static function call graph and n_r be a node reachable from n; let f be the function named like the label of n and f_r the function named like n_r. We say f_r is transitively called by f and f transitively calls f_r.

The static function call graph must be cycle-free.

S-2.11 (initialization and control-cycle functions / Name-analysis): Every *block* must contain a function named `Startup`; respective functions are called initialization function. Initialization functions must be stateful and *parameter-declaration*-free. Initialization functions must not locally call user-defined functions (i.e., initialization functions can only call builtin functions).

Every *block* must contain a function named `DoStep`; respective functions are called control-cycle function. Control-cycle functions must be stateful and *parameter-declaration*-free.

All user-defined functions, except the control-cycle and initialization functions, must be transitively called from the control-cycle function (thus, let $N_{user-defined}$ be the set of nodes of the static function call graph labeled with the name of a user-defined function, excluding the control-cycle and initialization functions, and let $n_{control-cycle}$ be the node labeled with the name of the control-cycle function: $\forall n_{user-defined} \in N_{user-defined}$: $n_{user-defined}$ *is reachable from* $n_{control-cycle}$).

> **R-1 (controller interface / Runtime-semantic):** According to **S-2.5**, the initialization and control-cycle functions are unique. They and the control-state are the controller interface, i.e., the functionality visible for the runtime environment executing the eFMU.

> **L-1 (initialization and control-cycle; control-state consistency / Runtime-semantic):** At runtime, the Production Code generated for the initialization function must be executed at least once before the production code for the control-cycle function is executed for the very first time; its purpose is to initialize the control-state at startup and provide the outputs for the first clock tick. Thereafter, the Production Code generated for the control-cycle function must be executed at every sampling-step to update the blocks's control-state and compute the block outputs.
>
> To ensure the consistency of the control-state and the computations based on it, the runtime environment must never call any function of the controller interface of an eFMU while any of its functions is still executing. Any runtime environment interaction with an eFMU must be via its controller interface; and any such interaction must satisfy above restrictions. This prohibits third parties, for example, to recalibrate an eFMU while its control-cycle function is executing or to execute user-defined functions that are not part of the controller interface.
>
> Note, that production code is not restricted in terms of parallel execution of different controllers (i.e., independent applications of the Production Code generated for a single or different GALEC programs) as long as the generated production code and its runtime environment ensure that each individual application (i.e., block) satisfies above restrictions.

S-2.12 (scalars, multi-dimensions, vectors and matrices / Dimensionality-analysis,

terminology): State entities, parameters and local variables without *constant-dimensions* are called scalar; otherwise multi-dimension. Let $d = [\alpha_1, ..., \alpha_n]$ be the *constant-dimensions* of a state entity, parameter or local variable v; in that case v is n-dimensional/multi-dimensional, n is the number of its dimensions and each α_i with $1 \leq i \leq n$ is its i'th dimension. Scalars are zero-dimensional. If, and only if, v is one-dimensional it is called vector; if, and only if, it is two-dimensional, matrix. The first dimension of a matrix are its rows, the second its columns.

E-1: The following *block* fragment declares various scalars and multi-dimensions (denoted by using capitals only):

```
 1 block S
 2     /*
 3         A 0-DIMENSIONAL state component, i.e.,
 4         a state component SCALAR:
 5     */
 6     C a;
 7     /*
 8         A 1-DIMENSIONAL state component, i.e.,
 9         a state component VECTOR:
10     */
11     C b[2];
12      /*
13         A 2-DIMENSIONAL state component, i.e.,
14         a state component MATRIX
15         with 2 ROWS and 3 COLUMNS:
16     */
17     C c[2,3];
18     /*
19         A 3-DIMENSIONAL state component, i.e.,
20         a MULTI-DIMENSIONAL state component, i.e.,
21         a state component MULTI-DIMENSION,
22         that is neither, a VECTOR nor a MATRIX:
23     */
24     C d[1,1,1];
25     Real r[3,3]; /* a state variable MULTI-DIMENSION */
26
27     function f
28         input  Real i[:,:]; /* an input parameter MATRIX */
29         output Integer o;   /* an output parameter SCALAR */
30     protected
31         Integer j[size(i,1)]; /* a MULTI-DIMENSIONAL local variable */
32         Real    k[size(i,2)]; /* a VECTOR, i.e., a MULTI-DIMENSION */
33     algorithm
34     end f;
35 end S;
```

S-2.13 (dimensional-sizes of state entities / Dimensionality-analysis): State entities must not contain *dimension-queries* or *derived-dimensions*.

TODO: More relaxed alternative: Contained *dimension-queries* must refer to state entities; the

resulting dependency graph must be cycle-free. More restrict alternative: The *constant-scalar-integer-expressions* of their *constant-dimensions* must derive to *positive-integers.*

TODO: Static computation of actual dimensions.

S-2.14 (dimensional-sizes of parameters and local variables / Dimensionality-analysis): Output parameters and local variables must not contain *derived-dimensions* (i.e., only input parameters can contain *derived-dimensions*).

TODO: Static computation of actual dimensions.

S-2.15 (signature of functions; procedures / Type-analysis, terminology): The parameters a function contains define its signature, i.e., its input-arity, output-arity and order of inputs and outputs.

Let S_{input} be the set of all input parameters contained in a function f; let S_{output} be the set of all output parameters contained in f. The inputs of f are the tuple $T_{input} = (p_1,...,p_n)$ with $n = |T_{input}| = |S_{input}|$ and $\forall p_i,p_j \in T_{input}$; $i,j \in \mathbb{N}+$; $i < j \le n$: $p_i \in S_{input} \land p_j \in S_{input} \land p_i$ *is preceding* p_j; likewise, the outputs of f are the tuple $T_{output} = (q_1,...,q_m)$ with $m = |T_{output}| = |S_{output}|$ and $\forall q_i,q_j \in T_{output}$; $i,j \in \mathbb{N}+$; $i < j \le m$: $q_i \in S_{output} \land q_j \in S_{output} \land q_i$ *is preceding* q_j. The input-arity of f is n; its output-arity is m.

An input parameter is called the i'th input of a function f, if, and only if, it is the i'th element of the inputs of f; likewise, an output parameter is called the i'th output, if, and only if, it is the i'th element of the outputs. Trivially, an input parameter part of a function f is an input of f and an output parameter an output.

Functions of output-arity *0* are called procedure.

R-1: The signature of a function defines its whole interface, since the types and dimensions of input and output parameters are already defined by ***S-2.7*** and ***S-2.14***. Given for example a function of input-arity *3*, one can talk about the type and dimensionality of its second input.

Expressions: Scalar and Multi-dimensional Arithmetic

G-3.1 — G-3.4 (statically- and dynamically-evaluated expressions):

```
1 expression =
2     constant
3     | reference
4     | dimension-query
5     | function-call
6     | parenthesized-expression
7     | if-expression
```

```
 8       | multi-dimension-constructor
 9       | unary-operation
10       | binary-operation ;
11
12 parenthesized-expression = "(", expression, ")" ;
13
14 dimension-query = "size", "(", reference, ",", constant-scalar-integer-expression, ")" ;
15
16 constant-scalar-integer-expression = expression ;
```

G-3.5 — G-3.10 (operations):

```
 1 unary-operation =
 2     (
 3            "-", reference
 4         | "not", ( parenthesized-expression | if-expression )
 5     ) ;
 6
 7 binary-operation = expression, binary-operator, expression ;
 8
 9 binary-operator = arithmetic-operator | relational-operator | logical-operator ;
10
11 arithmetic-operator = "+" | "-" | "*" | "/" | "^" ;
12
13 relational-operator = "<" | ">" | "<=" | ">=" | "==" | "<>" ;
14
15 logical-operator = "and" | "or" ;
```

G-3.11 — G-3.14 (multi-dimension constructors, function calls and conditional expressions):

```
 1 multi-dimension-constructor =
 2     "{",
 3     multi-dimension-constructor-element,
 4     { ",", multi-dimension-constructor-element },
 5     "}" ;
 6
 7 multi-dimension-constructor-element = expression | multi-dimension-constructor ;
 8
 9 function-call = name, "(", [ expression, { ",", expression } ], ")" ;
10
11 if-expression =
12     "(",
13     "if",
14     expression,
15     "then",
16     expression,
17     { "elseif", expression, "then", expression },
18     "else",
19     expression,
```

```
20      ")" ;
```

S-3.1 (statically- and dynamically-evaluated expressions / Syntactical-structure, terminology): *Expressions* and all children of such, as well as *constant-scalar-integer-expressions*, are called expression. Expressions part of, or that are, a *constant-scalar-integer-expression* are called statically-evaluated; all other expressions are called dynamically-evaluated.

References contained in statically-evaluated expressions must either, be the third child of a *dimension-query* or refer to a *loop-iterator-declaration*. *Function-calls* contained in statically-evaluated expressions must refer to builtin functions.

> **R-1:** Considering that builtin functions are stateless (cf. **S-2.9:R-2**) and *loop-iterator-declarations* are unrelated to state variables (the value of a *loop-iterator-declaration* is a statically-defined, finite sequence of iteration-values within a fixed range, cf. **S-TODO**), statically-evaluated expressions cannot use or change the control-state. Their evaluation is control-state independent and therefore independent of control-inputs and -outputs (cf. **S-TODO**); they can be evaluated throughout Production Code generation, hence, statically-evaluated.
>
> Dynamically-evaluated expression on the other hand can directly or indirectly depend on the control-state and, by means of assignments, change it.

S-3.2 (operations, operators and arguments of operations / Type-analysis, terminology): *Binary-operations* and *unary-operations* are also just called operation. The 2'nd child of a *binary-operation* and the 1'st child of an *unary-operation* are called its operator. The 1'st and 3'rd child of a *binary-operation O* are called its first and second argument respectively; they are the arguments of *O*. The 2'nd child of an *unary-operation* is called its argument.

Let $\oplus$ be lexically-equivalent to the operator $O_\oplus$ of an operation O; we call O an $\oplus$-operation and $O_\oplus$ the $\oplus$-operator. If, and only if, O is a *binary-operation* it and its operator are called binary; otherwise unary.

> **E-1:** The expression `-v` is a unary `-`-operation, whereas `v_1 - v_2` is a binary `-`-operation; both can be either, statically- or dynamically-evaluated (cf. **S-3.1**) depending on their application context. For example, in `A[v_1 - v_2] := -v * A[v_1 - v_2]`, the binary `-`-operations are statically-evaluated whereas the unary `-`-operation is dynamically-evaluated.
>
> `not`-operations are always unary and `and`- and `or`-operations are always binary. Note, that they can be statically-evaluated, like in `A[(if remainderEuclidean(i, 2) == 0 and i <= size(A, 1) then i else size(A, 1))]`.

S-3.3 (operator precedence and associativity / Meta-rules, syntactical-structure): The following table defines a unique disambiguation for the syntactic ambiguities of *binary-operations* by means of an operator precedence and associativity for sequences of operators with equivalent precedence:

Operator classes (highest precedence to lowest)	Associativity of contained operators
^	right-to-left
*, /	left-to-right
+, -	left-to-right
<, >, <=, >=	left-to-right
==, <>	left-to-right
and	left-to-right
or	left-to-right

Binary-operations must satisfy the defined operator precedence and associativity.

A *binary-operation O* satisfies operator precedence, if, and only if, it does not contain *binary-operations* whose operator has a lower operator precedence than the operator of *O* and which themselves are not contained within a precedence-overriding non-terminal part of *O*. The precedence-overriding non-terminals are: *reference*, *dimension-query*, *function-call*, *parenthesized-expression*, *if-expression* and *multi-dimension-constructor*.

Operator associativity is satisfied if, and only if, *binary-operations* are derived left-most if their operator's associativity is left-to-right and right-most otherwise.

L-1 (strict evaluation order of expressions / Runtime-semantic): Operator precedence and associativity, together with syntactic rules **G-3.5** to **G-3.10** imply a well-defined order for the evaluation of operation sequences — an evaluation order. For example, production code generated for a sequence of additions a + b + c must evaluate it from left-to-right, i.e., first add a and b followed by adding the respective result and c. Thus, the evaluation order must not be changed by Production Code generators even for expressions that are associative in mathematics. Doing so acknowledges, that computational arithmetic is limited considering value overflows or floating point imprecision and that typically only GALEC code generators have the physics-model-specific numerical knowledge to select an appropriate evaluation order (for which reason Production Code generators should not change it). Enforcing an exact evaluation order also improves computational consistency between different Production Code generators.

E-1: The following examples illustrate the disambiguation enforced by **S-3.3**. They leverage on the fact that, using parentheses, every syntax-wise ambiguous expression can be explicitly disambiguated such that **S-3.3** is not required.

Each example consists of a mathematical notation of the expression, followed by three semantically equivalent expressions, each on a separate line. The first line shows a valid — with minimal parenthesizing — GALEC version of the expression requiring **S-3.3** for disambiguation. The second line shows a version with minimal usage of parenthesis that does not requiring **S-3.3**. The third line discloses the actual evaluation order of the expression by parenthesizing even expression parts whose evaluation order is already well-defined by syntactic rules only.

Expression 1:

$a + b + c$

```
1  a + b  + c
2 (a + b) + c
3 (a + b) + c
```

Expression 2:

$a + b*c/d/e*f + g$

```
1  a + (   b * c  / d  / e  * f ) + g
2 (a + ((((b * c) / d) / e) * f)) + g
3 (a + ((((b * c) / d) / e) * f)) + g
```

Expression 3:

$-a^{(-b}2) * c$

```
1 ( -a ^  -b ^2 ) * c
2 ( -a ^( -b ^2)) * c
3 ((-a)^((-b)^2)) * c
```

Expression 4:

$3 - a^{(-b}2)$

```
1 3 - (a^  -b^2  )
2 3 - (a^( -b^2))
3 3 - (a^((-b)^2))
```

Expression 5:

$3 - -a^{(-b}2)$

```
1 3 - ( -a ^  -b ^2 )
2 3 - ( -a ^( -b ^2))
3 3 - ((-a)^((-b)^2))
```

Expression 6:

(a < b) <> (c < d) == e < f and (g == h) <> i or j + k < l - m and n or o

```
1   ( (a < b) <> (c < d)   == (e < f)) and (( g == h   <> i ) or ((j + k) < (l - m)))
    and (n or  -o)
2   ((((a < b) <> (c < d)) == (e < f)) and (((g == h) <> i ) or ((j + k) < (l - m))))
    and (n or  -o)
3   ((((a < b) <> (c < d)) == (e < f)) and (((g == h) <> i ) or ((j + k) < (l - m))))
    and (n or (-o))
```

E-2: The parenthesis of `a^(2*b)` cannot be omitted because `a^2*b` is an invalid expression and `(a^2)*b` is not equivalent. For example, `a^(2*b)` yields *64* if a is *2* and b is *3* whereas `(a^2)*b` yields *12*.

The parenthesis of `(a^b)^c` cannot be omitted because a^b^c is equivalent to a^(b^c). For example, `(a^b)^c` yields *1* if a is *-1*, b is *3* and c is *2* whereas a^(b^c) yields *-1*.

The parenthesis of `(a or b) and c` cannot be omitted because `or` and `and` have different operator precedences and **S-3.3** prohibits mixing of different operator precedences without explicit parenthesizing by using precedence-overriding non-terminals (like *parenthesized-expressions*).

The value assigned to a in `a := -b^2;` always will be positive whereas for `a := 0 - (b^2);` and `a := -1*(b^2);` it always will be negative. Note that `a := -(b^2);` is an invalid expression since the unary `-`-operation is only supported for *references*.

The strict left-to-right associativity of `a <> b <> c` is important for the expression to have a well-defined — i.e., unique — type. For example, if a and b are of type *Integer* and c of type *Boolean*, `(a <> b) <> c` is type-correct whereas `a <> (b <> c)` is illegal.

S-3.4 (type of operations / Type-analysis): Except for `^`-operations, the arguments of an operation must be equally typed. The arguments of *arithmetic-operators* and *relational-operators*, except `==` and `<>`-operators, must be of type *Integer* or *Real*. The arguments of `/`-operations must be of type *Real*. The arguments of *logical-operators* must be of type *Boolean*. The argument of unary `-`-operations must be of type *Real* or *Integer*; the argument of unary `not`-operations must be of type *Boolean*.

Except for `^`-operations and operations with an operator that is a *relational-operator*, the type of an operation is the type of its arguments. The type of `^`-operations is *Real*; the type of operations with an operator that is a *relational-operator* is *Boolean*.

R-1 (Real-type restriction of /-operation; absence of %-operator / Type-analysis): Division of *Integer* values via the `/`-operator is prohibited since there exists no common mathematical or formal language interpretation of such. Often, integer division is target-

specific. For example in C89, integer division with a negative operand has an implementation-defined behavior, whereas in C99 it corresponds to `divisionTowardsZero`. Programming languages differ on their interpretation of integer division and remainder; particularly regarding the latter a plethora of `mod`-function and `%`-operator interpretations exist. The problem is related to the implicitly applied rounding of integer divisions; GALEC is explicit however and provides a systematic scheme of rounding-related builtin functions (cf. **S-2.9**), like `roundUp`, `divisionUp` and `remainderUp` or `roundTowardsZero`, `divisionTowardsZero` and `remainderTowardsZero` for the rounding strategies to round plus and minus half up or towards zero respectively. Instead of some implicit-rounding `/` and `%`-operator on *Integer* values, the desired builtin function and explicit type casts via `real` and `integer` can be used.

R-2 (equality-tests of Real-typed variables / Coding recommendation): The support of *Real* arguments for `==` and `<>`-operations is mostly intended for tests against magic literal numbers like `0.0` or `1.0`, for example to enable *if-statements* protecting against division by zero. Since tests for exact equality of *Real* variables are otherwise error-prone, tools are advised to warn about such although they are not prohibited. Equality tests of *Real*-typed variables cannot be prohibited easily anyway, since such can be encoded in a plethora of different schemes using negation, other *relational-operators* and temporary variables, like:

```
1 b1 := r1 > r2;
2 b2 := r1 < r2;
3 if not(b1 or b2) /* r1 == r2 */ then
```

L-1 (target-specific ^-operation implementation / Runtime-semantic): The `^`-operator provides all kind of type-combinations for its base and exponent arguments. It is not restricted to just *Real* arguments, because specialized — and therefore more efficient — implementations for different base and exponent type combinations exist, often provided as target-specific hardware operations. By enabling, for example, both, *Real* and *Integer*-typed exponents, Production Code tools can choose the most efficient implementation available on a target platform. The return type is always *Real* however, since overflows in case of only *Integer* arguments are likely if the result would be implicitly forced to fit into the *Integer* representation of a target platform.

S-3.TODO (type and dimensionality of constant-scalar-integer-expressions / Type-analysis, dimensionality-analysis): The type and dimensionality of *constant-scalar-integer-expressions* are the type and dimensionality of their first child; they must be *Integer* and scalar respectively.

S-3.TODO (well-defined stateful function calls / Side-effect-analysis): Expressions containing a *function-call C* referring to a stateful function must not contain *function-calls* or *state-references* that are siblings of *C*. *If-expressions* must not contain *function-calls* referring to a stateful function.

R-1 (isolated side-effects of stateful function calls and parallel computing / Runtime-semantic): The restrictions on expressions regarding the combination of stateful *function-calls* with other *function-calls* and state variable references promote the isolation of side-effects into separate statements, such that complex expressions can be understood without consideration of control-state changes triggered throughout their evaluation. Moreover, the evaluation order of *function-call* arguments is undefined, such that the runtime-semantic of multiple argument-expressions with side-effects would become undefined without **S-3.TODO**; likewise, the runtime-semantic of *multi-dimension-constructors* containing multiple stateful *function-calls* would become ambiguous. And although the evaluation order of *binary-operations* is strict (cf. **S-3.3**), limiting side-effects in such highly improves clarity.

S-3.TODO also enables the generation of Production Code that computes different parts of a single expression in parallel, without requiring mutual exclusion or memory copying. For example, multiple *function-calls* and the evaluation of *function-call* arguments can be parallelized without the risk of race conditions.

The even more stringent restrictions on *if-expressions* are required to ensure their branches can be executed in parallel and afterwards the actual result selected (assuming evaluating the condition or other siblings of the *if-expression* requires significant time such that the early execution of branches in parallel is worthwhile); special care to only set the error signals of the actually succeeding branch is required however (cf. Section Section 3.2.5). The main motivation is however, that side-effects of expressions, if any at all, are defined regardless of actual control-flow. If an expression calls a stateful function, that very function will always be executed, regardless which branches of contained *if-expressions* are actually executed (because, although stateful *function-calls* can be part of expressions, they never will be part of *if-expressions* due to **S-3.TODO**). Conditional evaluation of stateful *function-calls* must be isolated in *if-statements* instead.

Note, that calling stateful functions cannot be completely prohibited within expressions; otherwise return values of such could not be used like in `(a, b) := m();`, `v := -m();`, `self.A := 2 * m(f(B), f(C));` or `self.A := m(m(self.A));` where `m` and `f` refer to a stateful and stateless function respectively. All of these statements are valid and their runtime-semantic is well-defined.

E-1: The following expression examples illustrate the restrictions on stateful *function-calls* within expressions. Illegal applications are marked by a respective comment; for valid expressions the parts that can be evaluated in parallel are marked. All `m_α` are stateful functions whereas `f_α` are stateless functions for any $α∈ℕ+$.

Expression 1:

```
1 f_1(
2     f_2(m_1()), /* Illegal: Sibling state variable reference. */
3     f_3(self.v))
```

Expression 1:

```
1 f_1(
2     f_2(m_1()), /* Illegal: Sibling stateful function-call. */
3     f_3(m_2())) /* Illegal: Sibling stateful function-call. */
```

Expression 1:

```
1 (2 * self.v) + m_1() /* Illegal: Sibling state variable reference. */
```

Expression 1:

```
1 m_1() /* Illegal: Sibling stateful function-call. */
2  +
3 m_2() /* Illegal: Sibling stateful function-call. */
```

Expression 2:

```
1 (if 0 < m_1() /* Illegal: Stateful function-call within if-expression. */
2     then f_1(self.v_1)
3     else 1.0)
```

Expression 3:

```
1 (if f_1(self.v_1)
2     then m_1() /* Illegal: Stateful function-call within if-expression. */
3     else self.v_1)
```

Expression 3:

```
 1 (
 2 if
 3 /* Parallelizable: Part of separately-evaluable sub-expression-set α: */
 4     f_1(A)
 5 then
 6 /* Parallelizable: Part of separately-evaluable sub-expression-set α: */
 7     f_2(A * B) * C
 8 else
 9 /* Parallelizable: Part of separately-evaluable sub-expression-set α: */
10     f_2(A - B) * C
11 )
12
```

```
13    + /* NOT parallelizable (cf. S-3.3:L-1). */
14
15 /* Parallelizable: Part of separately-evaluable sub-expression-set α: */
16 f_3(
17    D / (E - F),   /* Parallelizable: Part of separately-evaluable sub-expression-
   set β. */
18    E * F,         /* Parallelizable: Part of separately-evaluable sub-expression-
   set β. */
19    (
20        f_2(E) /* Parallelizable: Part of separately-evaluable sub-expression-set
   γ. */
21          *        /* Parallelizable: Part of separately-evaluable sub-expression-
   set β. */
22        f_2(F) /* Parallelizable: Part of separately-evaluable sub-expression-set
   γ. */
23    )
24 )
25
26    + /* NOT parallelizable (cf. S-3.3:L-1). */
27
28 /* Parallelizable: Part of separately-evaluable sub-expression-set α: */
29 (
30    A^3
31 )
```

To understand why expressions marked to be illegal are prohibited, consider that each of the following three can depend on control-state changes performed by previous stateful *function-calls*:

1. the value a *reference*, that refers to a state variable, will yield

2. the values a *function-call* (stateful or stateless) will return

3. the control-state changes a stateful *function-call* will perform

For example, given

```
 1 function f_1
 2    input Real x_1;
 3    input Real x_2;
 4    input Real x_3;
 5    output Real y;
 6 algorithm
 7    y := x_3 * (x_1 * self.a + x_2 * self.b);
 8 end f_1;
 9
10 method m_1
11    output Real y;
12 algorithm
13    self.a := self.a + 1;
14    self.b := self.a;
15    y := self.a;
16 end m_1;
17
18 method m_2
19    output Real y;
```

```
20 algorithm
21    self.b := 2 * self.b;
22    self.a := self.b;
23    y := self.b;
24 end m_2;
```

all three cases are demonstrated by the illegal expression `f_1(self.a, f_1(m_1(), self.b, m_1()), m_2())`. The argument values passed to each `f_1` call depend on when the `m_1` and `m_2` calls are executed, i.e., the order of argument evaluation. There exist *3! * 3! = 36* results if `self.a` and `self.b` are of type *Integer* and the evaluation of inner `f_1` call arguments is not mixed with outer call argument evaluation; if both can be mixed, *5! = 120* results exist (note, that mixing the evaluation of inner and outer arguments is not prohibited by **S-3.TODO (eager evaluation and pass-by-value)** although usually — except for result caching — inefficient).

S-3.TODO (function lookup / Name-analysis): Let f_c be a *function-call*. There must exist a function f_d named like the first child of f_c; according to **S-2.5**, f_d must be unique. We say f_c refers to f_d.

S-3.TODO (type of function calls used in expressions / Type-analysis): *Function-calls* part of expressions must refer to functions of output-arity *1*; their type is the type of the first output of the function they refer to.

R-1: According to **S-3.1**, the right-hand of *multi-assignments* (their *function-call* child) is not an expression; likewise, *function-calls* whose parent is a *statement* are not expressions. The rationale for either is, that expressions have a unique type and dimensionality characterising their potential values. The *function-call* child of a *multi-assignment* can refer to a function with several outputs however, each with an individual type and dimensionality; and for *function-calls* not part of an assignment outputs don't matter.

E-1: Let *p_1* and *p_2* be procedures of input-arities *1* and *2* respectively and let *f_1* and *f_2* be functions of input-arity *0* and output-arities *1* and *2* respectively. The *statements* `p_1(f_1());`, `p_2(f_1(), f_1());`, `f_1();`, `f_2();`, `( v ) := f_1();` and `( v_1, v_2 ) := f_2();` are valid, whereas `p_1(p_1(f_1()));` and `p_2(f_2());` are illegal.

Statements: State changes (intra-functional flowchart)

G-TODO.TODO — G-TODO.TODO (TODO)

```
 1 (* references *)
 2 reference = local-reference | state-reference ;
 3
 4 local-reference = name, [ computed-dimensions ] ;
 5
 6 state-reference =
 7     "self",
 8     ".",
 9     name,
10     [ computed-dimensions ],
11     { ".", name, [ computed-dimensions ] } ;
12
13 computed-dimensions =
14     "[",
15     constant-scalar-integer-expression,
16     { ",", constant-scalar-integer-expression },
17     "]" ;
18
19 (* statements *)
20 statement =
21     (
22           limit-statement
23         | function-call
24         | single-assignment
25         | multi-assignment
26         | if-statement
27         | for-loop
28     ),
29     ";" ;
30
31 limit-statement =
32     "limit",
33     ( "self" | reference ),
34     { ",", ( "self" | reference ) } ;
35
36 single-assignment = reference, ":=", expression ;
37
38 multi-assignment =
39     "(",
40     [ reference, { ",", reference } ],
41     ")",
42     ":=",
43     function-call ;
44
45 if-statement =
46     "if",
47     ( expression | error-signal-check ),
48     "then",
49     { statement },
50     { "elseif", ( expression | error-signal-check ), "then", { statement } },
51     [ "else", { statement } ],
52     "end",
53     "if" ;
54
55 error-signal-check =
56     "signal",
57     [ identifier ],
```

```
58    [
59        [ "not" ],
60        "in",
61        identifier,
62        { ",", identifier  }
63    ],
64    [ "or", expression ] ;
65
66 for-loop = "for", bounded-iteration, "loop", { statement }, "end", "for" ;
67
68 bounded-iteration =
69    [ loop-iterator-declaration, "in" ],
70    start-bound,
71    [ ":", iteration-step-size ],
72    ":",
73    termination-bound ;
74
75 loop-iterator-declaration = name ;
76
77 start-bound = constant-scalar-integer-expression ;
78
79 iteration-step-size = constant-scalar-integer-expression ;
80
81 termination-bound = constant-scalar-integer-expression ;
```

S-TODO.TODO (type of references / Type-analysis): The type of a *reference* is the type of the entity it refers to.

References referring to a state component must be the third child of a *dimension-query* or part of a *limit-statement*.

> **L-1 (limited application of state components / Runtime-semantic):** The semantic rule indirectly prohibits any runtime interaction with state components — like passing them as function or operation arguments or assignment of such — except to query their dimensionality by means of *dimension-queries* or limiting all their variables by means of *limit-statements*. In opposite to variables, state components as such do not exist at runtime; they have no runtime values — they are valueless. Only variables have a value that can be used in expressions or changed via assignment. As a consequence, Production Code generators do not have to preserve state components and are free to choose whichever runtime representation they consider most suitable for their nested entities; they can, for example, map the nested constants of a state-component to read-only memory or constant fold them or pack nested state variables together with other non-nested variables. This is a significant difference to for example C89 struct variables, which have a value that must be stored within a locally coherent piece of memory, a requirement necessary to enable efficient struct value assignment or referencing via pointers (neither exists in GALEC).

> **S-TODO.TODO (left- and right-hand of assignments / Side-effect-analysis, terminology):** *Single-assignments* and *multi-assignments* are called assignment. The first child of an assignment is called its left-hand; the third child its right-hand.

> **S-TODO.TODO (non-writeable control-inputs, input parameters and loop iterators; side-effect-freeness of stateless functions / Side-effect-analysis):** *State-references* contained in the left-hand of an assignment must not refer to control-inputs. *Local-references* contained in the left-hand of an assignment must not refer to input parameters or *loop-iterator-declarations*.
>
> Stateless functions must not contain an assignment whose left-hand contains a *state-reference*; and they must not transitively call stateful functions.

3.2.5. Error handling

GALEC incorporates dedicated language means for systematic, reliable and guaranteed error handling. Three integrated concepts can be distinguished: (1) error signals with enforced signal handling seamlessly incorporated into normal program control-flow, (2) well-defined floating point operations with guaranteed quiet Not-a-Number propagation and (3) variable ranges for guaranteed block saturation. Together, these concepts enable delayed, but ensured error handling avoiding any need to immediately check each and every possible failing operation by means of a plethora of exceptions.

The following sections present these three concepts.

§1: Error Signals

§1.1: *Error-signal-declaration* semantic

An *error-signal-declaration D* of the from

```
1 error-signal-declaration = "signal", identifier, ";" ;
```

is called an error signal. The name of an error signal is the name of its contained *identifier*; its name must be unique within the *block D* is part of.

Let *Predefined* be the following sequence of characters

```
1 signal INVALID_ARGUMENT;
2 signal OVERFLOW;
3 signal NAN;
4 signal SOLVE_LINEAR_EQUATIONS_FAILED;
5 signal NO_SOLUTION_FOUND;
6 signal UNSPECIFIED_ERROR;
```

Predefined implicitly follows the characters matched by the 6th child of *block*; its error signals are called predefined. Any other error signals are called user-defined.

> **R-1 (purpose of pre-defined error signals / Runtime-semantic):** Above specification implies that pre- and user-defined error signals are error signals and can therefore be explicitly signaled and checked by user-code.
>
> The typical application of pre-defined error signals is however not block function internal error handling, but to expose block-interface method errors to the runtime environment (cf. *§1.3* and *§1.6*). The intended meaning of each pre-defined error signal is:
>
> - INVALID_ARGUMENT: Unspecified error in one or more input arguments.
>
> - OVERFLOW: Some computed floating point value is $-\infty$ or $+\infty$.
>
> - NAN: Some computed floating point value is qNaN.
>
> - SOLVE_LINEAR_EQUATIONS_FAILED: Solving a linear equation system via the solveLinearEquations builtin function failed.
>
> - NO_SOLUTION_FOUND: Not used for solveLinearEquations, but for example if an optimizer, special nonlinear solver etc. does not find a solution.
>
> - UNSPECIFIED_ERROR: Error that is not further specified.

§1.2: *Error-signal-statement* semantic

A *error-signal* statement *S* of the form

```
1 error-signal-statement =
2     "signal",
3     identifier,         (* Set of signals set, at least one AND/OR signal-closure propagation
  *)
4     { ",", identifier } ; (* Set of signals set, at least one AND/OR signal-closure propagation
  *)
```

has the following semantic:

1. Each *identifier s* of *S* referring to a signal-closure variable *s* in scope sets all the signals of *s* whenever *S* is executed (cf. *§1.4* for the definition of signal-closure variables).

2. Any other identifier *s* of *S* must refer to an error signal *e*. Whenever *S* is executed, *e* is set.

3. The union of all error signals set by *S* is called the signal-set of *S*.

§1.3: Functional error interface and exposed error signals

A *function-declaration F* of the form

```
1 function-declaration =
2     ( "function" | "method" ),
3     name,
```

```
 4      [ signal-interface ], (* 3rd child defining the signal-set -- i.e, exposed error signals --
    of the function *)
 5      { parameter-declaration },
 6      [ "protected", { local-variable-declaration } ],
 7      "algorithm",
 8      { statement },
 9      "end",
10      name,
11      ";" ;
```

has the following semantic w.r.t. error handling:

1. Let all *identifiers* contained in the 3rd child of *F* form the signal-set *S* of *F*. Each element *s* of *S* must refer to an error signal *e*; each such *e* is called an exposed error signal of *F* and *F* is said to expose *e*.

2. Block-interface functions must not expose user-defined error signals.

3. The signal-set of *F* must be identical to the out-reachable-signals-set an imaginary final statement following the last statement of *F* would have (cf. *§1.5* for the definition of out-reachable-signals-set).

§1.4: *Error-signal-check* semantic

An *error-signal-check* of the form

```
 1 error-signal-check =
 2     "signal",
 3     [ identifier ],              (* Optional signal-closure *)
 4     [
 5         [ "not" ],               (* Optional signal-test-negation *)
 6         "in",
 7         identifier,              (* Set of signals tested, at least one *)
 8         { ",", identifier  },   (* Set of signals tested, at least one *)
 9     ],
10     [ "or", expression ] ;       (* Optional fallback-condition *)
```

has the following semantic:

1. A signal-closure is a scoped variable that captures the current error-state (i.e., all the currently set error signals). Its scope is, similar to loop-iterators, the branch-body of the *error-signal-check* — called error-signal-check-body. A signal-closure must never be assigned to.

2. We define the signal-test-set of an *error-signal-check* as follows (cf. *§1.5* for the definition of in-reachable-signals-set):

   ◦ **At least one signal tested is given:** If, and only if, no signal-test-negation is given, the signal-test-set comprises all signals tested; otherwise, it comprises the signals of the in-reachable-signals-set of the *error-signal-check* minus the set of all signals tested.

   ◦ **No signal tested is given:** The signal-test-set is the in-reachable-signals-set; the *error-signal-check* is called unrestricted.

The signal-test-set must be non-empty and a subset of the in-reachable-signals-set of the *error-*

signal-check.

1. An *error-signal-check* is signal-satisfied, if, and only if, any of the signals of its signal-test-set is set when it is executed.

2. An *error-signal-check* is conditional-satisfied, if, and only if, it is not signal-satisfied and has an optional fallback-condition that is satisfied when the *error-signal-check* is executed.

3. An *error-signal-check* is satisfied if it is signal-satisfied or conditional-satisfied.

4. The error-signal-check-body *B* of an *error-signal-check C* is the executed branch-body of its *if-statement*, if, and only if, *C* is satisfied. In this case, all signals of the signal-test-set of *C* are unset immediately before the execution of *B* but after initializing the signal-closure if any.

§1.5: Error signal propagation semantic: static signal propagation analysis and reachable-signals-set

§1.3 and *§1.4* require that (1) functions only expose error signals that can be signaled within their body but are not checked thereafter for any of their possible control-flows and (2) *error-checks* only check for error signals that can be set according to their preceding control-flow. To enforce these restrictions, a static analysis deciding which error signals could be set at any point of execution is required.

To that end, we define for expressions and statements which additional error signals they can set — their signal-set — and for statements and the branches of *if-statements*, particularly *error-signal-check* branches, which error signals can be set right before their execution and right after — their in-reachable-signals-set and out-reachable-signals-set.

The signal-sets of expressions and statements are:

1. The signal-set of a *function-call* is the referred function's signal-set. The signal-set of any other expression is the union of the signal-sets of its contained *function-calls*.

2. The signal-set of *single-assignments* and *multi-assignments* is the signal-set of their right-hand sides.

3. The signal-set of a *for-loop* is the out-reachable-signals-set of its last statement.

4. The signal-set of an *if-statement* is the union of the out-reachable-signals-sets of the last statements of its branch-bodies.

The in-reachable-signals-sets and out-reachable-signals-sets of statements and the branches of *if-statements* are:

1. The in-reachable-signals-set of the first statement *S* of a function-body is the empty set.

2. The in-reachable-signals-set of the first branch-condition of an *if-statement S* is the in-reachable-signals-set of *S*; for any further branch-condition of *S* it is the out-reachable-signals-set of its preceding branch-condition (important: preceding branch-condition, not branch-body of preceding branch).

3. The in-reachable-signals-set of the branch-body of a branch *B* of an *if-statement* is the out-reachable-signals-set of the branch-condition of *B*.

4. The in-reachable-signals-set of any other statement *S* is the union of the out-reachable-signals-sets of all its preceding statements (according to control-flow).

5. The out-reachable-signals-set of an *error-signal-check* branch-condition is its in-reachable-signals-set minus its signal-test-set, finally unified with the signal-set of its fallback-condition if any. The out-reachable-signals-set for a non *error-signal-check* branch-condition is its in-reachable-signals-set unified with the signal-set of its condition.

6. The out-reachable-signals-set of an *if-statement* is the out-reachable-signals-set of its last branch-condition unified with its signal-set.

7. The out-reachable-signals-set of any other statement is its in-reachable-signals-set unified with its signal-set.

§1.6: Production Code and exposing error signals to the runtime environment

Since block-interface methods can only expose the 6 pre-defined error signals (cf. *§1.3*), a definition of signal-communication with the runtime environment is only required for such. To that end a unique mapping of each pre-defined error signal to a unique bit position within a 32 bit integer value is defined. These mappings are bidirectional, such that all exposed error signals can be returned to the runtime environment encoded in a single 32 bit integer value. The bit positions of the pre-defined error signals are:

- Bit 0: INVALID_ARGUMENT
- Bit 1: OVERFLOW
- Bit 2: NAN
- Bit 3: SOLVE_LINEAR_EQUATIONS_FAILED
- Bit 4: NO_SOLUTION_FOUND
- Bit 5: UNSPECIFIED_ERROR

Bit positions 6 to 15 of the returned error value are reserved for the future if there is need to add further pre-defined error signals in later specification versions; for now these bits must be never set by error values returned to the runtime environment.

To enable easy Production Code generator implementation by encoding all error signals — i.e., pre- and user-defined — in single, uniquely laid out (i.e., uniform bit position accessible) 32 bit integer values, GALEC programs must contain at most 16 user-defined error signals (i.e., 32 - 6 pre-defined - 10 reserved).

> ***R-1 (need for bit-wise pre-defined error signals for Behavioral Model representation based testing):*** It is important to enable reference testing of the exposed error signals of block-interface methods. Besides the common case to ensure no errors are encounted, also testing for expected errors — if for example a block is misused outside its required operational range — are deliberately supported by eFMI Behavioral Model representations (cf. §1.1 and §4 of Section 4.2). Such tests require a reference trajectory for the expected error signals, whose values are in turn compared to the actual error signal values exposed by tested production or binary code. Hence, a well-defined encoding of exposed error signal values is required. To that end, exposed error signal values are restricted to pre-defined error signals encoded according to *§1.6* (cf. also *§1.1:R-1*).

Examples

E-1: The following example sketches a typical mixed-mode coding style, where some error cases are avoided in the first place by special operation modes of the controller and others are treated after something failed by testing for respective error signals:

```
 1  /*
 2      Safe common control-code, potentially selecting or deselecting special
 3      modes of operation:
 4  */
 5  ...
 6  v := f(A); /* Assume f may signal the error f_ERROR. */
 7  ...
 8  if signal in f_ERROR or not(check(v)) then
 9      /*
10          Error-handling path if f(A) signaled an f_ERROR or
11          returned a v not satisfying some check:
12      */
13      ...
14  elseif self.operation_mode == 1 then
15      /*
16          Safe control-code for some special operation mode:
17      */
18      ...
19  elseif self.operation_mode == 2 then
20      /*
21          Safe control-code for some special operation mode:
22      */
23      ...
24  else
25      /*
26          Control-code for normal mode of operations:
27      */
28      ...
29      x := solveLinearEquations(A, b * v);
30      ...
31      if signal in SOLVE_LINEAR_EQUATIONS_FAILED then
32          /*
33              Handle the special case that the system of linear equations
34              has no solution:
35          */
36          ...
37      elseif signal then
38          /*
39              Handle any other unexpected error of the NORMAL operation mode:
40          */
41          ...
42      end if;
43  end if;
44
45  if signal s then
46      /*
47          The common control-code or the special modes of operation that are
48          supposed to be safe missed some error case or introduced
49          errors themselves. We now can set the control-outputs and state
50          variables to some reasonable default values, e.g., reinitialize
```

```
51        the controller, and propagate the unexpected error signals to the runtime
    environment:
52    */
53    ...
54    signal s;
55 end if;
```

E-2: The following example summarises all possible combinations of error signaling and checking:

```
 1 method MyMethod
 2 /*
 3     (1) Signal interface of functions (signals exposed to callees):
 4 */
 5 signals invalid_gear_switch, to_high_velocity;
 6 protected
 7 algorithm
 8    ...
 9    /*
10        (2) Universal signal checks, catching and un-setting all signals set:
11    */
12    if signal then
13        ...
14    end if;
15    ...
16    /*
17        (3) Specialized signal checks, catching and un-setting
18            all signals within a specific set:
19    */
20    if signal in error1, error2 then
21        ...
22    end if;
23    ...
24    /*
25        (4) Restricted universal signal checks, catching any signal that is
26            not within a certain set:
27    */
28    if signal not in invalid_gear_switch, to_high_velocity then
29        ...
30    end if;
31    ...
32    /*
33        (3) Checks with signal variables capturing all signals
34            set at the check point:
35    */
36    if signal s then
37        ...
38        /*
39            (4) Propagation of signal variable, i.e., reset all
40                captured signals (they have been unset by the previous check):
41        */
42        signal s;
43        ...
44    else
```

```
45          ...
46      end if;
47      ...
48      if ... then
49          ...
50          /*
51              Explicit setting of signals, i.e., signaling of errors:
52          */
53          signal invalid_gear_switch, to_high_velocity;
54          ...
55      end if;
56      ...
57      /*
58          (*) And all kind of combinations of the above
59              (signals to check with signal variables, signal
60              propagation and explicit signaling):
61      */
62      if signal s in f1_error, f2_error or condition1 then
63          ...
64          signal s, invalid_gear_switch;
65          ...
66      elseif signal s not in invalid_gear_switch, to_high_velocity or condition2 then
67          ...
68          signal s;
69          ...
70      end if;
71      ...
72      /*
73          Catch all signals not exposed according to the function's interface:
74      */
75      if signal not in invalid_gear_switch, to_high_velocity then
76      end if;
77 end MyMethod;
```

E-3: The following example shows typical violations of error signal propagation, demonstrating the advantages of a strict static signal-propagation analysis for code hardening:

```
1 function f
2 /*
3      Invalid: Exposed error signal can never be signaled.
4
5      Error1 is always handled on all possible control-flow
6      paths within the function.
7
8      Invalid: Uncatched error signal not exposed.
9
10     There exists a control-flow path within the function
11     on which Error2 is signaled, but never handled.
12 */
13 signals UNSPECIFIED_ERROR;
14     input Real i;
15     output Real o;
16 protected
17 algorithm
```

```
18      if i > 100.0 then
19          signal Error1;
20      elseif i > 200.0 then
21          signal Error2;
22      end if;
23      o := 2.0 * i;
24      if signal in Error1 or o > 350.0 then
25          o := 350.0;
26      end if;
27 end f;
28
29 method DoStep
30 /*
31     Invalid: Uncatched error signal not exposed.
32
33     There exists a control-flow path within the function
34     on which UNSPECIFIED_ERROR is signaled, but never handled.
35
36     Invalid: Block-interface method exposes not
37             pre-defined error signal.
38
39     Error1 is not a pre-defined error signal.
40 */
41 signals Error1;
42 protected
43 algorithm
44     ...
45     f(1.0);
46     ...
47     if signal s then
48         ...
49         s := Error1; /* Invalid: Signal-closures must not be assigned to. */
50     elseif signal in Error1 then
51         /*
52             Invalid: Signal-test-sets must be non-empty.
53
54             The preceding branch already handles all error signals since it
55             is an unrestricted error-signal-check.
56         */
57     end if;
58
59     signal Error1;
60
61     if signal in Error1, UNSPECIFIED_ERROR then
62         /*
63             Invalid: The signal-test-set is not a subset of the
64                     in-reachable-signals-set.
65
66             UNSPECIFIED_ERROR can never be set at this point.
67         */
68         ...
69         signal UNSPECIFIED_ERROR;
70     elseif signal in UNSPECIFIED_ERROR then
71         /*
72             Invalid: The signal test-set is not a subset of the
73                     in-reachable-signals-set.
74
75             UNSPECIFIED_ERROR can never be set at this point.
```

```
76              The signal-set of the error-signal-check-body of the
77              preceding branch cannot be handled by this branch;
78              it requires handling in a completely separate if-statement.
79          */
80      end if;
81
82      signal Error1;
83      if signal in Error1 then
84      end if;
85      if signal in Error1 then
86          /*
87              Invalid: Signal-test-sets must be non-empty.
88
89              The preceding if-statement already implicitly unsets
90              Error1 when its single error-signal-check is satisfied.
91          */
92      end if;
93
94      signal Error1;
95 end DoStep;
96
97 method Startup
98 protected
99 algorithm
100     /* signal in E; */
101     /*
102         The following if-statement is invalid even if the
103         previous line is uncommented.
104     */
105     if signal not in E then /* Invalid: Signal-test-sets must be non-empty. */
106     end if;
107 end Startup;
```

Example 4: The following function fragment investigates interesting corner-cases of error signal propagation. It is well-suited to exercise the formal definitions of signal-set, in-reachable-signals-set and out-reachable-signals-set of if-statements. The left-out code hooks denoted by ... are assumed to be arbitrary code not setting or checking error signals.

```
1 function f
2     signals f_Error;
3     output Boolean b;
4 protected
5 algorithm
6     b := true;
7     signal f_Error;
8 end f;
9
10 method DoStep
11     ...
12 algorithm
13     ...
14     if signal then /* Unset all error signals. */
15     end if;
16     signal in TestDefinitions1, TestDefinitions2;
```

```
17    if signal in TestDefinitions1 then
18        ...
19        signal TestDefinitions3;
20        ...
21    elseif signal in TestDefinitions2 then
22        ...
23        if signal TestDefinitions3 then
24            ...
25        end if;
26        ...
27    elseif signal in TestDefinitions3 then
28        ...
29    end if;
30    /*
31        At this point still TestDefinitions2 and TestDefinitions3 WILL be
32        set because only the first branch was tested, its test signal-satisfied,
33        the tested signal TestDefinitions1 unset and its body executed.
34    */
35    ...
36    if signal then /* Unset all error signals. */
37    end if;
38    signal TestDefinitions1, TestDefinitions2;
39    if signal in TestDefinitions1 then
40        ...
41        if signal in TestDefinitions2 then
42            ...
43        end if;
44        ...
45    end if;
46    /* At this point no error signals WILL be set. */
47    ...
48    /*
49        Assume for the following code an execution where NotSetSignal
50        is not set:
51    */
52    if signal not in NotSetSignal then /* Unset all error signals except NotSetSignal. */
53    end if;
54    i := 2;
55    if signal in NotSetSignal or f() /* Cf. definition of f above! */ then
56        i := 2 * i;
57    elseif signal in f_Error then
58        i := 2 * i;
59        signal f_Error;
60    /*
61    The following branch would be invalid, because f_Error can never be set when it is tested:
62    elseif signal in f_Error then
63        i := 2 * i;
64    */
65    end if;
66    /* At this point i WILL be 8 and f_Error set. */
67 end DoStep;
```

§2: -∞, +∞ and quiet Not-a-Number propagation

GALEC assumes that the target system of the generated production code is compliant to IEEE Standard 754-2008. Even if GALEC code is as much as possible target independent, there are corner

cases in which the properties of the target system need to be taken into account in GALEC. If a target system is not fully compliant to IEEE 754-2019, it should still be possible to map GALEC code to such a target, since only a small subset of IEEE 754-2019 is used and/or potential deviations in corner cases might still be acceptable *[(for example, if a processor does not support -∞ or +∞ handling, but saturates automatically to the largest/smallest representable floating point number)]*. Note, in the following, IEEE 754 shall always mean IEEE 754-2019. Deviations to this standard are explicitly marked.

The language assumes, following IEEE 754 section 6, that exception handling of the processor is configured so that an overflow of Real numbers is handled automatically by the processor for all language operators without generating exceptions by mapping negative and positive overflows to -∞ and +∞ respectively (e.g. `2.0 < 1.0 / 0.0` is `true`). With built-in function `isInfinite(r)` it can be inquired whether a Real variable `r` is -∞ or +∞ (e.g. `isInfinite(1.0 / 0.0)` returns `true`).

The language also assumes that IEEE 754 exception handling of the processor is always configured to never generate an exception in case of underflow of Real numbers (so deviating from the default exception handling of IEEE 754, section 7.5).

If the result of a mathematical operation on Real numbers is mathematically undefined (for example `lg(-1.0)` or `0.0 / 0.0`), then the standard operators of the language return quiet Not-a-Number (qNaN) as defined by IEEE 754, section 7.5. It is assumed that the processor is configured so that qNaN values are automatically propagated through all operations without generating exceptions (hence *quiet* Not-a-Number). With built-in function `isNaN(r)` it can be inquired whether a `Real` variable `r` has qNaN as value or not.

All relational operators (`<`, `>`, `<=`, `>=`, `==`, `<>`) trigger error signal NAN if one of their operands is qNaN. In such a case the operator returns `false`. Conceptually, every relational operator a ⊕ b is mapped to a built-in function call `f_⊕(a, b)` with `f_⊕` defined as:

```
 1 function f_⊕
 2    signals NAN;
 3    input Real a;
 4    input Real b;
 5    output Boolean y;
 6 algorithm
 7    if isNaN(a) or isNaN(b) then
 8        signal NAN;
 9        y := false;
10    else
11        y := a ⊕ b;
12    end if;
13 end f_⊕;
```

[In C this function can be implemented efficiently for example as the expression `(isNaN(a) || isNaN(b) ? (error_signal |= Bitmask setting NAN, 0) : a ⊕ b)`.]

All built-in functions (see section Section 3.2.6) that can have qNaN input arguments and are not able to propagate qNaN because the output argument(s) are not of type `Real` trigger the NAN error signal.

[Note, potential issues as sketched in Agner 2019 [https://www.agner.org/optimize/nan_propagation.pdf] are

not critical because relational operators and builtin functions trigger the NAN error signal if a qNaN value cannot be propagated.]

For some built-in functions that can return qNaN, also companion built-in functions are provided, that do not return qNaN, provided none of the input arguments is qNaN. These functions start with the prefix `safe_` and achieve this behavior (conceptually) by automatic limitation of their input argument(s).

§3: Variable Ranges, explicit and implicit limitation and block saturation

All variables can be declared with range attributes `min` and/or `max`; variables with range attributes are called ranged.

Ranged variables are limited to their defined range at a particular point of execution by means of *limit-statements*. If a variable v is ranged with lower bound $\perp$ and upper bound $\top$, then the statement `limit v;` is equivalent to `v := (if v < ⊥ then ⊥ elseif v > ⊤ then ⊤ else v);`. If v has only a lower bound $\perp$, `limit v` is equivalent to `v := (if v < ⊥ then ⊥ else v);`. If v has only an upper bound $\top$, `limit v;` is equivalent to `v := (if v > ⊤ then ⊤ else v);`. Limiting a non-ranged entity has no effect.

[Above definition implies that limitation on qNaN values has no effect (the variable's value remains qNaN).]

`limit` can also be used to limit all state variables according to their ranges (using keyword `self`), or all nested state variables of a certain state component (by referring to that very state component):

```
1 limit self; /* Limits all ranged state variables. */
2 limit c; /* Assume c refers to a state component: limits all nested state variables of c. */
```

A single limit statement can limit a set of entities. For example,

```
1 limit self.c.d.vc, self.v, self.c, l;
```

limits the variable `self.c.d.vc` (assuming `self.c.d` refers to a state component and d is one of its variables), the state variable `self.v` (assuming `self.v` refers to a state variable), all nested variables of the state component `self.c` (assuming `self.c` refers to a state component) and the local variable `l`.

Every block-interface method implicitly limits all state entities whenever the method is entered and when it returns, except `Startup()`, which only limits on returning. The implicit semantic is:

```
1 method Startup
2 protected
3     ...
4 algorithm
5     ...
6     /* initialize stuff */
7     ...
8     limit self; /* Implicit by semantic of language. */
```

```
 9  end Startup;
10
11  method DoStep
12  protected
13    ...
14  algorithm
15    limit self; /* Implicit by semantic of language. */
16    ...
17    /* compute stuff */
18    ...
19    limit self; /* Implicit by semantic of language. */
20  end DoStep;
21
22  method Recalibrate
23  protected
24    ...
25  algorithm
26    limit self; /* Implicit by semantic of language. */
27    ...
28    /* compute stuff */
29    ...
30    limit self; /* Implicit by semantic of language. */
31  end Recalibrate;
```

Every function implicitly limits its inputs whenever the function is entered and its outputs when it returns.

[Implicit limitation at the very beginning and end of block-interface methods means, that from the perspective of the runtime environment ranged state variables are effectively saturated at their defined ranges; the block as such is saturated and guarantees operation within its limits (except for state variables with qNaN values that need special error handling).

Production Code generators are free to optimize and minimize limitation of variables. For example, limitation of constants, tunable parameters and dependent parameters will never be required in DoStep(), since such cannot be assigned new values and their limitation is already performed in Startup() and Recalibrate() respectively. Limitation of inputs is only needed at the very beginning of DoStep() code, because inputs are not changed afterwards. Limitation of outputs is only needed at the end of the DoStep() code. Limitation of states needs to be performed only at the end of Startup() and the end of DoStep(), because the states are just passed between DoStep() calls and then it is guaranteed that a state that is limited at the end of the previous DoStep() call remains limited at the very beginning of the next DoStep() call. Furthermore, interval arithmetic analyses can be used to conclude that a variable will never be outside of its valid range, such that limitation code for it can be avoided.

The rationale why limitation is not implicitly performed on every assignment to a ranged variable (i.e., why GALEC has no strict saturation arithmetic) is, that numerical algorithms and particularly integration typically fail if values are not continuous over time. For example, an integration algorithm such as a Runge-Kutta method of order 4 may not work as expected, if states are limited during one step because the smoothness requirements of the integration method are violated. Furthermore, limitations in the middle of computations often inadvertently break algebraic characteristics like distributivity and commutativity that are essential for symbolic processing and optimization. These pitfalls of limitation are however not violated by the implicit limitations at the very start and end of

block-interface methods; the block as such — its interface — is saturated from the perspective of the runtime environment. Throughout the execution of a block-interface method however, variables may very-well get values assigned outside of their defined ranges.]

§4: Error Handling Recommendations

In practice it is typically required that all control-outputs are guaranteed to never be qNaN and always be within their defined ranges. To that end, the following actions are recommended:

- Provide min/max values for state variables, particularly control-inputs, -outputs and tunable parameters. Implicit limitation will guarantee, that the state variables are in their defined ranges when a block-interface method returns, or the variable values are qNaN.

- Before leaving `DoStep()`, check that none of the control-outputs is qNaN **and** that the error signal is not NAN. If one of these conditions does not hold, take appropriate actions, for example restore the state from the previous sample instant, compute the control-outputs with a backup algorithm (e.g. P-controller) that does not produce qNaN values, or provide a default control-output, e.g. zero. In any case, the returned outputs should never be qNaN.

- Use the `safe_`⊕ builtin functions (see below) if this is possible, in order that qNaN values are not generated.

- Often problematic is the `/`-operator. A general approach to handle division in a meaningful way for all possible circumstances seems impossible. However, in many cases the time-varying denominator is guaranteed to not change sign; examples are: dividing by density, mass fraction, gear efficiency or slip. In such cases, the built-in operator `safe_posdiv(num, den, eps)` should be used that provides a meaningful approximation of `num / den` without generating qNaN values, if it is guaranteed that `den >= 0`.

3.2.6. Built-in Functions

In this section the built-in functions are defined. If the built-in function is also defined in IEEE 754-2019, the semantic of the built-in function is according to this standard.

Any function that has `Real` input and `Real` output arguments can usually return qNaN, because an input argument might be qNaN that is typically propagated to one or more outputs. Whenever a function can return qNaN (either because it is generated inside the function or a qNaN input can be propagated to an output), this is explicitly mentioned and also in which situation this occurs. For many built-in functions ⊕ that can generate qNaN, there is also a function `safe_`⊕ that approximates ⊕ so that no qNaN is generated, in case this approximation is useful (but of course such a function can still return qNaN if the input is qNaN).
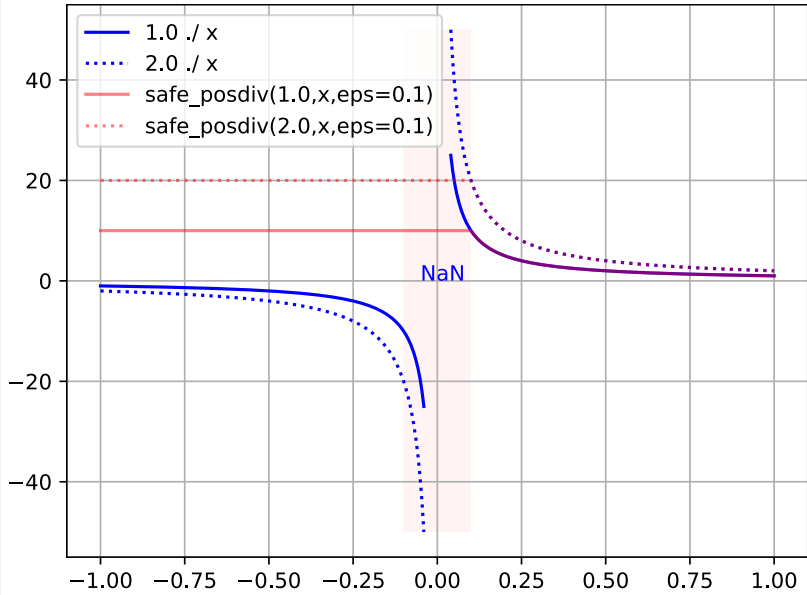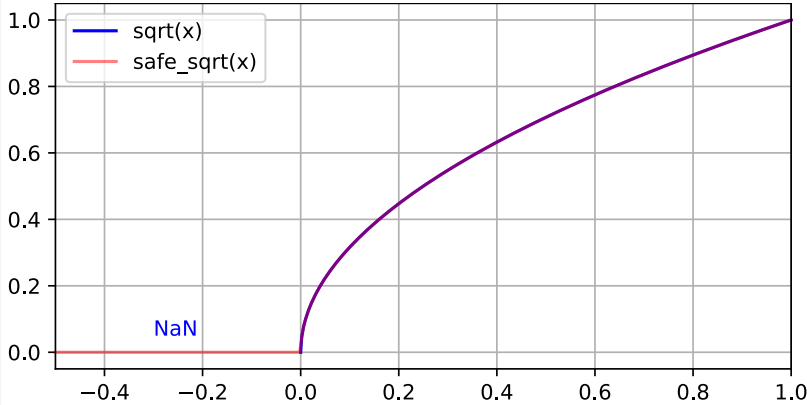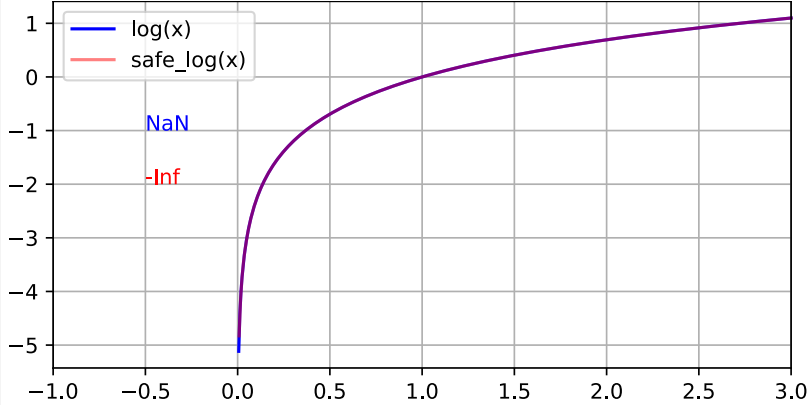
A built-in function only returns an error signal if explicitly mentioned in its definition below; most builtin functions do not signal any errors and instead rely on qNaN propagation.
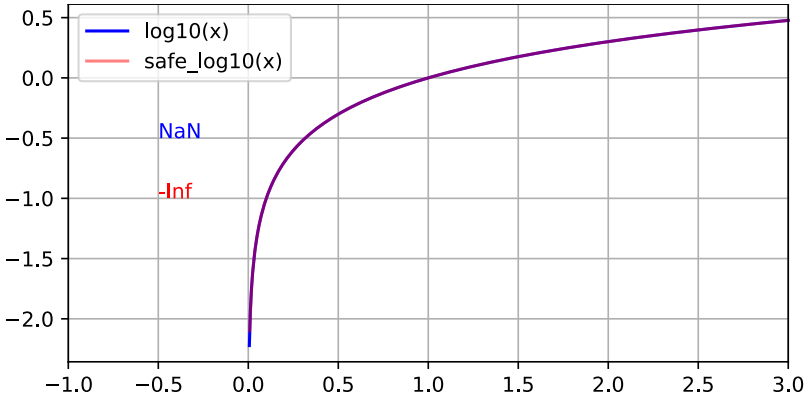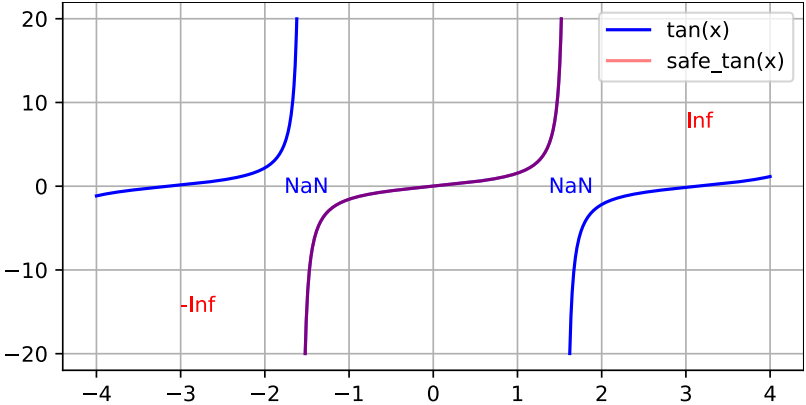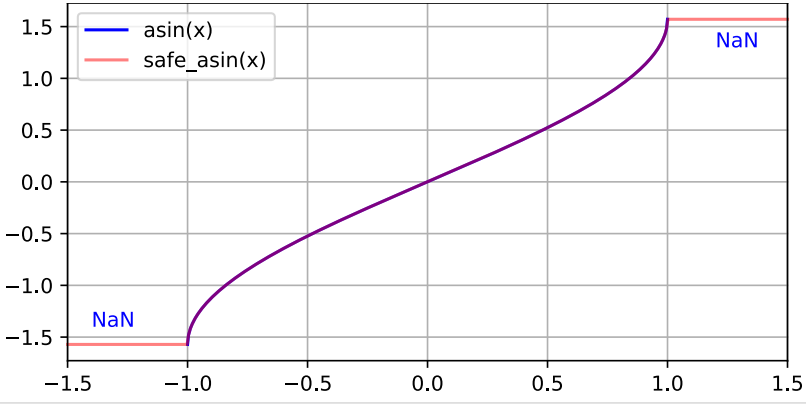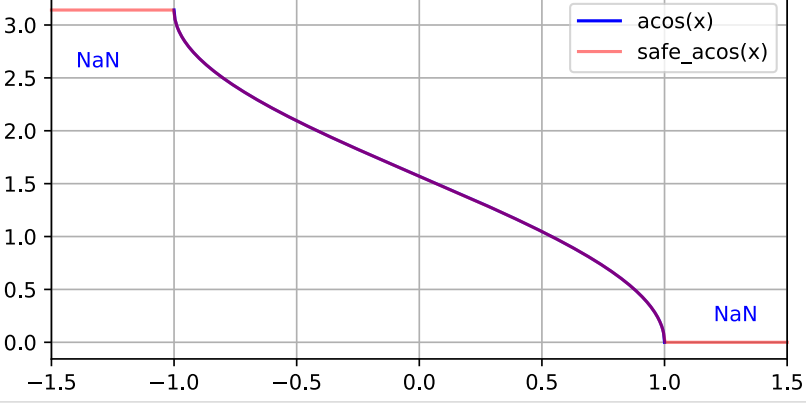
Overview

In the following table, an overview of the built-in functions is given (the follow-up sub-section contains the precise definition of the built-in functions):

Function-Name	Description
Properties of Integer	
`minInteger()`	Target-specific smallest Integer.
`maxInteger()`	Target-specific largest Integer.
Properties of Real	
`minReal()`	Target-specific smallest Real `r <> minusInfinite()`.
`maxReal()`	Target-specific largest Real `r <> plusInfinite()`.
`r := posMinReal()`	Target-specific smallest Real `r > 0.0`.
`r := epsReal()`	Target-specific largest Real `r > 0.0` such that `1.0 + r == 1.0`.
`nan()`	Target-specific quiet not-a-number representation (qNaN).
`isNaN(x)`	`true` if `x` is the target-specific qNaN representation; otherwise `false`.
`minusInfinite()`	Target-specific -∞ representation.
`plusInfinite()`	Target-specific +∞ representation.
`isInfinite(x)`	`true` if `x` is -∞ or +∞; otherwise `false`.
`isFinite(x)`	`true` if `x` is finite (neither -∞ nor +∞ nor qNaN); otherwise `false`.
Multi-dimensional properties of Real	
`hasNaN1D(x)`	`true` if at least one element of vector `x` is qNaN; otherwise `false`.
`hasNaN2D(x)`	`true` if at least one element of matrix `x` is qNaN; otherwise `false`.
Numeric type conversions	
`real(i)`	Convert Integer `i` to Real.
`integer(r)`	Convert Real `r` to Integer by truncation (`roundTowardsZero(r)`). Signals `NAN` if `r` is qNaN in which case `0` is returned. Signals `OVERFLOW` if `r` can not be represented as Integer, in which case `0` is returned.
Direct Real rounding	
`roundDown(r)`	Round `r` towards -∞ (also known as floor). Returns qNaN if `r` is qNaN.
`roundUp(r)`	Round `r` towards +∞ (also known as ceil). Returns qNaN if `r` is qNaN.
Nearest Real rounding (using a tie-breaking rule)	
`roundHalfToEven(r)`	Also known as convergent rounding, statistician's rounding, Dutch rounding. Returns qNaN if `r` is qNaN.
Division of Integers using rounding	
`divisionTowardsZero(i1, i2)`	Divide `i1` by `i2`, rounding the result towards zero. Same as `div(i1, i2)` in C99.

Function-Name	Description
Remainder of Integers using rounding	
`remainderTowardsZero( i1, i2)`	`i1` divided by `i2` and the quotient rounded towards zero. Same as `rem(i1, i2)` in C99.
Remainder of Reals using rounding	
`realRemainderTowardsZ ero(r1, r2)`	Real remainder with rounding towards zero (`r1 - r2 * roundTowardsZero(r1 / r2)`). Returns qNaN if `r1` or `r2` are qNaN.
Relational Integer functions	
`imin(i1, i2)`	Minimum of `i1` and `i2`.
`imax(i1, i2)`	Maximum of `i1` and `i2`.
Relational Real functions	
`min(r1, r2)`	Minimum of Real variables `r1` and `r2`. Returns qNaN if `r1` or `r2` are qNaN.
`max(r1, r2)`	Maximum of Real variables r1 and r2. Returns qNaN if `r1` or `r2` are qNaN.
Mathematical Real constants and functions	
`euler()`	Target-specific, most-precise representation of Euler's number e (= 2.71828...).
`y := sign(x)`	Sign of x (if x is positive: `y == 1.0`, negative: `y == -1.0`, zero: `y == 0.0`). Returns qNaN if x is qNaN.
`absolute(x)`	Absolute value of Real variable `x`. Returns qNaN if `x` is qNaN.
`fractional(x)`	Fractional part of Real variable `x`. Returns qNaN if `x` is qNaN.
`sqrt(x)`	Square root of `x`. Returns qNaN if `x` is qNaN or `x < 0.0`.
`exp(x)`	Natural base exponential of `x`.
`ln(x)`	Natural logarithm of x. Returns qNaN if `x` is qNaN or `x < 0.0`.
`lg(x)`	Logarithm of `x` to base 10. Returns qNaN if `x` is qNaN or `x < 0.0`.

Function-Name	Description
`safe_posdiv(xn, xd, eps)`	qNaN-free division of `xn` by `xd`: `xn / (if xd >= eps then xd else eps)`. Saturates `eps(min = posMinReal())`. Returns qNaN if `xn`, `xd` or `eps` are qNaN.
`safe_sqrt(x)`	qNaN-free square root of x: `sqrt(if x >= 0.0 then x else 0.0)`. Returns qNaN if x is qNaN.
`safe_ln(x)`	qNaN-free natural logarithm of x: `log(if x >= 0.0 then x else 0.0)`. Returns qNaN if x is qNaN.

Function-Name	Description
`safe_lg(x)`	qNaN-free logarithm to base 10 of x: `lg(if x >= 0.0 then x else 0.0)`. Returns qNaN if x is qNaN.
Trigonometric Real constants and functions	
`pi()`	Target-specific, most-precise representation of π (= 3.14159...), the ratio of a circle's circumference to its diameter.
`sin(x)`	Sine of x. Returns qNaN if x is qNaN, *-∞* or *+∞*.
`cos(x)`	Cosine of x. Returns qNaN if x is qNaN, *-∞* or *+∞*.
`tan(x)`	Tangent of x. Returns qNaN if x is qNaN, *-∞*, *+∞* or `isInfinite(sin(x) / cos(x))` (x is an odd multitude of *π/2*).
`y := asin(x)`	Inverse of `sin(x)` in the range *-π/2 ≤ y ≤ π/2*. Returns qNaN if x is qNaN, `x < -1.0` or `x > 1.0`.
`y := acos(x)`	Inverse of `cos(x)` in the range *0 ≤ y ≤ π*. Returns qNaN if x is qNaN, `x < -1.0` or `x > 1.0`.
`y := atan(x)`	Inverse of `tan(x)` in the range *-π/2 < y < π/2*. Returns qNaN if x is qNaN; *-π/2* if x is *-∞*; *π/2* if x is *+∞*.
`z := atan2(y, x)`	Inverse two-argument tangent in the range *-π < z ≤ π* (angle in the Euclidean plane, given in radians, between the positive x axis and the ray to the point (x, y)). Returns qNaN if y or x are qNaN or `y == 0.0 and x == 0.0`.
`sinh(x)`	Hyperbolic sine of x. Returns qNaN if x is qNaN.
`cosh(x)`	Hyperbolic cosine of x. Returns qNaN if x is qNaN.
`tanh(x)`	Hyperbolic tangent of x. Returns qNaN if x is qNaN.

Function-Name	Description
safe_tan(x)	qNaN-free tangent of x: `if x ≥ π/2 then ∞ elseif x ≤ -π/2 then - ∞ else tan(x)`. Returns qNaN if x is qNaN.
safe_asin(x)	qNaN-free inverse sine of x: `asin(if x > 1.0 then 1.0 elseif x < -1.0 then -1.0 else x)`. Returns qNaN if x is qNaN.
safe_acos(x)	qNaN-free inverse cosine of x: `acos(if x > 1.0 then 1.0 elseif x < -1.0 then -1.0 else x)`. Returns qNaN if x is qNaN.
Systems of linear equations	
x := solveLinearEquations(A, b)	Solution x for linear equations system A*x=b. Signals `SOLVE_LINEAR_EQUATIONS_FAILED` if no unique solution exists or `hasNaN2D(A) == true` or `hasNaN1D(b) == true`, in which case `allNaN1D(x) == true`.

Function-Name	Description
`(LU, pivots) := luFactorize(A)`	LU decomposition with partial pivoting of square matrix `A`. Signals `SOLVE_LINEAR_EQUATIONS_FAILED` if no unique solution exists or `hasNaN2D(A) == true`, in which case `allNaN2D(LU) == true`.
`x := luSolve(LU, pivots, b)`	Solution `x` for LU-factorized linear equations system `L*U*x = b[pivots]`, with `LU == L*U`. Signals `SOLVE_LINEAR_EQUATIONS_FAILED` if no unique solution exists or `hasNaN2D(LU) == true` or `hasNaN1D(pivots) == true` or `hasNaN1D(b) == true`, in which case `allNaN1D(x) == true`.
Interpolation in 1D/2D/3D	
`interpolation1D(x1, x1_data, nx1, y_data, ipo, expo)`	Constant/linear interpolation in 1D with extrapolation. Saturates `nx1(min = 2, max = size(x1_data, 1))`, `interpolation(min = 1, max = 2)`, `extrapolation(min = 1, max = 2)`.
`interpolation2D(x1, x2, x1_data, nx1, nx2_data, nx2, y_data, ipo, expo)`	Constant/linear interpolation in 2D with extrapolation. Saturates `nx1(min = 2, max = size(x1_data, 1))`, `nx2(min = 2, max = size(x2_data, 1))`, `interpolation(min = 1, max = 2)`, `extrapolation(min = 1, max = 2)`.
`interpolation3D(x1, x2, x3, x1_data, nx1, nx2_data, nx2, nx3_data, nx3, y_data, ipo, expo)`	Constant/linear interpolation in 3D with extrapolation.

Precise Definitions

> **S-2.9 (builtin functions / Syntactical-structure, terminology):** Let $C_{builtin} = C_{builtin1} \circ C_{builtin2} \circ C_{builtin3} \circ C_{builtin4}$ where each $C_{builtinn}$ with $n \in \{1,2,...,4\}$ is a sequence of characters defined in the following and $\circ$ is the left-to-right concatenation of sequences of characters. $C_{builtin}$ is implicitly appended to each program; its functions are called builtin. Functions that are not builtin are called user-defined.
>
> In Appendix TODO further built-in functions are defined that are not yet part of the eFMI standard but likely will be added in the future. Therefore, the names and functionality of these functions are reserved. The following definition of built-in functions may refer to functions defined in the appendix.
>
> $C_{builtin1}$ is the following sequence of characters:

```
1 /*
2    Note: We distinguish integer and Integer. Integer with uppercase first letter is the
  type
3        Integer -- a target-specific data-type -- whereas integer with lowercase first
4        letter is the mathematic term for numbers without fractional component.
```

```
     Likewise,
  5          we distinguish real and Real.
  6  */
  7
  8  /*******************************************************************************
     *****
  9      Properties of Integer:
 10  *******************************************************************************
     ****/
 11
 12  function minInteger
 13      output Integer i;
 14  algorithm /*
 15      i := target-specific smallest Integer;
 16  */ end minInteger;
 17
 18  function maxInteger
 19      output Integer i;
 20  algorithm /*
 21      i := target-specific largest Integer;
 22  */ end maxInteger;
 23
 24  /*******************************************************************************
     *****
 25      Properties of Real:
 26  *******************************************************************************
     ****/
 27
 28  function minReal
 29      outputs Real r;
 30  algorithm /*
 31      r := target-specific smallest, not -∞ representing, Real;
 32  */ end minReal;
 33
 34  function maxReal
 35      outputs Real r;
 36  algorithm /*
 37      r := target-specific largest, not +∞ representing, Real;
 38  */ end maxReal;
 39
 40  function posMinReal
 41      output Real r;
 42  algorithm /*
 43      r := target-specific smallest Real > 0.0;
 44  */ end posMinReal;
 45
 46  function epsReal
 47      output Real r;
 48  algorithm /*
 49      r := target-specific largest Real r > 0.0 such that 1.0 + r == 1.0;
 50  */ end epsReal;
 51
 52  function nan
 53      output Real r;
 54  algorithm /*
 55      r := target-specific not-a-number representation;
 56  */ end nan;
 57
```

```
58 function isNaN
59     input  Real r;
60     output Boolean b;
61 algorithm /*
62     b := true if r is target-specific not-a-number representation, false otherwise (if
   IEEE 754-2019 is satisfied: 'b := r <> r;');
63 */ end isNaN;
64
65 function minusInfinite
66     output Real r;
67 algorithm /*
68     r := target-specific -∞ representation;
69 */ end minusInfinite;
70
71 function plusInfinite
72     output Real r;
73 algorithm /*
74     r := target-specific +∞ representation;
75 */ end minusInfinite;
76
77 function isInfinite
78     input  Real r;
79     output Boolean b;
80 algorithm /*
81     b := r == minusInfinite() or r == plusInfinite();
82     if signal in NAN then
83         b := false;
84     end if;
85 */ end isInfinite;
86
87 function isFinite
88     input  Real r;
89     output Boolean b;
90 algorithm /*
91     b := not(isNaN(r)) and not(isInfinite(r));
92 */ end isFinite;
93
94 /*********************************************************************************************
   *****
95     Multi-dimensional properties of Real:
96 *********************************************************************************************
   ****/
97
98 function hasNaN1D
99     input  Real r[:];
100     output Boolean b;
101 algorithm /*
102     b := false;
103     for i in 1:size(r, 1)
104         if isNaN(r[i])
105             b := true;
106         end if;
107     end for;
108 */ end hasNaN1D;
109
110 function hasNaN2D
111     input  Real r[:, :];
112     output Boolean b;
```

```
113 algorithm /*
114     b := false;
115     for i in 1:size(r, 1)
116         for j in 1:size(r, 2)
117             if isNaN(r[i, j])
118                 b := true;
119             end if;
120         end for;
121     end for;
122 */ end hasNaN2D;
123
124 /*********************************************************************************
    *****
125     Numeric type conversions:
126 *********************************************************************************
    ****/
127
128 function real
129     input  Integer i;
130     output Real r;
131 algorithm /*
132     r := target-specific Real representation of i;
133 */ end real;
134
135 function integer
136     signals NAN, OVERFLOW;
137     input Real r;
138     output Integer i;
139 protected
140     Real tmp;
141 algorithm /*
142     i := 0;
143     tmp := roundTowardsZero(r); /* Returns qNaN if r is qNaN. */
144     if tmp < real(minInteger()) or tmp > real(maxInteger()) then
145         signal OVERFLOW;
146     elseif signal in NAN then
147         signal NAN; /* tmp was qNaN. */
148     else
149         i := target-specific Integer representation of tmp;
150     end if;
151 */ end integer;
152
153 /*********************************************************************************
    *****
154     Direct Real rounding:
155 *********************************************************************************
    ****/
156
157 function roundDown
158     input Real r;
159     output Real i;
160 algorithm /*
161     /* Also known as: flooring, round towards -∞. */
162     if isNaN(r) then
163         i := nan();
164     else
165         i := target-specific greatest integer ≤ r;
166     end if;
```

```
167  */ end roundDown;
168
169  function roundUp
170      input Real r;
171      output Real i;
172  algorithm /*
173      /* Also known as: ceiling, round towards +∞. */
174      if isNaN(r) then
175          i := nan();
176      else
177          i := target-specific least integer >= r;
178      end if;
179  */ end roundUp;
180
181  /************************************************************************************
     *****
182      Nearest Real rounding (using a tie-breaking rule):
183  ************************************************************************************
     ****/
184
185  function roundHalfToEven
186      input Real r;
187      output Real i;
188  algorithm /*
189      /* Also known as: convergent rounding, statistician's rounding, Dutch rounding,
190          Gaussian rounding, odd-even rounding, bankers' rounding. */
191      i := (if roundHalfDown(r) < roundHalfUp(r)
192          then (if (r + 0.5 is even) then r + 0.5 else r - 0.5)
193          else roundHalfDown(r));
194      if signal in NAN or isNaN(r) then
195          i := nan();
196      end if;
197  */ end roundHalfToEven;
198
199  /************************************************************************************
     *****
200      Relational Integer functions:
201  ************************************************************************************
     ****/
202
203  function imin
204      input  Integer u1;
205      input  Integer u2;
206      output Integer y;
207  algorithm /*
208      y := (if u1 < u2 then u1 else u2);
209  */ end imin;
210
211  function imax
212      input  Integer u1;
213      input  Integer u2;
214      output Integer y;
215  algorithm /*
216      y := (if u1 > u2 then u1 else u2);
217  */ end imax;
218
219  /************************************************************************************
     *****
```

```
220     Relational Real functions:
221 *********************************************************************
    ****/
222
223 function min
224     input  Real u1;
225     input  Real u2;
226     output Real y;
227 algorithm /*
228     y := (if u1 < u2 then u1 else u2);
229     if signal in NAN then
230         y := nan();
231     end if;
232 */ end min;
233
234 function max
235     input  Real u1;
236     input  Real u2;
237     output Real y;
238 algorithm /*
239     y := (if u1 > u2 then u1 else u2);
240     if signal in NAN then
241         y := nan();
242     end if;
243 */ end max;
244
245 /*******************************************************************
    *****
246     Mathematical Real constants and functions:
247 *********************************************************************
    ****/
248
249 function euler
250     output Real r;
251 algorithm /*
252     r := target-specific, most-precise representation of e;
253 */ end euler;
254
255 function sign
256     input  Real r;
257     output Real i;
258 algorithm /*
259     i := (if r > 0.0 then 1.0 elseif r < 0.0 then -1.0 else 0.0);
260     if signal in NAN then
261         i := nan();
262     end if;
263 */ end sign;
264
265 function fractional
266     input  Real x;
267     output Real y;
268 algorithm /*
269     y := x - roundTowardsZero(x);
270 */ end fractional;
271
272 function absolute
273     input  Real x;
274     output Real y;
```

```
275 algorithm /*
276     y := sign(x) * x;
277 */ end absolute;
278
279 function sqrt
280     input  Real x;
281     output Real y;
282 algorithm /*
283     if x < 0.0 then
284         y := nan();
285     elseif signal in NAN then
286         y := nan();
287     else
288         y := x^0.5;
289     end if;
290 */ end sqrt;
291
292 function exp
293     input  Real x;
294     output Real y;
295 algorithm /*
296     y := euler()^x;
297 */ end exp;
298
299 function ln
300     input  Real x;
301     output Real y;
302 algorithm /*
303     if x < 0.0 then
304         y := nan();
305     elseif signal in NAN then
306         y := nan();
307     else
308         y := natural logarithm of x;
309     end if;
310 */ end ln;
311
312 function lg
313     input  Real x;
314     output Real y;
315 algorithm /*
316     if x < 0.0 then
317         y := nan();
318     elseif signal in NAN then
319         y := nan();
320     else
321         y := logarithm to base 10 of x;
322     end if;
323 */ end lg;
324
325 function safe_posdiv
326     input  Real xn;
327     input  Real xd;
328     input  Real eps
329         (min = posMinReal());
330     output Real y;
331 algorithm /*
332     y := xn / (if xd >= eps then xd else eps);
```

```
333     if signal in NAN then
334         y := nan();
335     end if;
336 */ end safe_posdiv;
337
338 function safe_sqrt
339     input  Real x;
340     output Real y;
341 algorithm /*
342     y := sqrt(if x < 0.0 then 0.0 else x);
343     if signal in NAN then
344         y := nan();
345     end if;
346 */ end safe_sqrt;
347
348 function safe_ln
349     input  Real x;
350     output Real y;
351 algorithm /*
352     y := ln(if x < 0.0 then 0.0 else x);
353     if signal in NAN then
354         y := nan();
355     end if;
356 */ end safe_ln;
357
358 function safe_lg
359     input  Real x;
360     output Real y;
361 algorithm /*
362     y := lg(if x < 0.0 then 0.0 else x);
363     if signal in NAN then
364         y := nan();
365     end if;
366 */ end safe_lg;
367
368 /*******************************************************************************
    *****
369     Trigonometric Real constants and functions:
370 *******************************************************************************
    ****/
371
372 function pi
373     output Real r;
374 algorithm /*
375     r := target-specific, most-precise representation of π;
376 */ end pi;
377
378 function sin
379     input Real x;
380     output Real y;
381 algorithm /*
382     if not(isFinite(x)) then
383         y := nan();
384     else
385         y := sine of x;
386     end if;
387 */ end sin;
388
```

```
389 function cos
390     input Real x;
391     output Real y;
392 algorithm /*
393     if not(isFinite(x)) then
394         y := nan();
395     else
396         y := cosine of x;
397     end if;
398 */ end cos;
399
400 function tan
401     input Real x;
402     output Real y;
403 algorithm /*
404     if not(isFinite(x)) then
405         y := nan();
406     else
407         y := sin(x) / cos(x);
408     end if;
409     if isInfinite(y) then
410         y := nan();
411     end if;
412 */ end tan;
413
414 function asin
415     input  Real x;
416     output Real y;
417 algorithm /*
418     if -1.0 <= x and x <= 1.0 then
419         y := inverse of sin(x) in the range -π/2 ≤ y ≤ π/2;
420     elseif signal in NAN or true then
421         y := nan();
422     end if;
423 */ end asin;
424
425 function acos
426     input  Real x;
427     output Real y;
428 algorithm /*
429     if -1.0 <= x and x <= 1.0 then
430         y := inverse of cos(x) in the range 0 ≤ y ≤ π;
431     elseif signal in NAN or true then
432         y := nan();
433     end if;
434 */ end asin;
435
436 function atan
437     input Real x;
438     output Real y;
439 algorithm /*
440     if isNaN(x)) then
441         y := nan();
442     elseif isInfinite(x) then
443         y := sign(x) * pi() / 2.0;
444     else
445         y := inverse of tan(x) in the range -π/2 < y < π/2;
446     end if;
```

```
447 */ end atan;
448
449 function atan2
450     input  Real y;
451     input  Real x;
452     output Real z;
453 algorithm /*
454     z := (if     x > 0.0              then atan(y / x)
455           elseif x < 0.0 and y >= 0.0 then atan(y / x) + pi()
456           elseif x < 0.0 and y <  0.0 then atan(y / x) - pi()
457           elseif y > 0.0              then  pi() / 2.0
458           elseif y < 0.0              then -pi() / 2.0
459           else                             nan());
460     if signal in NAN then
461         z := nan();
462     end if;
463 */ end atan2;
464
465 function sinh
466     input Real x;
467     output Real y;
468 algorithm /*
469     y := (euler()^x - euler()^-x) / 2.0;
470 */ end sinh;
471
472 function cosh
473     input Real x;
474     output Real y;
475 algorithm /*
476     y := (euler()^x + euler()^-x) / 2.0;
477 */ end cosh;
478
479 function tanh
480     input Real x;
481     output Real y;
482 algorithm /*
483     y := sinh(x) / cosh(x);
484 */ end tanh;
485
486 function safe_tan
487     input  Real x;
488     output Real y;
489 algorithm /*
490     y := (if x >= pi() / 2.0 then plusInfinite()
491         elseif x <= -pi() / 2.0 then minusInfinite()
492         else tan(x));
493     if signal in NAN then
494         y := nan();
495     end if;
496 */ end safe_tan;
497
498 function safe_asin
499     input  Real x;
500     output Real y;
501 algorithm /*
502     y := asin(if x > 1.0 then 1.0 elseif x < -1.0 then -1.0 else x);
503     if signal in NAN then
504         y := nan();
```

```
505        end if;
506 */ end safe_asin;
507
508 function safe_acos
509        input  Real x;
510        output Real y;
511 algorithm /*
512        y := acos(if x > 1.0 then 1.0 elseif x < -1.0 then -1.0 else x);
513        if signal in NAN then
514            y := nan();
515        end if;
516 */ end safe_acos;
517
518 /**********************************************************************************************
    *****
519        Systems of linear equations:
520 **********************************************************************************************
    ****/
521
522 function solveLinearEquations
523        signals SOLVE_LINEAR_EQUATIONS_FAILED;
524        input  Real A[:, size(A,1)];
525        input  Real b[size(A,1)];
526        output Real x[size(A,1)];
527 algorithm /*
528        Solve system of linear equations A*x = b for x. Hereby it is assumed that matrix A is
529        regular. Typically, the function implements a direct Gaussian elimination with
    partial
530        pivoting. If A is singular, SOLVE_LINEAR_EQUATIONS_FAILED is signaled
531        and at least on element of x is set to qNaN.
532 */ end solveLinearEquations;
533
534 function luFactorize
535        signals SOLVE_LINEAR_EQUATIONS_FAILED;
536        input  Real    A[:, size(A, 1)];
537        output Real    LU[:, size(A, 1)];
538        output Integer pivots[size(A, 1)];
539        /*
540        The function returns the LU decomposition with partial pivoting of the square,
541        matrix A: P*L*U = A where P is the permutation matrix (implicitely defined by vector
542        pivots), L is a lower triangular matrix with unit diagonal elements and U is an upper
543        triangular matrix. Matrices L and U are stored in matrix LU on return (the diagonal
    of
544        L is not stored). With the companion function luSolve, the factorization is used to
545        solve the linear system L*U*x = b[pivots] with different right hand side vectors b.
546
547        If A is singular, SOLVE_LINEAR_EQUATIONS_FAILED is signaled.
548
549        The algorithm below is "conceptual". A more efficient implementation uses
550        BLAS functions, see, e.g., LAPACK function DGETRF.
551        */
552 protected
553        Integer n;
554        Integer p; /* Pivot index. */
555        Integer pk;
556        Real temp;
557        Real eta;
558        Real d;
```

```
559    Real d_max;
560    Real di;
561    Real di_abs;
562 algorithm
563    n := size(A,1);
564    LU := A;
565    p := 1:n;
566    if n < 1 then
567       return;
568    end if;
569
570    for k in 1:n-1 loop
571       /* Find pivot */
572       p :=k;
573       d := LU[k,k];
574       d_max :=absolute(d);
575       for i in k+1:n loop
576          di := LU[i,k];
577          di_abs := abs(di);
578          if di_abs > d_max then
579             p      := i;
580             d      := di;
581             d_max := di_abs;
582          end if;
583       end for;
584
585       /* Test pivot for singularity */
586       if d == 0 then
587          signals SOLVE_LINEAR_EQUATIONS_FAILED;
588       else
589          /* Swap LU[k,j] and LU[p,j], for j = 1,....,n
590             as well as pivots[k] and pivots[p] */
591          if k <> p then
592             for j in 1:n loop
593                temp :=LU[k, j];
594                LU[k,j] :=LU[p, j];
595                LU[p,j] :=temp;
596             end for;
597             pk :=pivots[k];
598             pivots[k] :=pivots[p];
599             pivots[p] :=pk;
600          end if;
601
602          /* LU factors */
603          for i in k+1:n loop
604             eta :=LU[i,k]/d;
605             LU[i,k] :=eta;
606
607             for j in k+1:n loop
608                LU[i,j] :=LU[i,j] - eta*LU[k,j];
609             end for;
610          end for;
611       end if;
612    end for;
613 end luFactorize;
614
615 function luSolve
616    signals SOLVE_LINEAR_EQUATIONS_FAILED;
```

```
617    input  Real    LU[:, size(LU, 1)];  /* Returned from luFactorize. */
618    input  Integer pivots[size(LU, 1)]; /* Returned from luFactorize. */
619    input  Real    b[size(LU, 1)];
620    output Real    x[size(LU, 1)];
621    /*
622    The function returns the solution x of the linear system of equations:
623        L*U*x = b[pivots]
624    where L*U and pivots are computed by the companion function luFactorize.
625    If a unique solution cannot be computed (i.e., U is singular),
626    SOLVE_LINEAR_EQUATIONS_FAILED is signaled and at least one element of x is qNaN.
627
628    The algorithm below is "conceptual". A more efficient implementation uses
629    BLAS functions, see, e.g., LAPACK function DGETRS.
630    */
631 protected
632    Integer n=size(LU,1);
633    Real y[size(LU,1)];
634 algorithm
635    if n < 1 then
636        return;
637    end if;
638
639    /* Forward elimination */
640    for i in 1:n loop
641       y[i] := b[pivots[i]];
642       for j in 1:i-1 loop
643          y[i] :=y[i] - LU[i, j]*y[j];
644       end for;
645    end for;
646
647    /* Backward substitution */
648    for i in n:-1:1 loop
649       x[i] :=y[i];
650       for j in i+1:n loop
651          x[i] := x[i] - LU[i,j]*x[j];
652       end for;
653       x[i] := x[i]/LU[i,i];
654       if isNaN(x[i])
655        signals SOLVE_LINEAR_EQUATIONS_FAILED;
656       end
657    end for;
658 end luSolve;
659
660
661 /********************************************************************************
   *****
662    Interpolation in 1D/2D/3D:
663
664    In all functions the following options are used:
665    - interpolation = 1: constant bottom interpolation
666                    = 2: linear interpolation
667    - extrapolation = 1: hold last value
668                    = 2: linear extrapolation through last two boundary points
669
670    A production code generator would typically trigger an error, if the folloing
671    conditions are not fulfilled when calling one of the interpolation functions:
672    - The values in x1_data[1:nx1], x2_data[1:nx2], x3_data[1:nx3] are
673      strict monotonically increasing.
```

```
674        - The data arguments (x1_data, x2_data, x3_data, nx1, nxs2, nx3) are parameters.
675        - The option arguments (interpolation, extrapolation) are literal constants.
676
677      The production code generator decides which "search" method to use to find the
678      respective interval, or whether it can be directly found because there is an
679      equidistant grid.
680  **********************************************************************************
     ****/
681
682  function interpolation1D
683      input Real x1;
684      input Real x1_data[:]; /* Must be strict monotonically increasing values. */
685      input Integer nx1
686          (min = 2, max = size(x1_data, 1));
687      input Real y_data[size(x1_data, 1)];
688      input Integer interpolation
689          (min = 1, max = 2);
690      input Integer extrapolation
691          (min = 1, max = 2);
692      output Real y;
693  algorithm /*
694      Constant or linear interpolation in [x1_data[1:nx1], y_data[1:nx1]]
695      given the abszissa value x1.
696  */ end interpolation1D;
697
698  function interpolation2D
699      input Real x1;
700      input Real x2;
701      input Real x1_data[:]; /* Must be strict monotonically increasing values. */
702      input Integer nx1
703          (min = 2, max = size(x1_data, 1));
704      input Real x2_data[:]; /* Must be strict monotonically increasing values. */
705      input Integer nx2
706          (min = 2, max = size(x2_data, 1));
707      input Real y_data[size(x1_data, 1), size(x2_data, 1)];
708      input Integer interpolation
709          (min = 1, max = 2);
710      input Integer extrapolation
711          (min = 1, max = 2);
712      output Real y;
713  algorithm /*
714      Constant or linear interpolation with x1_data[1:nx1], x2_data[1:nx2]
715      abszissa values and y_data[1:nx1, 1:nx2] ordinate values, given the abszissa value
     x1, x2.
716  */ end interpolation2D;
717
718  function interpolation3D
719      input  Real    x1;
720      input  Real    x2;
721      input  Real    x3;
722      input  Real    x1_data[:];              /* strict monotonically increasing values */
723      input  Integer nx1;                     /* 2 ≤ nx1 ≤ size(x1_data, 1) */
724      input  Real    x2_data[:];              /* strict monotonically increasing values */
725      input  Integer nx2;                     /* 2 ≤ nx2 ≤ size(x2_data, 1) */
726      input  Real    x3_data[:];              /* strict monotonically increasing values */
727      input  Integer nx3;                     /* 2 ≤ nx3 ≤ size(x3_data, 1) */
728      input  Real    y_data[size(x1_data, 1), size(x2_data, 1), size(x3_data, 1)];
729      input  Integer interpolation;
```

```
730     input  Integer extrapolation;
731     output Real    y;
732 algorithm /*
733     Constant or linear interpolation with x1_data[1:nx1], x2_data[1:nx2], x3_data[1:nx3]
734     abszissa values and y_data[1:nx1, 1:nx2, 1:nx3] ordinate values,
735     given the abszissa value x1, x2, x3.
736 */ end interpolation3D;
```

$C_{builtin2}$ defines builtin functions for *Integer* division:

```
 1 function divisionTowardsZero
 2     input Integer dividend;
 3     input Integer divisor;
 4     output Integer quotient;
 5 algorithm /*
 6     quotient := integer(roundTowardsZero(real(dividend) / real(divisor)));
 7 */ end divisionTowardsZero;
 8
 9 function remainderTowardsZero
10     input Integer dividend;
11     input Integer divisor;
12     output Integer remainder;
13 algorithm /*
14     remainder := dividend - divisor * divisionTowardsZero(dividend, divisor);
15 */ end remainderTowardsZero;
```

$C_{builtin3}$ defines builtin functions for *Real* division, where the quotient is forced to be an integer according to a rounding strategy:

```
 1 function realRemainderTowardsZero
 2     input Real dividend;
 3     input Real divisor;
 4     output Real remainder;
 5 algorithm /*
 6     remainder := dividend - divisor * roundTowardsZero(dividend / divisor);
 7 */ end realRemainderTowardsZero;
```

$C_{builtin4}$ lifts builtin functions with scalar in- and output parameters for usage with multi-dimensions. For every function named α of $C_{builtin1}, ..., C_{builtin3}$ with a scalar input parameter β and a scalar output parameter δ of types *T1,T3∈{Boolean, Integer, Real}* respectively, $C_{builtin4}$ contains the character sequence:

```
 1 function α1D
 2     input T1 β[:];
 3     output T3 δ[size(β, 1)];
 4 algorithm /*
 5     for i in 1:size(β, 1) loop
 6         δ[i] := α(β[i]);
 7     end for;
 8 */ end α1D;
 9
10 function α2D
```

```
11     input T1 β[:, :];
12     output T3 δ[size(β, 1), size(β, 2)];
13 algorithm /*
14     for i in 1:size(β, 1) loop
15         for j in 1:size(β, 2) loop
16             δ[i, j] := α(β[i, j]);
17         end for;
18     end for;
19 */ end α2D;
```

For every function named α of $C_{builtin1},...,C_{builtin3}$ with two scalar input parameters β and γ and a scalar output parameter δ of types *T1,T2,T3∈{Boolean, Integer, Real}* respectively, $C_{builtin4}$ contains the character sequence:

```
 1 function α1D
 2     input T1 β[:];
 3     input T2 γ[size(β, 1)];
 4     output T3 δ[size(β, 1)];
 5 algorithm /*
 6     for i in 1:size(β, 1) loop
 7         δ[i] := α(β[i], γ[i]);
 8     end for;
 9 */ end α1D;
10
11 function α2D
12     input T1 β[:, :];
13     input T2 γ[size(β, 1), size(β, 2)];
14     output T3 δ[size(β, 1), size(β, 2)];
15 algorithm /*
16     for i in 1:size(β, 1) loop
17         for j in 1:size(β, 2) loop
18             δ[i, j] := α(β[i, j], γ(i, j));
19         end for;
20     end for;
21 */ end α2D;
```

Above functions are in lexical order w.r.t. their names; they constitute $C_{builtin4}$ in its entirety.

L-1 (semantic of builtin functions / Runtime-semantic): $C_{builtin}$ defines the semantic of each builtin function in prose via the *comment* part of it; the actual implementation is up to Production Code generators however (cf. **L-2**).

The builtin functions `divisionDown`, `divisionUp` and `divisionTowardsZero` are also known as floored division, ceiled division and truncated division respectively.

According to their definition, the remainder returned by `remainderDown`, `remainderTowardsZero` and `remainderEuclidean` is signed like the `divisor`, `dividend` or always positive respectively.

R-1: Builtin functions are derived by the `{ function-declaration }` factor of **G-2.1** in

the order of their definition in $C_{builtin}$ and — because $C_{builtin}$ is appended — follow user-defined functions.

R-2: Builtin functions are without 6'th child, i.e., without *statements* and therefore implementation body. The motivation to implicitly append their signatures and thereby making them part of *blocks* as described in **R-1** is to cover builtin functions under the umbrella of functions, such that the common syntactic and semantic rules for such apply for builtin as well as user-defined functions; only exceptional cases for either have to be additionally defined. In fact, **S-2.9** already encapsulates all differences between builtin and user-defined functions. For example, according to **S-2.5**, functions must have unique names, implying that user-defined functions must not be named like a builtin function. And considering $C_{builtin}$ and **S-2.3**, all builtin functions are stateless. Likewise, according to **S-2.10**, builtin functions do not locally — and therefore neither transitively — call functions.

L-2 (target-specific builtin function implementation; statically-evaluated builtin functions / Runtime-semantic): The actual implementation of builtin functions is up to Production Code generators, which are supposed to optimize such for the targeted runtime environment. The only restrictions are, that the execution of builtin functions must always terminate and be side-effect-free — i.e., not change or depend on the control-state.

Optimizations include, for example, the implementation of builtin functions in terms of inlined code or even the replacement of builtin function calls and sequences thereof by target-specific — but semantic-wise equivalent — hardware operations. The `roundHalfToEven` builtin function for example is the default rounding mode used in the IEEE 754-2019 standard for floating-point arithmetic and therefore likely hardware supported. Also `integer` is often provided as single CPU-instruction like `CVTTSS2SI` or `CVTTSD2SI` of Streaming SIMD Extensions 2 (SSE2); and `roundDown`, `roundUp`, `roundTowardsZero` and `roundHalfToEven` are provided by `ROUNDSS` and `ROUNDSD` of SSE4. Particularly the multi-dimensions support of $C_{builtin4}$ likely can be much more efficient than the given naïve iterative solution; SSE4 for example provides for most single data instructions corresponding multiple data instructions (SIMD hardware operations: single instruction, multiple data).

Builtin functions that are part of statically-evaluated expressions must be applied already for Production Code generation since they define dimensional-sizes, multi-dimension queries or loop iteration bounds which are subject to well-formedness constraints. The well-formedness and results of such statically-evaluated builtin function calls depend on the targeted runtime environment. For example, in a 32-bit environment `integer(roundUp(2.0^31 - 1.0))` likely is an error due to an integer overflow, which in turn would result in `integer` signaling `OVERFLOW` which is not permitted within statically-evaluated expressions (cf. **S-X:TODO:error-signal-freeness-of-statically-evaluated-expressions**).

E-1: The following *block* uses the builtin function solveLinearEquations to compute a control-output vector based on a single control-input:

```
 1 block TestSolveLinearEquations
 2     input  Real u;
 3     output Real y[2];
 4
 5 protected
 6
 7 public
 8     method Startup
 9     protected
10     algorithm
11         self.y := {0.0, 0.0};
12     end Startup;
13
14     method DoStep
15     protected
16     algorithm
17         self.y := solveLinearEquations(
18             {
19                 {1.0       , 2.0*self.u},
20                 {4.0*self.u, 5.0}
21             },
22             {-2.0      , 4.0*self.u});
23         /* Rudimentary error handling */
24         if signal or hasNaN(self.y) then
25             self.y = {0.0, 0.0}
26         end;
27     end DoStep
28 end TestSolveLinearEquations;
```

E-2: The following *block* uses luFactorize and luSolve to solve two systems of linear equations $A*x = b$ for the same regular matrix *A* but varying *b*:

```
 1 block TestLuSolve
 2     input  Real u;
 3     output Real y[2];
 4
 5 protected
 6
 7 public
 8     method Startup
 9     protected
10     algorithm
11         self.y := {0.0, 0.0};
12     end Startup;
13
14     method DoStep
15     protected
16         Real LU[2,2];
17         Real pivots[2];
```

```
18     algorithm
19         (LU, pivots) := luFactorize(
20             {
21                 {1.0,         2.0*self.u},
22                 {4.0*self.u, 5.0}
23             });
24         self.y := luSolve(
25             LU,
26             pivots,
27             luSolve(
28                 LU,
29                 pivots,
30                 {-2.0, 4.0*self.u})
31             + {-3.0, 6.0*self.u});
32         /* Rudimentary error handling */
33         if signal or isNaN(self.y) then
34             self.y = {0.0, 0.0}
35         end;
36     end DoStep;
37 end TestLuSolve;
```

LU decomposition typically is more efficient than naïvely using several solveLinearEquations calls, at least when *A* has more realistic sizes than the tiny *2x2* in above example which has been selected for demonstration purposes only.

E-3: The following *block* interpolates in a vector of data points:

```
1 block TestInterpolation
2     input  Real x;
3     output Real y;
4
5     parameter Real x_data[7];  /* Define x-axis data points as tuneable parameter
   vector. */
6     parameter Real y_data[7];  /* Define y-axis data as tuneable parameter vector.
   */
7     parameter Integer nx;      /* Number of elements to interpolate (1 ≤ nx ≤ 7).
   */
8
9 protected
10
11 public
12     method Startup
13     protected
14         Real x;
15     algorithm
16         x := 0.0;
17         self.nx := 4;
18         self.x_data := {1.0, 2.0, 3.0, 4.0 , 0.0, 0.0, 0.0};
19         self.y_data := {1.0, 4.0, 9.0, 16.0, 0.0, 0.0, 0.0};
20         self.y := interpolation1D(x, self.x_data, self.nx, self.y_data, 2, 2);
21     end Startup;
22
23     method DoStep
24     protected
```

```
25      algorithm
26          self.y := interpolation1D(2*self.x, self.x_data, self.nx, self.y_data, 2,
    2);
27      end DoStep;
28 end TestInterpolation;
```

3.2.7. Example Application Scenarios

Modelica-modeled PID-controller

The following example has its origin in a Modelica model for a speed controller — a PID controller with output limitations — of a DC motor. The block diagram of the Modelica model has two input signals wLoadRef and wMotor. The input signal wLoadRef is the desired value of the speed of the motor load whereas wMotor is the current speed of the motor. The output of the controller is vMotor — the voltage to be applied to the DC motor.
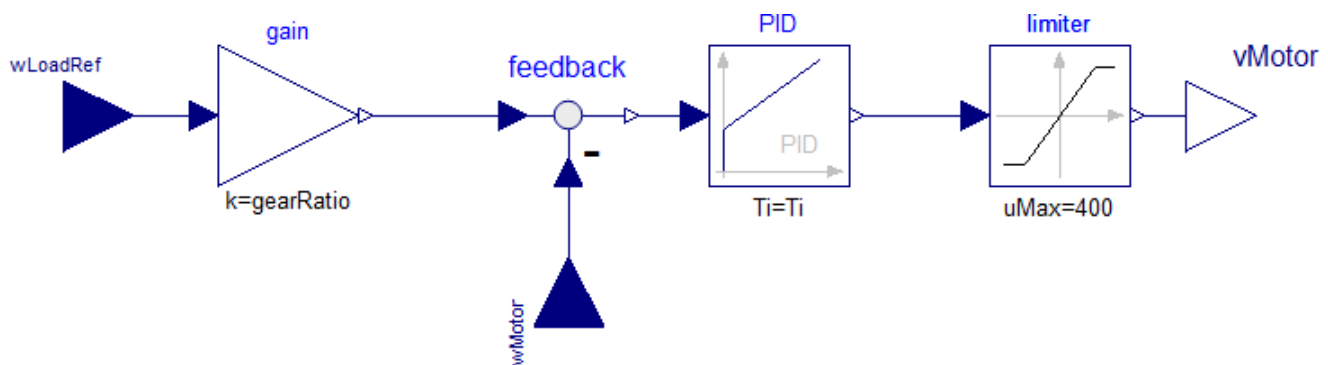
It follows one possible transformation of this Modelica model into an eFMI GALEC program. The discretization of the dynamic parts of the PID controller is realized by the Explicit Euler method. The respective eFMI GALEC program is:

```
 1 block PID_Controller
 2     input  Real wLoadRef(min = -1.0e5, max = 1.0e5);
 3     input  Real wMotor  (min = -1.0e5, max = 1.0e5);
 4     output Real vMotor  (min = -1.0e7, max = 1.0e7);
 5
 6     // Tunable parameters (can be changed via recalibration):
 7     parameter Real 'limiter.uMax'(min = 1.0, max = 1.0e5);
 8     parameter Real gearRatio(min = 10.0, max = 500.0);
 9     parameter Real Ti(min = 1.0e-7, max = 100.0);
10     parameter Real Td(min = 1.0e-7, max = 100.0);
11     parameter Real kd(min = 0.0, max = 1000.0);
12     parameter Real k(min = 0.0, max = 1000.0);
13     parameter Real stepSize // Can be local constant (if recalibration is not supported).
14         (min = 1.0e-10, max = 0.01 /* in physics-simulation tested sampling-range */ );
15
16 protected
17     // Dependent parameters:
18     parameter Real 'limiter.uMin'(min = -1.0e5, max = -1.0);
19
20     // Discrete states:
```

```
21      Real 'PID.I.x';
22      Real 'PID.D.x';
23      Real 'previous(feedback.y)';
24      Boolean firstTick;
25
26  public
27      method Startup
28      algorithm
29          // Initialize tunable parameters:
30          self.'limiter.uMax' := 400.0;
31          self.gearRatio := 105.0;
32          self.Ti := 0.1;
33          self.Td := 0.1;
34          self.kd := 0.1;
35          self.k := 10.0;
36          self.stepSize := 1e-3;
37
38          // Initialize dependent parameters:
39          self.'limiter.uMin' := -self.'limiter.uMax';
40
41          // Initialize discrete states:
42          self.'PID.I.x' := 0.0;
43          self.'PID.D.x' := 0.0;
44          self.'previous(feedback.y)' := 0.0;
45          self.firstTick := true;
46
47          // Initialize outputs:
48          self.vMotor := 0.0;
49      end Startup;
50
51
52      method Recalibrate
53      algorithm
54          // Update dependent parameters:
55          self.'limiter.uMin' := -self.'limiter.uMax';
56      end Recalibrate;
57
58
59      /*
60          Control-cycle function: Called at every clock tick.
61      */
62      method DoStep
63      protected
64          Real 'gain.y';
65          Real 'feedback.y';
66          Real 'derivative(PID.I.x)';
67          Real 'derivative(PID.D.x)';
68          Real 'PID.D.y';
69          Real 'PID.y';
70      algorithm
71
72          if self.firstTick then
73              self.firstTick := false;
74          else
75              'derivative(PID.I.x)' := self.'previous(feedback.y)' / self.Ti;
76              'derivative(PID.D.x)' := (self.'previous(feedback.y)' - self.'PID.D.x') / self.Td;
77
78              self.'PID.I.x'          := self.'PID.I.x' + self.stepSize * 'derivative(PID.I.x)';
```

```
79              self.'PID.D.x'        := self.'PID.D.x' + self.stepSize * 'derivative(PID.D.x)';
80          end if;
81
82          'gain.y'      := self.gearRatio * self.wLoadRef;
83          'feedback.y' := 'gain.y' - self.wMotor;
84
85          'PID.D.y' := self.kd * ('feedback.y' - self.'PID.D.x') / self.Td;
86          'PID.y'   := self.k * ('PID.D.y' + self.'PID.I.x' + 'feedback.y');
87
88          self.vMotor := (
89              if 'PID.y' > self.'limiter.uMax' then
90                  self.'limiter.uMax'
91              elseif 'PID.y' < self.'limiter.uMin' then
92                  self.'limiter.uMin'
93              else
94                  'PID.y'
95          );
96
97          self.'previous(feedback.y)' := 'feedback.y';
98      end DoStep;
99 end PID_Controller;
```

The manifest for the controller, just describing its interface, is:

```xml
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <Manifest
 3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 4     xsi:noNamespaceSchemaLocation="../schemas/AlgorithmCode/efmiAlgorithmCodeManifest.xsd"
 5     xsdVersion="0.14.0"
 6     kind="AlgorithmCode"
 7     efmiVersion="1.0.0"
 8     id="{1e111db5-90e6-4e17-b2e5-4e215dbbdd49}"
 9     name="PID controller discretized by Explicit Euler method"
10     version="0.1"
11     generationDateAndTime="2020-11-10T12:33:22Z"
12     generationTool="Manual"
13     license="MIT">
14
15     <Files>
16         <File
17             name="Controller.alg"
18             id="FileID_1"
19             path="./"
20             needsChecksum="false"
21             role="Code"/>
22     </Files>
23
24     <Clock id="ID_Clock" variableRefId="ID_7"/>
25
26     <BlockMethods fileRefId="FileID_1">
27         <BlockMethod id="ID_Startup" kind="Startup"/>
28         <BlockMethod id="ID_Recalibrate" kind="Recalibrate"/>
29         <BlockMethod id="ID_DoStep" kind="DoStep"/>
30     </BlockMethods>
31
32     <ErrorSignalStatus id="ID_ErrorSignalStatus"/>
```

```
33
34      <Variables>
35          <RealVariable
36              name="'limiter.uMin'"
37              id="ID_1"
38              blockCausality="dependentParameter"
39              start="-400.0"
40              min="-1.0e5"
41              max="-1.0"/>
42          <RealVariable
43              name="'limiter.uMax'"
44              id="ID_2"
45              blockCausality="tunableParameter"
46              start="400.0"
47              min="1.0"
48              max="1.0e5"/>
49          <RealVariable
50              name="Ti"
51              id="ID_3"
52              blockCausality="tunableParameter"
53              start="0.1"
54              min="1.0e-7"
55              max="100.0"/>
56          <RealVariable
57              name="Td"
58              id="ID_4"
59              blockCausality="tunableParameter"
60              start="0.1"
61              min="1.0e-7"
62              max="100.0"/>
63          <RealVariable
64              name="kd"
65              id="ID_5"
66              blockCausality="tunableParameter"
67              start="0.1"
68              min="0.0"
69              max="1000.0"/>
70          <RealVariable
71              name="k"
72              id="ID_6"
73              blockCausality="tunableParameter"
74              start="10.0"
75              min="0.0"
76              max="1000.0"/>
77          <RealVariable
78              name="stepSize"
79              id="ID_7"
80              blockCausality="tunableParameter"
81              start="1e-3"
82              min="1.0e-10"
83              max="0.01"/>
84          <RealVariable
85              name="gearRatio"
86              id="ID_8"
87              blockCausality="tunableParameter"
88              start="105.0"
89              min="10.0"
90              max="500.0"/>
```

```
 91          <RealVariable
 92              name="wLoadRef"
 93              id="ID_9"
 94              blockCausality="input"
 95              start="0.0"
 96              min="-1.0e5"
 97              max="1.0e5">
 98          </RealVariable>
 99          <RealVariable
100              name="wMotor"
101              id="ID_10"
102              blockCausality="input"
103              start="0.0"
104              min="-1.0e5"
105              max="1.0e5">
106          </RealVariable>
107          <RealVariable
108              name="vMotor"
109              id="ID_11"
110              blockCausality="output"
111              start="0.0"
112              min="-1.0e7"
113              max="1.0e7">
114          </RealVariable>
115          <RealVariable
116              name="'PID.I.x'"
117              id="ID_12"
118              blockCausality="state"
119              start="0.0"/>
120          <RealVariable
121              name="'PID.D.x'"
122              id="ID_13"
123              blockCausality="state"
124              start="0.0"/>
125          <RealVariable
126              name="'previous(feedback.y)'"
127              id="ID_14"
128              blockCausality="state"
129              start="0.0"/>
130          <BooleanVariable
131              name="firstTick"
132              id="ID_15"
133              blockCausality="state"
134              start="true"/>
135      </Variables>
136
137 </Manifest>
```

Mathematical Example using builtin Functions

The following example implements a linearly implicit second order differential equation system of the form $M(x)*x'' = F(x,u)$, $y = g(x)$ with an invertible matrix $M(x)$ for a state vector x, inputs u and outputs y. The vector functions F and g describe the right hand sides of the dynamical system and the output equation respectively.

The following implementation in eFMI GALEC code is based on a discretization by the Explicit Euler

method. Further, there are several expressions in *M* and *F* that use builtin functions like `sin`, `cos` and `exp`. Additionally, the builtin function `solveLinearEquations` is used to solve the linear system of equations. The respective eFMI GALEC program is:

```
1 block LinearEquationSystem
2     input Real u[4] (min=-1.0e7, max=1.0e7);
3     output Real y[4];
4
5 protected
6     // Constants:
7     constant Real pi;
8     constant Real stepSize;
9
10    // Discrete states:
11    Real x[4];
12    Real v[4];
13    Real 'derivative(x)'[4];
14    Real 'derivative(v)'[4];
15
16
17 public
18    /*
19        Startup function: Called once at startup to initialize the
20          internal memory of the block and return initial outputs.
21    */
22    method Startup
23    algorithm
24        // Initialize constants
25        self.pi := 3.141592653589793;
26        self.stepSize := 1.0e-2;
27
28        // Initialize discrete states:
29        self.x := {-3.0, 7.0, 19.0, 1.0};
30        self.v := {0.0, 0.0, 0.0, 0.0};
31
32        // Initial values for derivatives:
33        self.'derivative(x)' := {0.0, 0.0, 0.0, 0.0};
34        self.'derivative(v)' := {0.0, 0.0, 0.0, 0.0};
35
36        // Return initial control-outputs:
37        self.y := {0.0, 0.0, 0.0, 0.0};
38    end Startup;
39
40
41    method Recalibrate
42    algorithm
43    end Recalibrate;
44
45
46    /*
47        Control-cycle function: Called at every clock tick.
48    */
49    method DoStep
50    protected
51        Real M[4,4];
52        Real F[4];
```

```
53
54    algorithm
55        self.x := self.x + self.stepSize * self.'derivative(x)';
56        self.v := self.v + self.stepSize * self.'derivative(v)';
57
58        self.y := {
59            sin(self.x[1]) + self.x[3],
60            -self.x[2],
61            self.pi * 2.0 * cos(self.x[4] - self.x[2]),
62            self.x[3] + self.x[1] / self.x[4]
63        };
64
65        // Check for NaN, e.g. if there was no solution of the linear system in the previous
   call
66        if isNaN(self.y[1]) or isNaN(self.y[2]) or isNaN(self.y[3]) or isNaN(self.y[4]) then
67            // Re-initialize the whole system to its start state
68            self.x := {-3.0, 7.0, 19.0, 1.0};
69            self.v := {0.0, 0.0, 0.0, 0.0};
70            self.y := {0.0, 0.0, 0.0, 0.0};
71        end if;
72
73        M := {
74            {
75                -sin(self.x[3] + self.x[4]),
76                self.x[4]^2 - self.x[2]^3,
77                -4.0 * exp(self.x[3] * self.x[1]),
78                cos(-self.x[2]) * self.x[3]
79            },
80            {
81                (self.x[2] + 2.0 * self.x[4]) / self.x[1],
82                -self.x[1],
83                self.x[1] * self.x[2],
84                sin(self.x[1] * self.x[2] * self.x[3])
85            },
86            {
87                -self.x[4] + self.x[2] * self.x[1],
88                6.0 * self.pi * cos(self.x[2]),
89                -self.x[2],
90                2.0 * (self.x[1] + sin(self.x[3] * self.pi))
91            },
92            {
93                self.x[1]+cos(self.x[3]),
94                -2.0*self.x[3]*self.x[4],
95                -4.0 * self.x[3] * cos(self.x[2]),
96                self.x[4] - self.x[1] * self.x[2]
97            }
98        };
99        F := {
100            self.u[1] - self.x[3]^2,
101            -self.u[4] + self.x[2] * cos(self.x[1]),
102            -self.u[4] + self.u[2] * self.x[4],
103            self.u[2] + self.u[3]
104        };
105
106        self.'derivative(v)' := solveLinearEquations(M, F);
107        self.'derivative(x)' := self.v;
108
109    end DoStep;
```

```
110 end LinearEquationSystem;
```

The manifest summarising the controller's interface is:

```xml
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Manifest
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:noNamespaceSchemaLocation="../schemas/AlgorithmCode/efmiAlgorithmCodeManifest.xsd"
5      efmiVersion="1.0.0"
6      xsdVersion="0.14.0"
7      id="{351131cd-1e50-46d0-913a-240451d247c7}"
8      kind="AlgorithmCode"
9      name="Dynamic system discretized by Explicit Euler method"
10     generationDateAndTime="2020-10-15T16:49:20Z"
11     version="0.4.0"
12     generationTool="Manual"
13     license="MIT">
14
15     <Files>
16         <File
17             name="Controller.alg"
18             id="FileID_1"
19             path="./"
20             needsChecksum="false"
21             role="Code"/>
22     </Files>
23
24     <Clock id="ID_Clock" variableRefId="ID_2"/>
25
26     <BlockMethods fileRefId="FileID_1">
27         <BlockMethod id="ID_Startup" kind="Startup"/>
28         <BlockMethod id="ID_Recalibrate" kind="Recalibrate"/>
29         <BlockMethod id="ID_DoStep" kind="DoStep"/>
30     </BlockMethods>
31
32     <ErrorSignalStatus id="ID_ErrorSignal"/>
33
34     <Variables>
35         <RealVariable
36             name="pi"
37             id="ID_1"
38             blockCausality="constant"
39             start="3.141592653589793"/>
40         <RealVariable
41             name="stepSize"
42             id="ID_2"
43             blockCausality="constant"
44             start="1e-2"/>
45         <RealVariable
46             name="u"
47             id="ID_3"
48             blockCausality="input"
49             start="0.0 0.0 0.0 0.0"
50             min="-1.0e7"
51             max="1.0e7">
52             <Dimensions>
```

```
53              <Dimension number="1" size="4"/>
54            </Dimensions>
55          </RealVariable>
56          <RealVariable
57            name="y"
58            id="ID_4"
59            blockCausality="output"
60            start="0.0 0.0 0.0 0.0">
61            <Dimensions>
62              <Dimension number="1" size="4"/>
63            </Dimensions>
64          </RealVariable>
65          <RealVariable
66            name="v"
67            id="ID_5"
68            blockCausality="state"
69            start="0.0 0.0 0.0 0.0">
70            <Dimensions>
71              <Dimension number="1" size="4"/>
72            </Dimensions>
73          </RealVariable>
74          <RealVariable
75            name="x"
76            id="ID_6"
77            blockCausality="state"
78            start="-3.0 7.0 19.0 1.0">
79            <Dimensions>
80              <Dimension number="1" size="4"/>
81            </Dimensions>
82          </RealVariable>
83          <RealVariable
84            name="'derivative(x)'"
85            id="ID_7"
86            blockCausality="state"
87            start="0.0 0.0 0.0 0.0">
88            <Dimensions>
89              <Dimension number="1" size="4"/>
90            </Dimensions>
91          </RealVariable>
92          <RealVariable
93            name="'derivative(v)'"
94            id="ID_8"
95            blockCausality="state"
96            start="0.0 0.0 0.0 0.0">
97            <Dimensions>
98              <Dimension number="1" size="4"/>
99            </Dimensions>
100         </RealVariable>
101       </Variables>
102
103 </Manifest>
```

Vehicle model with implicit integration method

The following example presents a discretized vehicle model. The model equations and parameters are according to *Section 6.8 Rollover Avoidance* of the book *J. Ackermann et al.: Robust Control,*

Springer 2002 with some further assumptions. The vehicle model is a single track model with roll augmentation. The discretization is realized by a linear implicit Runge-Kutta method of order 1 (Rosenbrock method, linear implicit Euler method) suited for stiff systems. For such methods the input signals have to be differentiated, therefore the derivatives of the original input variables are added as inputs of the discretized model.

The example demonstrates the use of for-loops, vectors and matrices as well as several builtin functions, particularly for solving linear equation systems. The eFMI GALEC program is:

```
 1 block VehicleModel
 2     input Real u[2](min=-1.0e7, max=1.0e7);
 3     input Real 'derivative(u)'[2](min=-1.0e7, max=1.0e7);
 4     output Real x[8];
 5
 6     // Tunable parameters (can be changed via recalibration):
 7     parameter Real FdF;
 8     parameter Real m;
 9     parameter Real m2;
10     parameter Real h;
11     parameter Real lF;
12     parameter Real lR;
13     parameter Real g;
14     parameter Real Jx2;
15     parameter Real mu;
16     parameter Real cF;
17     parameter Real cR;
18     parameter Real Jz1;
19     parameter Real Jz2;
20     parameter Real Jy2;
21     parameter Real cphi;
22     parameter Real dphidot;
23     parameter Real b1;
24     parameter Real b2;
25     parameter Real stepSize;
26
27 protected
28     // Dependent parameters:
29     parameter Real FlV;
30     parameter Real FzR;
31     parameter Real FzF;
32
33     // Discrete states:
34     Real q[4];
35     Real dx[8];
36
37 public
38     /*
39         Startup function: Called once at startup to initialize the
40          internal memory of the block and return initial outputs.
41     */
42     method Startup
43     algorithm
44         // Initialize tunable parameters
45         self.FdF := 15.0;
46         self.m := 14300.0;
47         self.m2 := 12487.0;
```

```
 48         self.h := 1.15;
 49         self.lF := 1.95;
 50         self.lR := 1.54;
 51         self.g := 9.81;
 52         self.Jx2 := 24201.0;
 53         self.mu := 1.0;
 54         self.cF := 582.0e+3;
 55         self.cR := 783.0e3;
 56         self.Jz1 := 3654.0;
 57         self.Jz2 := 34917.0;
 58         self.Jy2 := 3491.7;
 59         self.cphi := 457.0e+3;
 60         self.dphidot := 100.0e3;
 61         self.b1 := 0.2;
 62         self.b2 := 0.1;
 63         self.stepSize := 1.0e-2;
 64
 65         // Initialize dependent parameters
 66         self.FlV := self.FdF;
 67         self.FzR := self.m*self.g*self.lF/(self.lR + self.lF);
 68         self.FzF := self.m*self.g - self.FzR;
 69
 70         // Initialize inputs
 71         // u = {0.0, 0.0};
 72         // 'derivative(u)' = {0.0, 0.0};
 73
 74         // Initialize states and outputs
 75         self.q := {0.0, 0.0, 0.0, 0.0};
 76         self.dx := {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
 77         self.x := {0.0, 0.0, 0.0, 0.0, 10.0, 0.0, 0.0, 0.0};
 78     end Startup;
 79
 80     /*
 81         Recalibration function: Called to change tunable parameters
 82           during operation.
 83     */
 84     method Recalibrate
 85     algorithm
 86         // Update dependent parameters:
 87         self.FlV := self.FdF;
 88         self.FzR := self.m*self.g*self.lF/(self.lR + self.lF);
 89         self.FzF := self.m*self.g - self.FzR;
 90     end Recalibrate;
 91
 92     /*
 93         Control-cycle function: Called at every clock tick.
 94     */
 95     method DoStep
 96     protected
 97         Real sx;
 98         Real sy;
 99         Real psi;
100         Real phi;
101         Real vx;
102         Real vy;
103         Real r;
104         Real phidot;
105         Real delta;
```

```
106        Real FyD;
107        Real q1;
108        Real q2;
109        Real q3;
110        Real q4;
111        Real deltadot;
112        Real FyDdot;
113
114        Real FdF;
115        Real FlV;
116        Real m;
117        Real m2;
118        Real h;
119        Real lF;
120        Real lR;
121        Real g;
122        Real Jx2;
123        Real mu;
124        Real cF;
125        Real cR;
126        Real Jz1;
127        Real Jz2;
128        Real Jy2;
129        Real FzR;
130        Real FzF;
131        Real cphi;
132        Real dphidot;
133        Real b1;
134        Real b2;
135
136        Real G[4,4];
137
138        Real rs2[4];
139        Real dx1[4];
140
141        Real help1;
142        Real help2;
143        Real help3;
144
145    algorithm
146
147        for i in 1:8 loop
148            self.x[i] := self.x[i] + self.dx[i];
149        end for;
150
151        for i in 1:4 loop
152            self.q[i] := self.dx[4+i]/self.stepSize;
153        end for;
154
155        sx    := self.x[1];
156        sy    := self.x[2];
157        psi   := self.x[3];
158        phi   := self.x[4];
159        vx    := self.x[5];
160        vy    := self.x[6];
161        r     := self.x[7];
162        phidot := self.x[8];
163
```

```
164        delta := self.u[1];
165        FyD   := self.u[2];
166
167        q1    := self.q[1];
168        q2    := self.q[2];
169        q3    := self.q[3];
170        q4    := self.q[4];
171
172        deltadot := self.'derivative(u)'[1];
173        FyDdot   := self.'derivative(u)'[2];
174
175        FdF := self.FdF;
176        FlV := self.FlV;
177
178        m   := self.m;
179        m2  := self.m2;
180        h   := self.h;
181        lF  := self.lF;
182        lR  := self.lR;
183        g   := self.g;
184        Jx2 := self.Jx2;
185
186        mu := self.mu;
187        cF := self.cF;
188        cR := self.cR;
189
190        Jz1    := self.Jz1;
191        Jz2    := self.Jz2;
192        Jy2    := self.Jy2;
193        FzR    := self.FzR;
194        FzF    := self.FzF;
195        cphi   := self.cphi;
196        dphidot := self.dphidot;
197
198        b1 := self.b1;
199        b2 := self.b2;
200
201        help1 := sqrt(vx^2 + vy^2);
202        help2 := (vx^2 + vy^2)^1.5;
203        help3 := h^2*m2 + Jy2 - Jz2;
204
205        G[1,1] :=
206            (
207                  mu*(lF*r*vx + help1*vy)*self.stepSize*cF*sin(delta)
208               + help2*m
209            )
210            / (help2*self.stepSize);
211        G[1,2] :=
212            -(
213                  mu*(-lF*r*vy + help1*vx)*cF*sin(delta)
214               + help2*r*m
215            )
216            / help2;
217        G[1,3] :=
218            (
219                  2.0*h*m2*phidot*cos(phi)*self.stepSize*help1
220               - mu*cF*lF*sin(delta)*self.stepSize
221               + h*m2*sin(phi)*help1
```

```
222                 - m*vy*self.stepSize*help1
223             )
224             / (self.stepSize*help1);
225         G[1,4] :=
226             h*m2*(
227                     -2.0*sin(phi)*phidot*r*self.stepSize
228                 + cos(phi)*q3*self.stepSize
229                 + 2.0*r*cos(phi)
230             );
231         G[2,1] :=
232             (
233                     (-cos(delta)*cF*mu*vy - cR*mu*vy + m*r*(vx^2+vy^2))*help1
234                 - r*mu*vx*(cos(delta)*cF*lF - cR*lR)
235             )
236             / help2;
237         G[2,2] :=
238             (
239                     (cos(delta)*cF*mu*vx*self.stepSize + cR*mu*vx*self.stepSize + m*(vx^2 +
     vy^2))*help1
240                 - self.stepSize*r*mu*vy*(cos(delta)*cF*lF - cR*lR)
241             )
242             / (help2*self.stepSize);
243         G[2,3] :=
244             (
245                     2.0*h*m2*r*sin(phi)*help1
246                 + mu*cF*lF*cos(delta)
247                 + m*vx*help1
248                 - mu*cR*lR
249             )
250             / help1;
251         G[2,4] :=
252             m2*(
253                     (-1.0 + (phidot^2 + r^2)*self.stepSize^2)*cos(phi)
254                 + self.stepSize*sin(phi)*(q4*self.stepSize + 2.0*phidot)
255             )
256             * (h/self.stepSize);
257         G[3,1] :=
258             (
259                 (
260                         -cos(delta)*cF*lF*mu*vy*self.stepSize
261                     + h*m2*(vx^2 + vy^2)*sin(phi)
262                     + cR*lR*mu*vy*self.stepSize
263                 ) * help1
264                 - self.stepSize*r*mu*vx*(lF^2*cF*cos(delta) + lR^2*cR)
265             )
266             / (help2*self.stepSize);
267         G[3,2] :=
268             -(
269                     (-cos(delta)*cF*lF*mu*vx + h*r*m2*(vx^2 + vy^2)*sin(phi) + cR*lR*mu*vx)*help1
270                 + vy*r*mu*(lF^2*cF*cos(delta) + lR^2*cR)
271             )
272             / help2;
273         G[3,3] :=
274             2.0*(
275                 (
276                         (-0.5*h^2*m2 - 0.5*Jy2 + 0.5*Jz2)*cos(phi)^2
277                     + phidot*self.stepSize*sin(phi)*help3*cos(phi)
278                     - 0.5*sin(phi)*h*m2*vy*self.stepSize
```

```
279                     + 0.5*h^2*m2
280                     + 0.5*Jy2
281                     + 0.5*Jz1
282                 ) * help1
283                 + 0.5*mu*self.stepSize*(lF^2*cF*cos(delta) + lR^2*cR)
284             )
285             / (help1*self.stepSize);
286     G[3,4] :=
287             4.0*phidot*self.stepSize*r*help3*cos(phi)^2
288         + (2.0*help3*(q3*self.stepSize + r)*sin(phi) - h*self.stepSize*m2*(r*vy -
   q1))*cos(phi)
289             - 2.0*phidot*self.stepSize*r*help3;
290     G[4,1] := -h*m2*r*cos(phi);
291     G[4,2] := -h*m2*cos(phi) / self.stepSize;
292     G[4,3] := -2.0*(help3*r*sin(phi) + 0.5*h*m2*vx)*cos(phi);
293     G[4,4] :=
294         (
295                 -2.0*self.stepSize^2*r^2*help3*cos(phi)^2
296             - cos(phi)*g*h*m2*self.stepSize^2
297             + self.stepSize^2*h*m2*(r*vx + q2)*sin(phi)
298             + (help3*r^2 + cphi)*self.stepSize^2
299             + dphidot*self.stepSize
300             + h^2*m2
301             + Jx2
302         )
303         / self.stepSize;
304
305     rs2[1] :=
306         2.0*(
307             (
308                     -0.5*self.stepSize*(cF*mu*(delta - atan2(vy, vx))*cos(delta) +
   sin(delta)*(cF*mu + FlV))*deltadot
309                 + 0.5*atan2(vy, vx)*sin(delta)*cF*mu
310                 - 0.5*sin(delta)*cF*delta*mu
311                 - 0.5*h*m2*phidot*(q3*self.stepSize + 2.0*r)*cos(phi)
312                 + 0.5*FlV*cos(delta)
313                 + r*(sin(phi)*h*m2*phidot^2*self.stepSize + 0.5*m*vy)
314             ) * help1
315             + 0.5*cF*lF*mu*r*(deltadot*cos(delta)*self.stepSize + sin(delta))
316         )
317         / help1;
318     rs2[2] :=
319         -(
320             (
321                 (
322                     mu*cF*(delta - atan2(vy, vx))*sin(delta)
323                     - cos(delta)*(cF*mu + FlV)
324                 ) * (self.stepSize*deltadot)
325                 - FyDdot*self.stepSize
326                 + mu*(cos(delta)*cF + cR)*atan2(vy, vx)
327                 - cos(delta)*cF*delta*mu
328                 + h*m2*(phidot*q4*self.stepSize + phidot^2+r^2)*sin(phi)
329                 + h*phidot*self.stepSize*m2*(phidot^2 + r^2)*cos(phi)
330                 + m*r*vx
331                 - FlV*sin(delta)
332                 - FyD
333             ) * help1
334             - mu*r*(self.stepSize*sin(delta)*deltadot*cF*lF - cos(delta)*cF*lF + cR*lR)
```

```
335              )
336              / help1;
337         rs2[3] :=
338             -2.0*(
339                 (
340                     0.5*lF*(mu*cF*(delta - atan2(vy, vx))*sin(delta) - cos(delta)*(cF*mu +
     FlV))*self.stepSize*deltadot
341                     - 0.5*FyDdot*b1*self.stepSize
342                     + 0.5*mu*(cos(delta)*cF*lF - cR*lR)*atan2(vy, vx)
343                     + 2.0*r*phidot^2*self.stepSize*help3*cos(phi)^2
344                     + phidot*(help3*(q3*self.stepSize + r)*sin(phi) + 0.5*h*self.stepSize*m2*(-
     r*vy + q1))*cos(phi)
345                     - 0.5*sin(phi)*h*m2*r*vy
346                     - 0.5*cos(delta)*cF*delta*lF*mu
347                     - 0.5*FlV*sin(delta)*lF
348                     - r*phidot^2*self.stepSize*help3
349                     - 0.5*b1*FyD
350                 ) * help1
351                 - 0.5*r*mu*(deltadot*sin(delta)*cF*lF^2*self.stepSize - lF^2*cF*cos(delta) -
     lR^2*cR)
352             )
353             / help1;
354         rs2[4] :=
355              self.stepSize*b2*FyDdot
356             + 2.0*phidot*self.stepSize*r^2*help3*cos(phi)^2
357             + (r^2*help3*sin(phi) + h*m2*(g*phidot*self.stepSize + r*vx))*cos(phi)
358             + (-phidot*(r*vx+q2)*self.stepSize + g)*h*m2*sin(phi)
359             - phidot*(help3*r^2 + cphi)*self.stepSize
360             - cphi*phi
361             - dphidot*phidot
362             + b2*FyD;
363
364         dx1 := solveLinearEquations(G, rs2);
365         for i in 1:4 loop
366             self.dx[4+i] := dx1[i];
367             self.dx[i] := self.stepSize*(self.x[4+i]+dx1[i]);
368         end for;
369
370         // Check for NaN, caused by e.g. a failed solution of the linear system
371         if isNaN(self.x[1]) or isNaN(self.x[2]) or isNaN(self.x[3]) or isNaN(self.x[4]) or
372             isNaN(self.x[5]) or isNaN(self.x[6]) or isNaN(self.x[7]) or isNaN(self.x[8]) then
373             self.q := {0.0, 0.0, 0.0, 0.0};
374             self.dx := {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
375             self.x := {0.0, 0.0, 0.0, 0.0, 10.0, 0.0, 0.0, 0.0};
376         end if;
377
378     end DoStep;
379 end VehicleModel;
```

The resulting manifest is:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Manifest efmiVersion="1.0.0"
3          generationDateAndTime="2020-10-15T16:52:13Z"
4          generationTool="Manual"
5          id="{e3eae104-6417-4783-8c05-7c14e6fab8a6}"
```

```
 6            kind="AlgorithmCode"
 7            license="MIT"
 8            name="Vehicle model discretized by Linearly implicit Euler method"
 9            version="0.2"
10            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
11            xsdVersion="0.14.0"
12
   xsi:noNamespaceSchemaLocation="../schemas/AlgorithmCode/efmiAlgorithmCodeManifest.xsd">
13    <Files>
14      <File
15            id="FileID_1"
16            name="Controller.alg"
17            needsChecksum="false"
18            path="./"
19            role="Code" />
20    </Files>
21    <Clock id="ID_Clock" variableRefId="ID_1" />
22    <BlockMethods fileRefId="FileID_1">
23      <BlockMethod id="ID_Startup" kind="Startup" />
24      <BlockMethod id="ID_DoStep" kind="DoStep" />
25      <BlockMethod id="ID_Recalibrate" kind="Recalibrate" />
26    </BlockMethods>
27    <ErrorSignalStatus id="ID_ErrorSignal"/>
28    <Variables>
29      <RealVariable blockCausality="tunableParameter"
30            id="ID_1"
31            name="stepSize"
32            start="1e-2" />
33      <RealVariable blockCausality="tunableParameter"
34            id="ID_2"
35            name="FdF"
36            start="15.0" />
37      <RealVariable blockCausality="dependentParameter"
38            id="ID_3"
39            name="FlV"
40            start="15.0" />
41      <RealVariable blockCausality="tunableParameter"
42            id="ID_4"
43            name="m"
44            start="14300.0" />
45      <RealVariable blockCausality="tunableParameter"
46            id="ID_5"
47            name="m2"
48            start="12487.0" />
49      <RealVariable blockCausality="tunableParameter"
50            id="ID_6"
51            name="h"
52            start="1.15" />
53      <RealVariable blockCausality="tunableParameter"
54            id="ID_7"
55            name="lF"
56            start="1.95" />
57      <RealVariable blockCausality="tunableParameter"
58            id="ID_8"
59            name="lR"
60            start="1.54" />
61      <RealVariable blockCausality="tunableParameter"
62            id="ID_9"
```

```xml
            name="g"
            start="9.81" />
      <RealVariable blockCausality="tunableParameter"
            id="ID_10"
            name="Jx2"
            start="24201.0" />
      <RealVariable blockCausality="tunableParameter"
            id="ID_11"
            name="mu"
            start="1.0" />
      <RealVariable blockCausality="tunableParameter"
            id="ID_12"
            name="cF"
            start="582e3" />
      <RealVariable blockCausality="tunableParameter"
            id="ID_13"
            name="cR"
            start="783e3" />
      <RealVariable blockCausality="tunableParameter"
            id="ID_14"
            name="Jz1"
            start="3654.0" />
      <RealVariable blockCausality="tunableParameter"
            id="ID_15"
            name="Jz2"
            start="34917.0" />
      <RealVariable blockCausality="tunableParameter"
            id="ID_16"
            name="Jy2"
            start="3491.7" />
      <RealVariable blockCausality="dependentParameter"
            id="ID_17"
            name="FzR"
            start="0.0" />
      <RealVariable blockCausality="dependentParameter"
            id="ID_18"
            name="FzF"
            start="0.0" />
      <RealVariable blockCausality="tunableParameter"
            id="ID_19"
            name="cphi"
            start="457.0e+3" />
      <RealVariable blockCausality="tunableParameter"
            id="ID_20"
            name="dphidot"
            start="100.0e3" />
      <RealVariable blockCausality="tunableParameter"
            id="ID_21"
            name="b1"
            start="0.2" />
      <RealVariable blockCausality="tunableParameter"
            id="ID_22"
            name="b2"
            start="0.1" />
      <RealVariable blockCausality="input"
            id="ID_23"
            name="u"
            start="0.0 0.0"
```

```
121            min="-1.0e7"
122            max="1.0e7">
123        <Dimensions>
124          <Dimension number="1"
125                   size="2" />
126        </Dimensions>
127      </RealVariable>
128      <RealVariable blockCausality="input"
129            id="ID_24"
130            name="&#39;derivative(u)&#39;"
131            start="0.0 0.0"
132            min="-1.0e7"
133            max="1.0e7">
134        <Dimensions>
135          <Dimension number="1"
136                   size="2" />
137        </Dimensions>
138      </RealVariable>
139      <RealVariable blockCausality="output"
140            id="ID_25"
141            name="x"
142            start="0.0 0.0 0.0 0.0 10.0 0.0 0.0 0.0">
143        <Dimensions>
144          <Dimension number="1"
145                   size="8" />
146        </Dimensions>
147      </RealVariable>
148      <RealVariable blockCausality="state"
149            id="ID_26"
150            name="q"
151            start="0.0 0.0 0.0 0.0">
152        <Dimensions>
153          <Dimension number="1"
154                   size="4" />
155        </Dimensions>
156      </RealVariable>
157      <RealVariable blockCausality="state"
158            id="ID_27"
159            name="dx"
160            start="0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0">
161        <Dimensions>
162          <Dimension number="1"
163                   size="8" />
164        </Dimensions>
165      </RealVariable>
166    </Variables>
167 </Manifest>
```

[1] I.e., after detecting an error, normal program execution is suspended until the error is handled and the current control-cycle terminated with the error signaled

[2] Only the *bounded-iteration* rule has *loop-iterator-declaration* within its definition-list.

Chapter 4. Behavioral Model Representation

The optional *Behavioral Model* representation defines test scenarios with reference trajectories to allow automatic verification of other model representations like Algorithm, Production or Binary Code representations. Reference trajectories are stored within a Behavioral Model container in comma-separated values (CSV) files — so called reference trajectories files. A single Behavioral Model container can have several reference trajectories files, used for varying test scenarios described in its manifest. Besides the test scenarios backed by the container's reference trajectories files, the manifest defines acceptable tolerances (absolute and relative), type and dimensionality (any GALEC type of any dimensionality, i.e., boolean, integer or real scalar, vector, matrix or an even higher dimensionality), causality (any GALEC block-interface causality, i.e., input, output, tunable parameter or dependent parameter) and unit of each reference trajectory. Each test scenario thereby defines an open loop test for the GALEC block — and any model representations derived from it — of an Algorithm Code container of the eFMU.

To that end, reference trajectories in Behavioral Model manifests are linked to the GALEC block variables of an Algorithm Code manifest for which they define reference values. Other containers derived from the same algorithmic solution — like Production or Binary Code containers — can in turn be tested by matching their own tracelinks to GALEC block variables with the ones of the Behavioral Model manifest. This indirect linking of reference tests with tested components via an Algorithm Code manifest provides an universal test framework where different, independently designed Behavioral Models can be used to test independently developed executable model representations.

Two types of test scenarios are distinguished, such with variable and such with equidistant time grid. Variable time grid scenarios do not restrict by any means the time trajectory of reference trajectories; the time distance between two values of a reference trajectory (two reference points) is variable. As a consequence, application of such scenarios for testing sampled data systems (like production code) requires interpolation if reference points are not exactly at sampling points. Equidistant time grid scenarios on the other hand define reference points in equidistant intervals; interpolation is not required. Variable time grids are more common in the continuous physics simulation world, and offer higher flexibility at the potential cost of precision due to interpolation, whereas equidistant time grids are the norm for hardware-in-the-loop (HiL) tests in the embedded domain. Although free of any interpolation issues, equidistant time grid scenarios are not as flexible when it comes to actual sampling trajectories used throughout testing; late stage sampling period changes, causing the actual sampling trajectory to misalign with the equidistant time trajectory of the reference results, are not supported.

Note that, reference trajectories themselves are not causalized. The actual sampling and causalization (i.e., are reference trajectories block inputs, outputs, tunable parameters, or dependent parameters) used for testing a GALEC block are from the linked Algorithm Code manifest, whereas the reference trajectories of a reference trajectories file can stem from varying modeling and simulation scenarios. Trajectories could be derived from whole system models where the tested block is coupled to plant models or simple open loop or table-driven test scenarios, either based on a continuous, a-causal simulation or a fixed step-size simulation with well-defined input/output causalities or any combination thereof. The eFMI Standard does not anticipate any specific tooling or simulation technique used for producing reference trajectories files. In case of variable time grid,

tooling used to generate Behavioral Model containers and their reference trajectories files should take care however, that the simulated behavior and respectively derived reference trajectories are of appropriate detail regarding tolerances and reference point density to replay the designed test scenario using a fixed step-size sampled algorithm (i.e., GALEC program and derived production and binary codes). To that end, it is recommended that originating models are numerical stable, including stability of used integration methods, the discontinuities of discrete or piecewise continuous trajectories are properly reflected and sufficient reference points for reasonable interpolation of continuous trajectories are provided.

Tolerances defined in the Behavioral Model manifest can help to overcome acceptable imprecisions. Supported are relative and absolute error tolerances for individual reference trajectories. Well-defined interpolation semantics for variable time grid scenarios ensure unique interpretation regarding applied values and whether actual results are within tolerance or not. Also the acceptable time tolerance between the sampling period of the Algorithm Code manifest and equidistant time grid scenarios is well-defined, as is the acceptable time tolerance between sampling trajectory and the time trajectory given in reference trajectories files.

The structure and content of the manifest describing an eFMI Behavioral Model container and its test scenarios are defined in Section 4.1. The final interpretation semantic of a test scenario defined in a Behavioral Model manifest — i.e., the semantic of reference trajectories files for testing other model representations — is defined in Section 4.2.

4.1. Behavioral Model manifest

Every eFMI Behavioral Model container is described by its manifest, an Extensible Markup Language (XML) document satisfying the XML Schema Definition (XSD) and rules given in the following.

§1: Encoding of manifests

Behavioral Model manifests must satisfy the *Extensible Markup Language (XML) 1.0 (Fifth Edition)* standard; and be *Unicode® Standard* conforming documents encoded in *UTF-8* without byte-order mark (*ISO/IEC 10646:2020* including *ISO/IEC 10646:2020/Amd 1:2023*). To denote these encoding restrictions, Behavioral Model manifests must start with `<?xml version="1.0" encoding="utf-8"?>`.

The XSD of eFMI Behavioral Model manifests, described in the following, is according to the *W3C XML Schema Definition Language (XSD) 1.1 Part 1 & 2* standard.

§2: General structure and content of manifests

The following figure summarizes the general structure of Behavioral Model manifests; the individual XML elements will be described in detail in the subsequent sections:
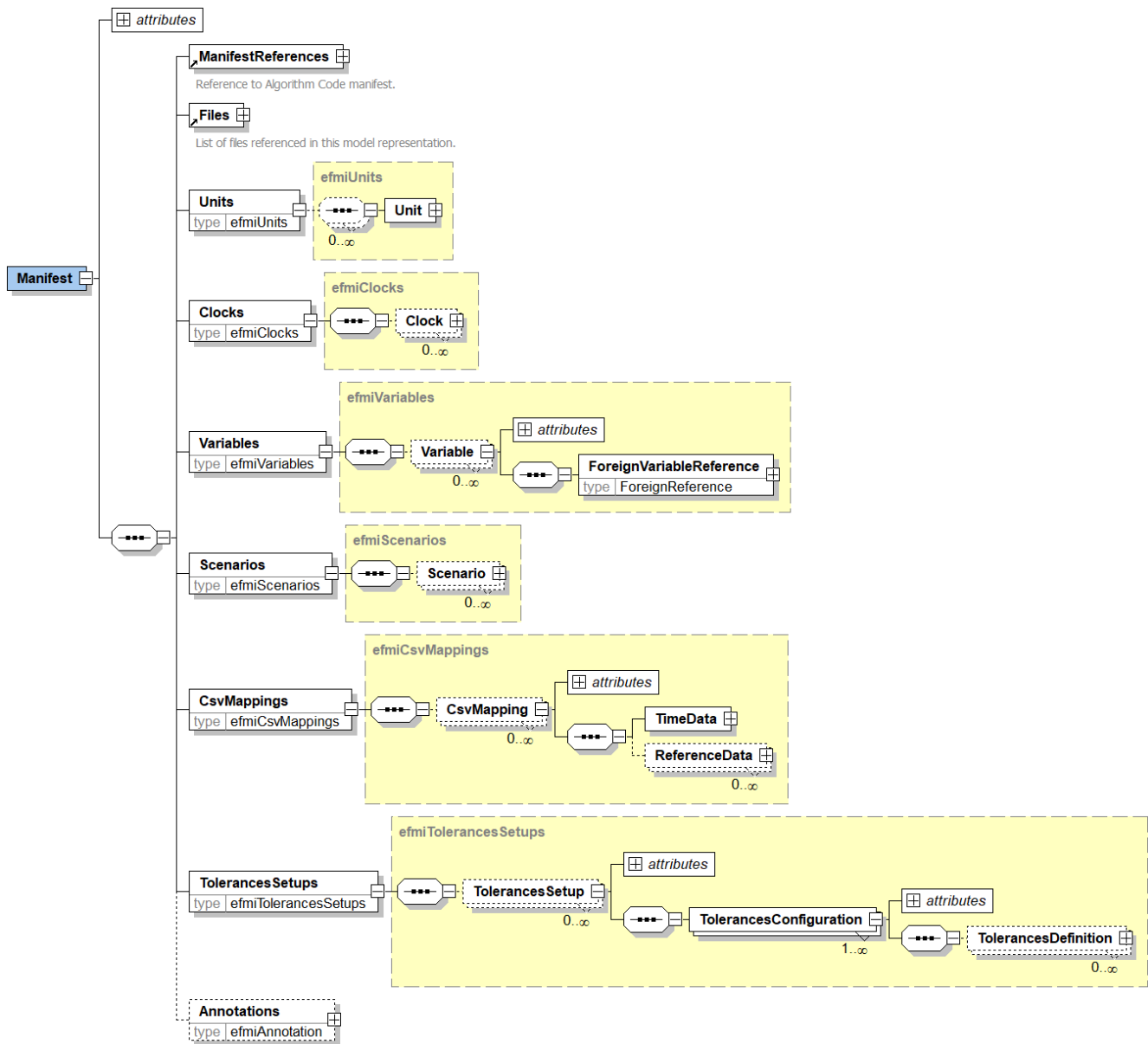
The focal point of the root `<Manifest>` element is the `<Scenarios>` list defining arbitrary many test scenarios, each an individual `<Scenario>` element. Each scenario compiles the assets defining it by referencing to them (cf. Section 2.3.4.1 about local manifest references); actual asset definitions are in separate lists such that different scenarios can share parts of their configuration. The assets a scenario refers to are:

- its reference trajectories file (`<File>` in `<Files>` list)

- its mapping configuration (`<CsvMapping>` in `<CsvMappings>` list), defining the mapping of the reference time grid and the element-index-wise reference trajectories of inputs, tunable parameters, dependent parameters and output variables to column names of the reference trajectories file

- its tolerances setup (`<TolerancesSetup>` in `<TolerancesSetups>` list), with error tolerances configurations (`<TolerancesConfiguration>`) for the different floating-point precisions the scenario supports

Each tolerance configuration defines the error tolerances to be used if tests are conducted with its floating-point precision. It defines error tolerances for all recalibrated tunable parameters and tested dependent parameters and outputs, each in an individual tolerances definition

(`<TolerancesDefinition>`). The resulting XML containment structure clearly defines which configuration a tolerance is defined for (parent `<TolerancesConfiguration>` of `<TolerancesDefinition>`) and which setup the configuration is part of (parent `<TolerancesSetup>` of `<TolerancesConfiguration>`). A tolerances setup therefore can be understood as the definition of acceptable error tolerances for a set of floating-point precisions, which define the precisions supported by scenarios referencing it; if it misses a configuration for some floating-point precision, scenarios referencing it cannot be used for testing eFMU containers based on that very precision.

Besides listing scenarios, reference trajectories files, mapping configurations and tolerance setups, Behavioral Model manifests also link variables back to their respective definition in the Algorithm Code manifest. To that end, all used variables are compiled in the `<Variables>` list, with each `<Variable>` element using a `<ForeignVariableReference>` to define the link (cf. Section 2.3.4.3 about foreign manifest references). The causality (input, tunable parameter, dependent parameter or output) of listed variables can be resolved from the Algorithm Code manifest via these backlinks. Behavioral Model manifests also compile the units (`<Units>` list) referenced from variables, tolerances for such, reference time grids and clocks of equidistant time grid scenarios. Since the mapping configuration of scenarios map variables to the columns of their reference trajectories file, the causalities and units of reference trajectories are well-defined (cf. §1.1 and §1.3 of Section 4.2). Likewise, Behavioral Model manifests compile the clocks referenced from equidistant time grid scenarios (`<Clocks>` list) to define their presupposed sampling period. All local units and presupposed sampling periods of a Behavioral Model manifest must be consistent with the Algorithm Code manifest M_A the Behavioral Model manifest refers to; only if so, model representations derived from M_A, like Production Code containers, can be tested (cf. Section 4.2 for the respective consistency rules).

> **R-1 (system integration of Behavioral Models and tested model representations in test environments):** The indirect linkage from a Behavioral Model manifest M_B via an Algorithm Code manifest M_A to other model representation manifests transitively referencing M_A — e.g., a Production or Binary Code manifest M_{PB} — defines what can be tested (everything indirectly, transitively and reflexively linked). The consistency rules of Section 4.2 between M_B and M_A ensure that M_{PB} (which is derived from M_A) can be tested with M_B. And the GALEC block-interface and life-cycle of Section 3.2.3 defines how the test must be instantiated and conducted (cf. §4 of Section 4.2 for details). The latter is particularly important, since only the GALEC block-interface of M_A is fixed, whereas the interface of derived production or binary code in M_{PB} is not; but a testing tool can resolve the actual production or binary code interface of M_{PB} and system integrate it into a test environment for testing M_{PB} with M_B by collecting the required information from the interlinked manifests.

§3: Root of manifest

The root element of a Behavioral Model manifest is the `<Manifest>` element, defined in `BehavioralModel/efmiBehavioralModelManifest.xsd`. It shares the optional `<Annotations>` list defined in Section 2.3.4.5 and the attributes defined in Section 2.3.1, which are common for the manifests of any eFMI model representation; its manifest kind specific attributes are `kind="BehavioralModel"`.

The XSD of `<Manifest>` is:
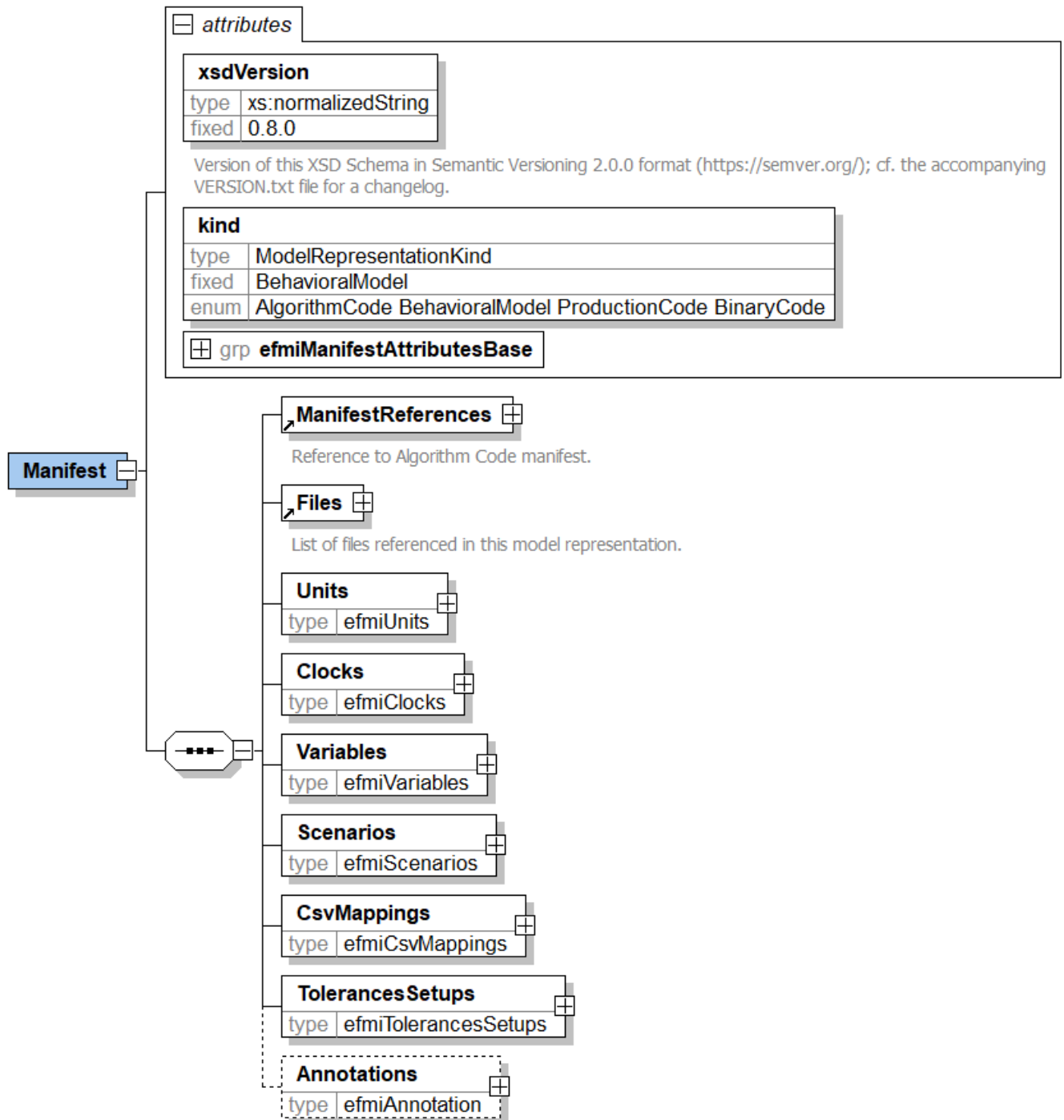
The Behavioral Model specific elements of `<Manifest>` are:

Name	Description
`<ManifestReferences>`	Foreign manifest references as defined in Section 2.3.4.3. In case of Behavioral Model manifests, a single reference to the manifest of the Algorithm Code for which this Behavioral Model defines test scenarios. The causalities of the block-interface variables, exposed error signals of block-interface methods and sampling period (called actual sampling period in the context of Behavioral Models) of the referenced Algorithm Code manifest are used to instantiate actual tests — with the test scenarios defined by this Behavioral Model — for the referenced Algorithm Code container itself or any Production or Binary Code container whose manifest transitively (i.e., directly or indirectly via a chain of references) refers to the very same Algorithm Code manifest.
`<Files>`	List of referenced files of the model representation as defined in Section 2.3.3. In case of Behavioral Model manifests, only container local files can be referenced and no `<ForeignFile>`. The files referenced (contained `<File>` elements) are the individual reference trajectories files of the Behavioral Model container; each must be in turn referenced by at least one test scenario.
`<Units>`	List of units as defined in Section 3.1.5. The defined units are referenced by the individual mappings of mapping configurations to define the unit of reference time grids and variable trajectories stored in reference trajectories files (cf. §5), tolerances definitions to define the unit of absolute tolerances (cf. §6), and the clock of equidistant time grid scenarios to define the unit of the presupposed sampling period. The units in between Behavioral Model entities and with their respective Algorithm Code entities must align according to §1.3 of Section 4.2.
`<Clocks>`	List of clocks referenced by equidistant time grid scenarios to define their presupposed sampling period and its time unit (`samplingPeriod` and `unitRefId` attributes of contained `<Clock>` elements). The presupposed sampling periods of clocks must be GALEC `Real` values and must align with the actual sampling period of the Algorithm Code manifest the Behavioral Model manifest refers to according to §2.2 of Section 4.2. The XSD of `<Clocks>` and `<Clock>` is defined in `BehavioralModel/efmiClocks.xsd`.
`<Variables>`	List of variables, each variable (contained `<Variable>` element) linked to an Algorithm Code manifest variable via a `<ForeignVariableReference>` (cf. Section 2.3.4.3 about foreign manifest references) to denote the GALEC variable it represents. The defined variables are referenced by tolerance definitions to denote which variable they define a tolerance for (cf. §6) and the individual mappings of mapping configurations to denote which variable they define a column mapping for (cf. §5). Variables must be linked to inputs, tunable parameters, dependent parameters or output variables of the Algorithm Code manifest; and each Algorithm Code manifest variable must be linked at most once. The XSD of `<Variables>` and `<Variable>` is defined in `BehavioralModel/efmiVariables.xsd`.
`<Scenarios>`	List of test scenarios defined by the Behavioral Model container (cf. §4).

Name	Description
`<CsvMappings>`	List of mapping configurations used by test scenarios to map the individual element-indexes of variables to column names of reference trajectories files (cf. §5).
`<TolerancesSetups>`	List of tolerance setups used by test scenarios to define error tolerances for tunable parameter, dependent parameter and output variables for different floating-point precisions (cf. §6).

§4: Test scenarios of manifest

The test scenarios of a Behavioral Model manifest are defined by its `<Scenarios>` list; each test scenario as an individual `<Scenario>` element.

The XSD of `<Scenarios>` (complex type `efmiScenarios`) and `<Scenario>` — defined in `BehavioralModel/efmiScenarios.xsd` — is:
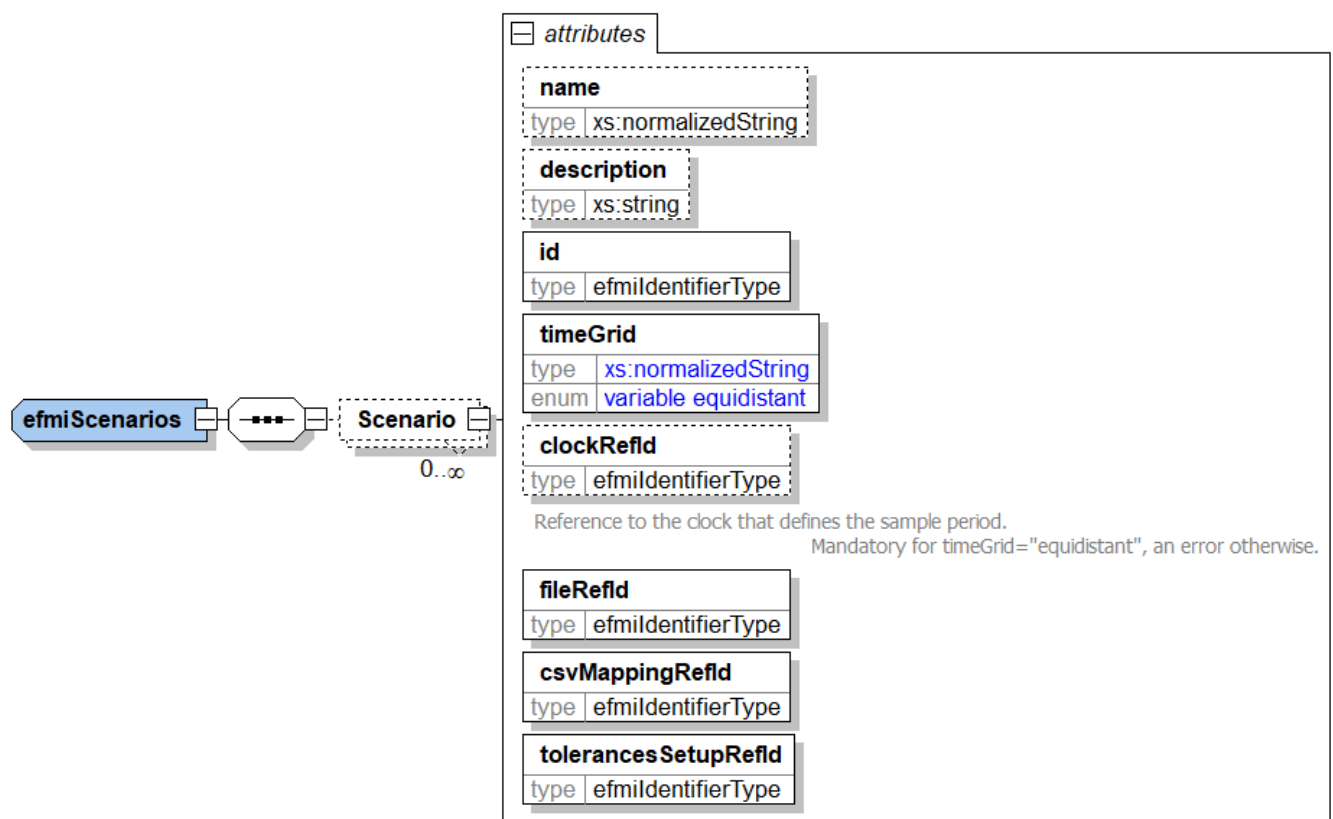
The attributes of `<Scenario>` are:

Name	Description
`name`	Optional scenario name.
`description`	Optional scenario description.
`timeGrid`	Scenario type; either `"variable"` or `"equidistant"` denoting that the scenario is a variable or equidistant time grid scenario respectively.

Name	Description
`clockRefId`	Optional reference to the clock defining the scenario's presupposed sampling period and its time unit. Must be given, if, and only if, the scenario is an equidistant time grid scenario.
`fileRefId`	Reference to the `<File>` element defining the scenario's reference trajectories file.
`csvMappingRefId`	Reference to the scenario's mapping configuration (cf. §5).
`tolerancesSetupRefId`	Reference to the scenario's tolerances setup (cf. §6).

R-2 (design rationale for the separate definition of reference trajectories files, variables, clocks, mapping configurations and tolerances setups): Test scenarios only refer to their assets; the actual assets like a scenario's reference trajectories file, mapping configuration, tolerances setup, clock (in case it is an equidistant time grid scenario) and the variables used by such are each defined in separate lists. This enables to reuse assets in different scenarios, i.e., they can be shared. A typical use-case is, for example, to share a single mapping configuration among all test scenarios; to use equivalent column names to represent the same reference trajectories among different scenarios that are created using the same tooling is not unlikely. Another use-case for shared assets is to have two test scenarios referencing the very same reference trajectories file: one scenario with variable time grid and one with equidistant. Such setup provides high flexibility: if the actual sampling period of the model to test fits the presupposed one of the scenario, the precise equidistant time grid scenario can be used, otherwise one can fallback to the variable time grid scenario. In such setup, the variable time grid scenario might use a less strict tolerances setup than the equidistant one to account for interpolation imprecisions.

The drawback of separately defined assets is, that if such are shared and an update is supposed to only change a certain of the shared test scenarios, one cannot just update the existing asset but instead must create a copy that now can be respectively changed. Just updating the existing asset would change all test scenarios referencing it. Handling correct update behavior and optimize sharing is a tool support issue however, whereas providing the means to define comprehensible test scenario narratives — like which scenarios share the same tolerances setup, finding all tolerances definitions for a variable, representing test scenarios as a collection of their assets and not vice-versa — is a standardization challenge.

Every alternative design approach has its own issues. Embedding every asset as element, such that each test scenario contains its whole definition, likely implies many copies of the very same definitions — like a common mapping configuration — if there are many test scenarios. And using another dominant composition, like letting variables define tolerances and mapping configurations and compile the actual test scenarios indirectly via links from each variable to the test scenario it is part of, destroys the convenient narrative of test scenarios being the vocal point and introduces many complicated semantic restrictions to ensure consistency and completeness (e.g., because supporting test scenarios varying in tolerances and mapping configurations would require several definitions for the very same variable, each linking to its

§5: Mapping configurations of manifest

The mapping configurations of a Behavioral Model manifest are defined by its `<CsvMappings>` list; each configuration as an individual `<CsvMapping>` element. Besides the `id` used to reference a mapping configuration, configurations define in which columns the trajectories for the reference time grid and individual variables are stored (`<TimeData>` and `<ReferenceData>` elements) if a reference trajectories file is structured according to the configuration. The trajectory values for a variable can require several columns if the variable is a multi-dimensional; in that case, the individual trajectories of the element-indexes are scalarized into separate columns defined by respective `<ColumnMapping>` elements. Completeness rules ensure that every mapping configuration defines a mapping for all trajectories of all variables that are required for testing; and consistency rules ensure that there are no contradictions that might cause misinterpretations (for example due to different variables mapping to the same column).

The XSD of `<CsvMappings>` (complex type `efmiCsvMappings`), `<CsvMapping>`, `<TimeData>`, `<ReferenceData>` and `<ColumnMapping>` — defined in `BehavioralModel/efmiCsvMappings.xsd` — is:
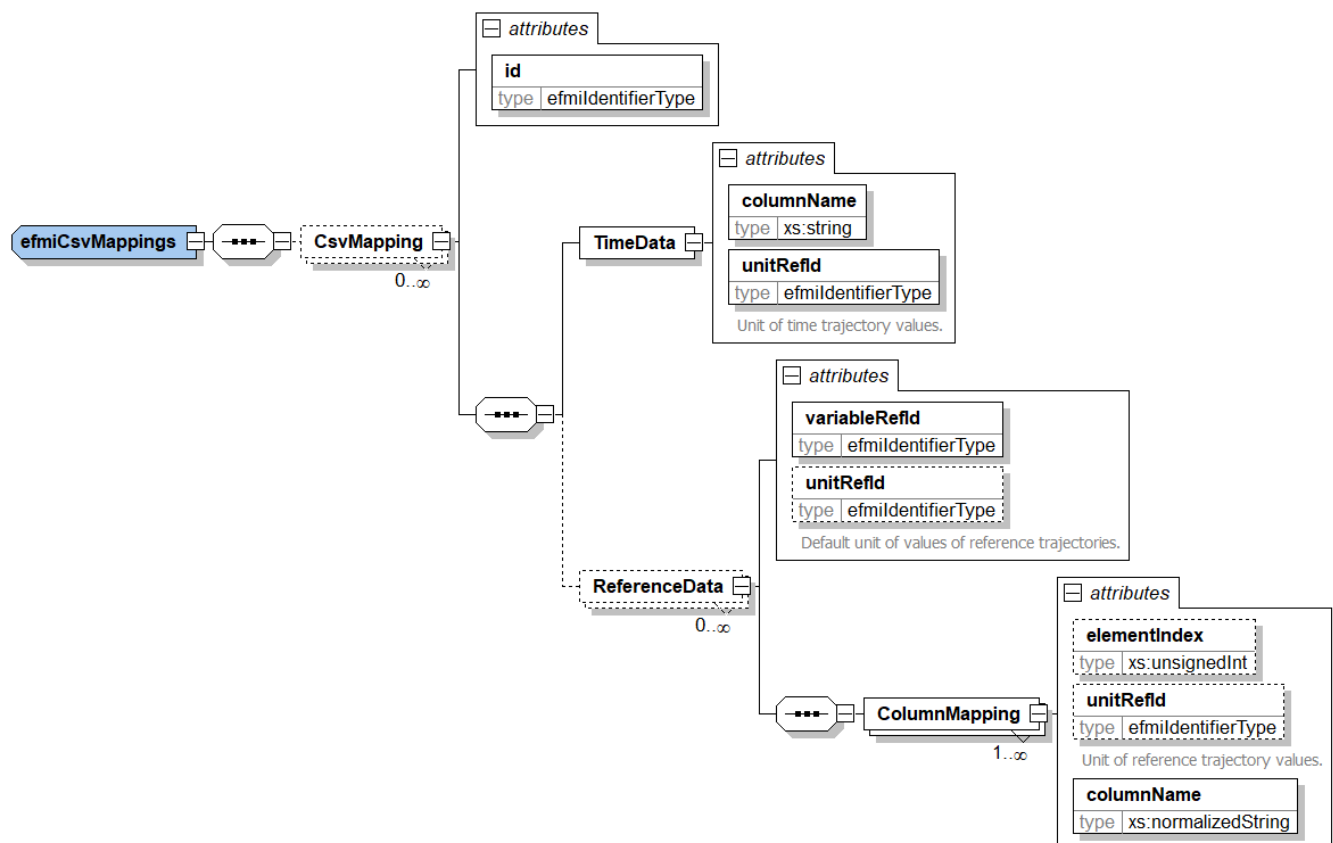
The attributes of `<TimeData>` are:

Name	Description
columnName	Name of the reference time grid column. The column names of a mapping configuration must be distinguished according to §1.1 of Section 4.2.

Name	Description
`unitRefId`	Reference to the unit of the reference time grid. All time-related units of a test scenario (like the units of actual sampling period, presupposed sampling period and reference time grid) must align according to §1.3 of Section 4.2.

The attributes and elements of `<ReferenceData>` are:

Name	Description
`variableRefId`	Reference to the variable *v* for which a mapping to the column(s) of reference trajectories files is defined. The referenced variables of a mapping configuration must be distinguished according to §1.1 of Section 4.2.
`unitRefId`	Optional reference to the default unit of all trajectories of the variable. Must not be given if the referenced variable *v* is scalar. Otherwise (i.e., *v* is multi-dimensional), the unit holds for the individual trajectories of each element-index whose unit is not refined by `<ColumnMapping>` elements. All units a test scenario compiles for the same variable and element-index (like the units defined for it in the Algorithm Code manifest, mapping configuration and tolerance definition) must align according to §1.3 of Section 4.2.
`<ColumnMapping>`	List of at least one `<ColumnMapping>` defining the column(s) of reference trajectories files onto which the referenced variable *v* is mapped. If *v* is scalar, exactly one `<ColumnMapping>` element without any `elementIndex` must be given; otherwise, all contained `<ColumnMapping>` elements must have an distinguished `elementIndex`.

The attributes of `<ColumnMapping>` are:

Name	Description
`elementIndex`	Optional index of the element for which the `<ColumnMapping>` defines a column. Must be given, if, and only if, the variable *v* the containing `<ReferenceData>` refers to is multi-dimensional. Let $(s_1, s_2, ..., s_n)$, $n \in \mathbb{N}_1$ be the sizes of the dimensions of *v*. The value *i* of `elementIndex` must be a GALEC `Integer` and be in range $1 \leq i \leq s_1 \times s_2 \times ... \times s_n$; it refers to the *i*'th element of *v* in row-major order.
`unitRefId`	Optional reference to the unit of the trajectory. A given value overrides any default unit of the containing `<ReferenceData>` element for the given element-index. All units a test scenario compiles for the same variable and element-index (like the units defined for it in the Algorithm Code manifest, mapping configuration and tolerance definition) must align according to §1.3 of Section 4.2.
`columnName`	Name of the column storing the trajectory for the (element-index of the) variable the containing `<ReferenceData>` refers to. The column names of a mapping configuration must be distinguished according to §1.1 of Section 4.2.

According to §1.1 of Section 4.2, mapping configurations must be complete, i.e., define column links for (each element-index of) all input variables according to the Algorithm Code manifest the Behavioral Model manifest refers to; and according to §1.3 of Section 4.2, must always define a unit for (the element-index of) a variable if the Algorithm Code manifest defines a unit for the respective
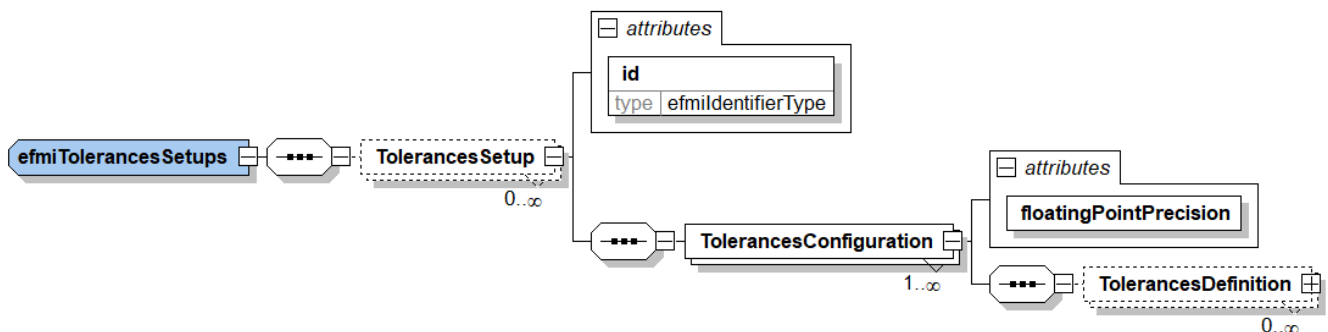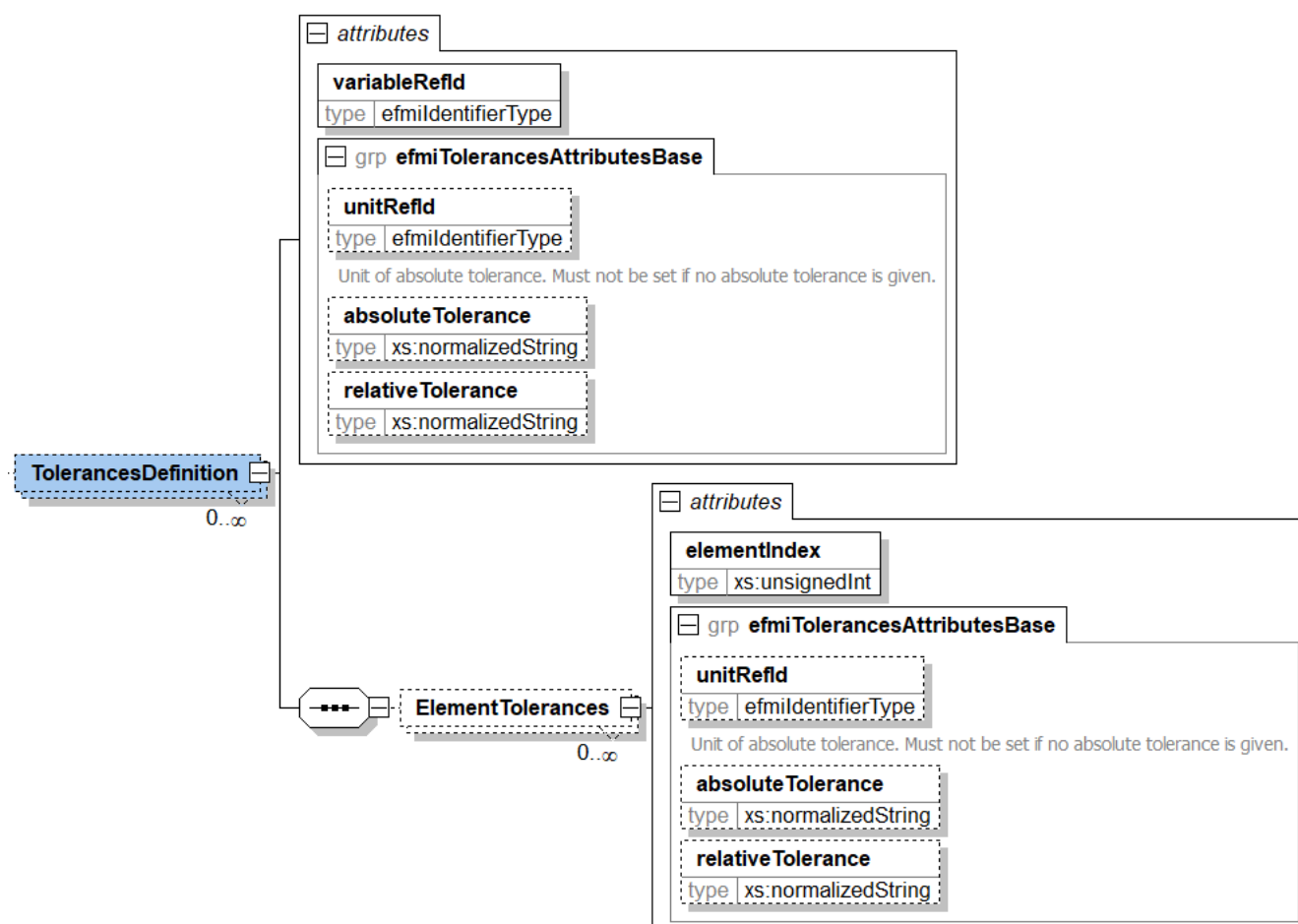
(element-index of the) variable (thus, units are only optional if, and only if, the Algorithm Code manifest defines none; but mapping configurations can introduce units if the respective (element-index of the) variable has none in the Algorithm Code manifest).

§6: Tolerances setups of manifest

The tolerances setups of a Behavioral Model manifest are defined by its `<TolerancesSetups>` list; each setup as an individual `<TolerancesSetup>` element. Besides the `id` used to reference a tolerances setup, setups define a tolerances configuration (contained `<TolerancesConfiguration>` elements) for each floating-point precision they support; and each tolerances configuration in turn can define absolute and relative tolerances in tolerance definitions (contained `<TolerancesDefinition>` elements) for the tunable parameters, dependent parameters and outputs relevant for testing. Tolerances can be defined for each individual trajectory — i.e., element-index — of a multi-dimensional variable, either via a default tolerance for all trajectories or by refined tolerances overriding the default for a certain element-index; the latter can be defined via individual `<ElementTolerances>` within a tolerances definition. Completeness rules ensure that every tolerances configuration defines tolerances for all trajectories of all variables that are of interest for testing; and consistency rules ensure that there are no contradictions that might cause misinterpretations (for example due to several, contradicting tolerances definitions for the very same variable or contradicting units between trajectory values and their absolute tolerance).

The XSD of `<TolerancesSetups>` (complex type `efmiTolerancesSetups`), `<TolerancesSetup>`, `<TolerancesConfiguration>`, `<TolerancesDefinition>` and `<ElementTolerances>` — defined in `BehavioralModel/efmiTolerances.xsd` — is:

The attributes and elements of `<TolerancesConfiguration>` are:

Name	Description
floatingPointPrecision	IEEE 754-2019 floating-point precision for which this tolerances configuration defines acceptable tolerances. One of 16-bit, 32-bit, 64-bit, 128-bit, 256-bit or any. If any, the configuration is the default configuration to be used for testing Algorithm Code containers or whenever no other specialized configuration is given for an actual test setup (i.e., when the tested Production or Binary Code container uses a floating-point precision for which the used tolerances setup misses a tolerances configuration). The floating-point precisions between the different tolerances configurations of a tolerances setup must be distinguished.
`<TolerancesDefinition>`	List of tolerances definitions to be used when comparing actual vs. reference values, each defining the error tolerances applicable for (a certain element-index of) a certain variable in test scenarios conducted with the floating-point precision of this tolerances configuration. The referenced variables of the tolerances definitions of a tolerances configuration must be distinguished.

In case a tolerances configuration does not contain any tolerances definition defining the absolute tolerance for (the element-index of) a variable suited as `variableRefId` value, the respective absolute tolerance is *0*; and likewise for relative tolerances.

> **E-3 (minimal, generic tolerances setup):** Thanks to default tolerances in case absolute and relative tolerances are not defined, and the possibility to denote that a tolerances

configuration can be used in any floating-point precision test environment (`floatingPointPrecision="any"`) in case there is no more specific configuration available, it is possible to define a minimal generic tolerances setup working for any test scenario, block-interface and test environment:

```
1 <TolerancesSetups>
2   <TolerancesSetup id="MinimalGenericSetup">
3     <TolerancesConfiguration floatingPointPrecision="any"/>
4   </TolerancesSetup>
5 </TolerancesSetups>
```

The drawback of such definition is, that, according to §3-4 of Section 4.2, an absolute *and* relative tolerance of *0* for (the element-index of) a variable *v* means, that no deviations between actual and reference values are tolerated for (the respective element-index of) *v*, or, in case *v* is a tunable parameter, recalibration is conducted for any value changes regardless how minimal. This means, that above minimal, generic tolerances setup enforces tested code to match perfectly the expected reference trajectories. Such strict conditions are unlikely to be intentional and are very hard to satisfy considering that reference trajectories are typically not generated from the GALEC code of an eFMU's Algorithm Code container, but instead from independent offline simulations of the model subject to eFMU generation. Hence, the simulation and tested eFMU code are likely using completely different numerical approaches and floating-point arithmetics are prone to minor deviations for even the simplest change in just the order of floating-point operations.

The attributes and elements of `<TolerancesDefinition>` are:

Name	Description
`variableRefId`	Reference to the variable for which this tolerances definition defines tolerances. The referenced variable must be a tunable parameter, dependent parameter or output of type `Integer` or `Real`.
`unitRefId`	Optional reference to the unit of the absolute default tolerance for the variable. Must be given, if, and only if, an absolute default tolerance is defined. All units a test scenario compiles for the same variable and element-index (like the units defined for it in the Algorithm Code manifest, mapping configuration and tolerance definition) must align according to §1.3 of Section 4.2.
`absoluteTolerance`	Optional absolute default tolerance for all trajectories of the variable. If given, the value *t* must be a GALEC `Integer` if the referenced variable *v* is of type `Integer`; otherwise it must be a GALEC `Real`; in any case, it must hold that $t \geq 0$. If *v* is multi-dimensional, the tolerance holds for the individual trajectories of each element-index whose absolute tolerance is not refined by `<ElementTolerances>` elements.
`relativeTolerance`	Optional relative default tolerance for all trajectories of the variable. If given, the value *t* must be a GALEC `Real` with $t \geq 0$. If the referenced variable *v* is multi-dimensional, *t* holds for the individual trajectories of each element-index whose relative tolerance is not refined by `<ElementTolerances>` elements.

Name	Description
`<ElementTolerances>`	List of refined error tolerances for the individual element-indexes of the variable *v* this tolerances definition refers to. Must be empty if *v* is not a multi-dimensional (i.e., is a scalar); otherwise, all contained `<ElementTolerances>` elements must have an distinguished `elementIndex`.

A tolerances definition for a scalar variable must define at least one tolerance (i.e., either `absoluteTolerance` or `relativeTolerance` or both are given).

The attributes of `<ElementTolerances>` are:

Name	Description
`elementIndex`	Index of the element for which the `<ElementTolerances>` defines error tolerances. Let $(s_1, s_2, ..., s_n)$, $n \in \mathbb{N}_1$ be the sizes of the dimensions of the variable *v* the containing tolerances definition refers to. The value *i* of `elementIndex` must be a GALEC `Integer` and be in range $1 \leq i \leq s_1 \times s_2 \times ... \times s_n$; it refers to the *i*'th element of *v* in row-major order.
`unitRefId`	Optional reference to the unit of the absolute tolerance. Must be given, if, and only if, an absolute tolerance is defined. All units a test scenario compiles for the same variable and element-index (like the units defined for it in the Algorithm Code manifest, mapping configuration and tolerance definition) must align according to §1.3 of Section 4.2.
`absoluteTolerance`	Optional absolute tolerance for the trajectory. If given, the value *t* must be a GALEC `Integer` if the referenced variable *v* of the containing tolerances definition is of type `Integer`; otherwise it must be a GALEC `Real`; in any case, it must hold that $t \geq 0$. *t* overrides any absolute default tolerance of the containing `<TolerancesDefinition>` element for the given element-index.
`relativeTolerance`	Optional relative tolerance for the trajectory. If given, the value *t* must be a GALEC `Real` with $t \geq 0$. *t* overrides any relative default tolerance of the containing `<TolerancesDefinition>` element for the given element-index.

An `<ElementTolerances>` element must define at least one tolerance (i.e., either `absoluteTolerance` or `relativeTolerance` or both are given).

4.2. Behavioral Model semantic

Assuming a valid Behavioral Model manifest according to Section 4.1 is given, this section defines the semantics for testing another Algorithm, Production or Binary Code container of the eFMU according to one of the test scenarios of the manifest. To that end, we define restrictions on the reference trajectories file of the scenario (i.e., under which conditions it defines reference trajectories suited for testing) and how to apply its reference trajectories for testing a GALEC block-interface and life-cycle (cf. Section 3.2.3) based program.

§1: Structure and content of reference trajectories files

Reference trajectories files contained in a Behavioral Model representation must satisfy RFC 4180 *"Common Format and MIME Type for Comma-Separated Values (CSV) Files"* including its errata (cf. https://www.rfc-editor.org/). Additionally to RFC 4180, the constraints defined in the following apply.

Double quoted fields are only permitted in the header line, whose fields must not contain linefeed or carriage return characters (hexadecimal 0A and 0D in ANSI X3.4-1986).

> ***L-1 (restriction to printable ANSI X3.4-1986 values for CSV fields):*** RFC 4180 only permits the subset of printable characters (hexadecimals 20 to 7E) and double quoted comma, carriage return, linefeed and quote (hexadecimals 2C, 0D, 0A and 22) of ANSI X3.4-1986 (US-ASCII) as field values. This restriction is not in conflict with names of GALEC *identifiers*, multi-dimensional queries of such and the values of `Integer` and `Real`; such are all composed of printable characters (i.e., within the restricted subset of US-ASCII) and therefore can be used as CSV field values.

A reference trajectories file must have at least two lines (which, according to §1.1 and §1.2, are a header line defining columns followed by another line with reference values for the first sampling).

§1.1: CSV header line and column linking

The first line of a reference trajectories file is a CSV header line; it defines the columns and their names. Each column name must be unique; its value is the respective CSV field's value without enclosing double quotes if any, and each escaped double quote (`2DQUOTE` in RFC 4180) replaced by a single double quote character (`"`).

The first column must be the reference time grid as defined in the Behavioral Model manifest by name via the `columnName` attribute of the `<TimeData>` element of the scenario's mapping configuration (cf. §5 of Section 4.1). Optional `Recalibrate()` and `DoStep()` columns define reference values for the error signals returned by the `Recalibrate()` and `DoStep()` block-interface methods respectively (hence, `Recalibrate()` and `DoStep()` are reserved column names with special meaning). All other column names (i.e., except the reference time grid and error signal columns) can be used by the scenario's mapping configuration — as defined in the Behavioral Model manifest — to bidirectional link columns to individual element-indexes of block variables of the Algorithm Code manifest.

Each column can be linked to at most one block variable element-index (with scalar block variables treated as 1-dimensional with size 1), and each block variable

- in case it is scalar, can be linked to at most one column

- in case it is multidimensional, each of its element-indexes can be linked to at most one column

within a single reference trajectories file. Not all columns have to be linked; but whenever the manifest links a column by name and element-index via the `columnName` and `elementIndex` attributes of a `<ColumnMapping>` element of a `<ReferenceData>` element of the scenario's mapping configuration (cf. §5 of Section 4.1), the respective column must exist in the reference

trajectories file.

The reference time grid and error signal columns must not be linked. All block inputs must, for each element-index, be linked to a column. Columns for the element-indexes of tunable parameters, dependent parameters and block outputs are optional; if any element-index of such is not linked to a column, it simply is not tested. Linking block constants or states to columns is forbidden.

> **R-2 (restriction of reference time grid as first column):** In general, mapping configurations could define any column besides the first to be the reference time grid. In practice, CSV processing — like plotting curves from a CSV file — is often much more convenient in third party tooling lacking dedicated eFMI support if the reference time grid is stored in the first column.
>
> Likewise, the whole reference time grid column could be skipped for equidistant time grid scenarios. The sampling period of the clock of an equidistant time grid scenario, together with the knowledge about how many reference points (i.e., lines) the scenario's reference trajectories file has, would be sufficient to define the presupposed time grid. Further third party processing of references trajectories files would become complicated however, since a reference time grid is typically required and now would have to be constructed on the fly. An explicit reference time grid also has the advantage that recording errors when constructing the reference trajectories can be detected by the consistency rules between reference time grid, actual time grid and presupposed time grid (cf. §2.2). If, for example, a references trajectories file stems from real-time recordings of hardware-in-the-loop (HiL), any real-time violations are captured when the actual recording times are stored as reference time grid.

§1.2: CSV non-header lines and reference points

Each line following the CSV header line defines the value of each column at a certain time point; the respective time point is the value given in the reference time grid column. The tuple *(t, v)* of the field value *v* of a column *c* for a certain time point *t* is called a reference point of *c*.

All column values must be according to the GALEC syntax for `Integer` and `Real` constants. Values of columns linked to GALEC `Integer` variables must be integers; values of columns linked to GALEC `Real` variables must be reals. Values of columns linked to GALEC `Boolean` variables have to be represented by the `Integer` values `0` for `false` and `1` for `true`. Values of error signal columns must be integers (GALEC `Integer`) representing any valid combination of the error signals exposed by their respective block-interface method according to the Algorithm Code manifest (cf. the GALEC specification for details regarding pre-defined and exposed error signals). The values of the reference time grid column must be reals (GALEC `Real`).

> **L-3 (whitespace-free trajectory values):** In general, RFC 4180 fields can contain whitespace, but above definition prohibits whitespace characters — except within header lines — since field values are GALEC `Integer` or `Real` values (i.e., without any whitespace preceding, following or inbetween).

§1.3: Unit of column values and tolerances

All values of a column, except error signal columns and the values of the CSV header line, have the same unit, defined by the scenario's mapping configuration (cf. §5 of Section 4.1); the unit of a column is

- in case of the reference time grid column, the unit defined by the `<TimeData>` element linking to the column

- otherwise, the unit defined by the `<ColumnMapping>` (preferred) or `<ReferenceData>` element linking to the column if either defines a unit

- T otherwise (no unit defined, therefore all units are valid w.r.t. any constraints)

Error signal columns have no unit.

The unit of the reference time grid column must be equivalent to the unit of the `<Clock>` variable of the Algorithm Code manifest the Behavioral Model manifest links to. For equidistant time grid scenarios, the units of their `<Clock>` and reference time grid column must be equivalent. The unit of an absolute tolerance for a linked column c must be equivalent to the unit of c. Let u_B be the unit of a linked column and u_A be the unit

- for the element-index of the variable v (in case v is multi-dimensional)

- the variable v (in case v is scalar)

in the Algorithm Code manifest the column links to; one of the following must hold:

- $u_B = u_A$ (including the special case that both are T) or

- $u_A = T$ (and u_B can be anything, i.e., introduce a unit for the trajectory)

> **L-4 (unit equivalence between Algorithm Code and Behavioral Model containers):** Since the unit of a linked column of the reference trajectories file, the block variable it is linked to, and potential tolerances must be equivalent, there is no need for unit conversions. This restriction eases implementation effort for tools, but likely will be dropped in a future version of the standard, which then supports, constraints and requires respective automatic value conversions.
>
> Units are nevertheless explicitly defined in Behavioral Model manifests to enable the detection of misalignments, when, for example, the reference points of a scenario have been recorded using different units than a Production Code to test or absolute tolerances have different units than the trajectory they are defined for. Such errors would be nasty pitfalls if there would be no explicit units in Behavioral Model manifests, for example, when regenerating GALEC and production codes with changed units without updating Behavioral Model containers accordingly.

§2: Time grids and reference values at actual sampling points

§2.1: Reference and actual time grids

Values of the reference time grid column must be strictly monotonically increasing from line to line; their ascending ordered sequence is the reference time grid S_T. It must hold that $\forall t \in S_T: t \geq 0$. We denote the i'th value of a sequence s via $value(i, s)$; the first and last value of the reference time grid are therefore $value(1, S_T)$ and $value(|S_T|, S_T)$ respectively.

Let $s_A \in \mathbb{R}$ be the actual sampling period according to the clock of the Algorithm Code manifest the Behavioral Model manifest links to; it must hold that $s_A > 0$. The time tolerance $\Delta_{t\pm}$ for time related deviations is

- in case of equidistant time grid scenarios, $\Delta_{t\pm} = 0.05 \times s_A$ (i.e., 5% of the actual sampling period s_A)

- otherwise, $\Delta_{t\pm} = 0$

The first actual sampling point $t_{start} \in \mathbb{R}$ is $t_{start} = value(1, S_T)$ (i.e., at the smallest reference time grid value). The last actual sampling point $t_{end} \in \mathbb{R}$ is $t_{end} = t_{start} + i_{end} \times s_A$ where $i_{end} \in \mathbb{Z}$ is the largest number such that $t_{end} \leq value(|S_T|, S_T) + \Delta_{t\pm}$ holds. All other actual sampling points are at time points $t \in \mathbb{R}$, $t_{start} < t < t_{end}$ for which holds: $\exists i \in \mathbb{Z}: t = t_{start} + i \times s_A$. The ascending ordered sequence of all actual sampling points is the actual time grid S_A.

> ***R-5 (difference between reference and actual time grids):*** Above definition of actual sampling points has no strict implications or restrictions on the reference time grid as defined in the reference trajectories file, which — in particular for variable time grid scenarios — is independently defined from the actual time grid. For variable time grid scenarios, the values of the reference time grid can be actual sampling points, between actual sampling points or even after the last actual sampling point (e.g., a reference time grid for a test scenario with 1s sampling period could have time points at 3.1s, 5.7s and 7.9s, yielding an actual time grid with 5 points at 3.1s, 4.1s, 5.1s, 6.1s and 7.1s). For equidistant time grid scenarios, this freedom is restricted in §2.2 to ensure that the reference time grid aligns with the actual time grid within the time tolerance $\Delta_{t\pm}$.

> ***R-6 (well-defined time span of tests):*** Above definitions ensure that there are no inept implicit assumptions about the starting point and time span of simulations used to generate reference trajectories files, while still ensuring unique interpretation of such:
>
> - Sampling starts with the first reference time grid value, whatever it is.
>
> - There is no need for extrapolation (because for variable time grid scenarios, $\Delta_{t\pm} = 0$ and therefore $t_{end} \leq value(|S_T|, S_T)$; and for equidistant time grid scenarios, the last actual sampling point is at most $\Delta_{t\pm}$ after $value(|S_T|, S_T)$, i.e., within the time tolerance up to which we want to conduct tests and for which reference values are defined according to §2.3).
>
> Thus, it is possible to do snap-shot recordings of long running simulations — where start and end time are not necessarily of any specific value — for generating reference trajectories files.

§2.2: Presupposed time grid and acceptable time deviation of equidistant time grid scenarios

Let $s_B \in \mathbb{R}$ be the presupposed sampling period according to the clock the Behavioral Model manifest defines for an equidistant time grid scenario; it must hold that $s_B > 0$. The first presupposed sampling point $t_{start} \in \mathbb{R}$ is $t_{start} = value(1, S_T)$ (i.e., at the smallest reference time grid value). The last presupposed sampling point $t_{end} \in \mathbb{R}$ is $t_{end} = t_{start} + i_{end} \times s_B$ where $i_{end} \in \mathbb{Z}$ is the largest number such that $t_{end} \leq value(|S_T|, S_T) + \Delta_{t\pm}$ holds. All other presupposed sampling points are at time points $t \in \mathbb{R}$, $t_{start} < t < t_{end}$ for which holds: $\exists i \in \mathbb{Z}: t = t_{start} + i \times s_B$. The ascending ordered sequence of all presupposed sampling points is the presupposed time grid S_B.

For equidistant time grid scenarios it must hold that

- $|S_A| = |S_B| = |S_T|$ (all time grids align on the number of samplings conducted for the scenario)
- $\forall i \in \mathbb{Z}$, $1 \leq i \leq |S_A|$, $b_{[} = value(i, S_A) - \Delta_{t\pm}$, $b_{]} = value(i, S_A) + \Delta_{t\pm}$: $b_{[} \leq value(i, S_B) \leq b_{]} \land b_{[} \leq value(i, S_T) \leq b_{]}$ (actual and presupposed time grids align for the scenario within the permitted time tolerance, i.e., for this scenario, the presupposed sampling period of the Behavioral Model manifest is close enough to the actual sampling period of the Algorithm Code manifest it links to; likewise, each reference time point of the reference time grid aligns with an unique actual sampling point within the permitted time tolerance; thanks to the previous constraints, and $\Delta_{t\pm}$ being significantly smaller than $0.5 \times s_A$ by definition, the permitted time tolerance around actual sampling points do never overlap for which reason the alignments are bidirectional)

> **R-7 (difference between actual and presupposed time grids):** The actual time grid defines the sampling assumed by the tested code (Algorithm, Production or Binary Code container); this is the sampling required by that very code to work properly (for example, the code might use this sampling period internally to compute derivatives). The actual time grid defines the sampling to use for testing the behavior of that very code, and therefore used to execute any test scenario. The presupposed time grid on the other hand defines the sampling under which the reference values of the reference trajectories file of an equidistant time grid scenario have been recorded.
>
> Both must be close enough to each other (i.e., align) for the equidistant time grid scenario to be applicable for testing the given code. Otherwise, the tested code might use a wrong sampling period internally. Note, that alignment is not defined by some tolerance for the sampling period as such, for which we could always find a scenario where it is to permissive — given the scenario just runs long enough — due to sampling differences adding up over time. Instead alignment is defined for the *actual* scenario by applying the acceptable time tolerance $\Delta_{t\pm}$ on each individual given actual sampling point.

§2.3: Reference values at actual sampling points

The reference value v of a column c at an actual sampling point $t_s = value(i, S_A)$, $i \in \mathbb{Z}$, $1 \leq i \leq |S_A|$ (i.e., the i'th sampling) is

- in case of equidistant time grid scenarios, the value of the reference point (t, v) of c for which $t_s - \Delta_{t\pm} \leq t \leq t_s + \Delta_{t\pm}$ (i.e., for which $t = value(i, S_T)$, and therefore v is the field value of c given on the i'th line of the reference trajectories file, excluding the CSV header line from counting)

- otherwise (i.e., in case of variable time grid scenarios)
  - in case $\exists t \in S_T: t = t_s$, the value of the reference point of c for t
  - otherwise, the interpolation result between the two reference points of c closest to t_s; the two closest reference points $r_0 = (t_0, v_0)$ and $r_1 = (t_1, v_1)$ are the reference points of c for which holds: $t_0 < t_s < t_1 \wedge \nexists t^0, t^1 \in S_T: t_0 < t^0 < t_s \vee t_s < t^1 < t_1$ (i.e., the reference points of c uniquely defined by the smallest index i for which $t_0 = value(i, S_T)$, $t_1 = value(i + 1, S_T)$, $t_0 < t_s < t_1$)

Two interpolation methods are supported for the last case, constant-segment and linear interpolation. In case of constant-segment interpolation the reference value v is v_0. In case of linear interpolation, let $v_{linear} = v_0 + ((t_s - t_0) / (t_1 - t_0)) \times (v_1 - v_0)$; then v is

- in case c is linked to a GALEC `Boolean` or `Integer` variable, the round half to even of v_{linear} (cf. GALEC builtin function `roundHalfToEven`)

- otherwise, v_{linear}

Which interpolation method is used, depends on c

- in case c is an error signal column or linked to a tunable or dependent parameter, constant-segment interpolation

- otherwise, linear interpolation

If no error signal column for a block-interface method is defined, the reference value for the respective method's error signal is `0` (i.e., a reference value for error signals of block-interface methods is always defined and tested even if no column for such exists).

> **R-8 (characteristics of time grids and reference trajectories):** All time grids are discrete-time; their plots are points of unique value and not a curve with a continuous time axis. The other trajectories (linked block variables and error signals) are continuous-time in case of variable time grid scenarios and discrete-time in case of equidistant time grid scenarios.
>
> These characteristics are not only of interest for correct plotting (continuous curves vs. points), but for any further processing by third party tooling. For example, interpolating the values of reference trajectories files of equidistant time grid scenarios as if they are continuous — i.e., using the CSV data like continuous curves — is highly questionable.

> **R-9 (reference time grids for variable time grid scenarios should follow significant trajectory changes):** Variable time grid scenarios use linear interpolation; although well-defined compared to the plethora of spline interpolation schemes, linear interpolation is a rather coarse grained interpolation scheme. Whenever trajectories have significant derivative changes or discontinuities, it is highly recommended that the reference time grid of reference trajectories files contains entries close to these points of interest. Otherwise, test results might not be meaningful; they might miss to test points of interest (undersample) and/or fail because linear interpolation does not find good reference values for actual sampling points that are far off from reference time grid points. Oversampling — i.e., to have many more time points — is not problematic for variable time grid scenarios.

> **R-10 (life-time of reference values w.r.t. the GALEC block life-cycle):** The reference values for tunable parameters are right before calling `Recalibrate()` and are valid at least until `DoStep()` finished. The reference values for dependent parameters are right after calling `Recalibrate()` and are valid at least until the next `Recalibrate()` call. The reference values for block inputs are right before calling `DoStep()` and are valid at least until `DoStep()` finished. The reference values for block outputs are right after calling `DoStep()` and are valid at least until the next `DoStep()` call. The reference values for error signals of block-interface methods are right after calling their respective method and are valid at least until the next call of their respective method.

§3: Tolerances and acceptable deviation at actual sampling points

Let c be a linked column for (some element-index of) a tunable parameter, dependent parameter or output variable V. According to §6 of Section 4.1, an absolute and relative tolerance $\Delta_{absolute}$ and $\Delta_{relative}$ will be defined for c if V is of type `Integer` or `Real`, either explicitly in the Behavioral Model manifest by some tolerances definition or implicitly if such is missing. Let v_r be the reference value of c at some actual sampling point. A value v is within tolerance of v_r, if, and only if,

- in case V is of type `Integer` or `Real`, $|v_r - v| \leq \Delta_\pm$ where $\Delta_\pm$ is the maximum of $\Delta_{absolute}$ and $\Delta_{relative} \times |v_r|$

- otherwise (i.e., V is of type `Boolean`), $v = v_r$

In case V is of type `Integer`, above equation is evaluated in real arithmetics by first promoting v_r and $\Delta_{absolute}$ to `Real` according to the GALEC builtin function `real()`; this implies in particular that $\Delta_\pm$ can be a proper real value in case the fractional part of $\Delta_{relative}$ is not 0, and $\leq$ is a real comparison.

§4: Interpretation of test scenarios

The execution of a test scenario within a certain test environment (i.e., the interpretation of a scenario for testing) follows the GALEC block-interface and life-cycle (cf. Section 3.2.3), with the sampling — and therefore interactions with block-interface variables and methods — following the actual time grid S_A. Let B be the tested block:

- The `Startup()` method of B is executed once, before any other block-interface interaction and never executed thereafter. If the error signal returned by `Startup()` is not 0, the test fails.

- At each actual sampling point, the linked tunable parameters and block inputs of B are set to their reference values before calling any block-interface method.

- The `Recalibrate()` method of B is executed once at the very first actual sampling point if the default start value, according to the Algorithm Code manifest, of a linked tunable parameter is either, undefined, or not within tolerance of its reference value. The `Recalibrate()` method is also executed at any further actual sampling point $t_s \in S_A$ where the reference value of the previous actual sampling point of a linked tunable parameter is not within tolerance of the reference value at t_s.

- The `DoStep()` method of B is executed once at each actual sampling point, after any `Recalibrate()` execution. Afterwards, the values of linked dependent parameters and linked

outputs of *B* (the actual values) are checked if they are within tolerance of their respective reference values; if any is not, the test fails.

- Whenever a block-interface method of *B* is executed, its returned error signal is compared to its respective reference value (according to §2.3, such is always defined); if the returned error signal and reference value are not equivalent, the test fails.

- There are no further interactions with *B* than the ones defined above.

A test passes, if, and only if, it doesn't fail.

R-11 (invalid Behavioral Model containers vs. nonapplicable scenarios vs. failed tests): In case of any violations of the manifest schema as defined in Section 4.1, the Behavioral Model container, and all its scenarios, are invalid. Invalid containers and scenarios cannot be used for testing according to §4.

Otherwise (i.e., if the manifest satisfies all rules of Section 4.1), but in case of any violations of rules §1-2.1 (i.e., w.r.t. a certain scenario and its reference trajectories file but excluding alignment rules related to reference time grids of equidistant time grid scenarios), the respective scenario is invalid. Invalid scenarios cannot be used for testing but other scenarios of the container might still be valid and usable for testing.

Scenarios violating only rules of §2.2 are nonapplicable (due to a too large reference time grid deviation compared to the actual sampling period); they too cannot be used for testing, but are easy to fix by regenerating the reference points of the equidistant time grid scenario with appropriate resolution (by using a sampling period aligned with the sampling period of the Algorithm Code container).

Scenarios missing a tolerances configuration for a certain floating-point precision and without a tolerances configuration with `floatingPointPrecision="any"` are nonapplicable for testing other eFMU containers requiring that very precision.

If a scenario is neither invalid nor nonapplicable, it is valid. Only valid scenarios can be used for testing as described in §4.2, in which case they either pass or fail.

L-1 (optional, production and binary code dependent support for recalibration): A test tool must reject — i.e., consider to be nonapplicable — scenarios requiring recalibration for testing Production or Binary Code containers which dropped recalibration support (i.e., assume all tunable parameters are constants fixed to their default initialization value and therefor do not provide an implementation of the block's `Recalibrate()` method). A scenario requires recalibration, if a column of a used reference trajectories file is linked to a tunable or dependent parameter and contains any value different to the parameter's default initialization value (considering tolerances).

L-2 (undecidable applicability of variable time grid scenarios): For equidistant time grid scenarios applicability for a given Algorithm, Production or Binary Code container is

---

decidable. The alignment rules of §2.2 between time grids ensure that the sampled system can be tested with the given reference trajectories without the risk of undersampling. In fact, for each sampling exact reference points are defined. This is not the case for variable time grid scenarios; that is why rationale **R-9**, that the reference time grids of variable time grid scenarios should follow significant trajectory changes, is important. A user of a variable time grid scenario cannot decide if its reference time grid is suited for the actual sampling period; he must trust that the producer recorded good enough reference trajectories.

Chapter 5. Production Code Model Representation

A Production Code Model Representation of an eFMU container contains the actual sources that implement the algorithm expressed in Algorithm Code Model Representation of the same eFMU container.

As mentioned before an eFMU container can contain any number of Production Code Model Representations.
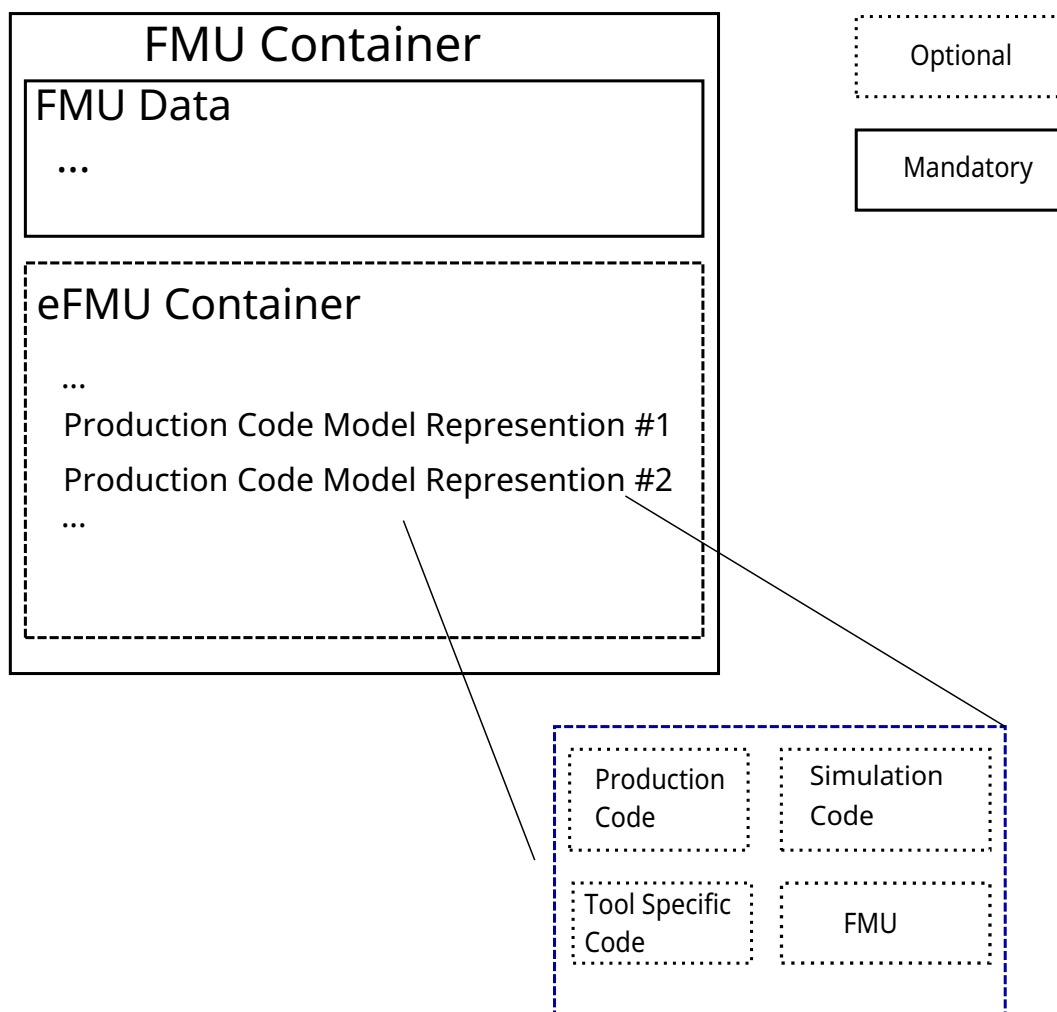
The following code parts may be present inside each Production Code Model Representation:

- Production Code: This section contains the actual Production Code running on the embedded device. In later development steps it shall be compiled and linked to be integrated on the target embedded device.

- [optional] Simulation Code: This code is used to simulate model target environment of the Production Code. It may provide stub functions for communication with other software functions.

- [optional] Tool Specific Code: Tool Specific Code may help tools to integrate the Production Code

in their (execution) environment.

- [optional] FMU container: This FMU container may be extracted and copied to the surrounding FMU Data to be consumed by FMI compatible tools directly.

The structure of the Model Representation is organized in a folder structure, but not standardized. Instead, the actual structure of the Model Representations's content, e.g. code at least as far as interfaces and externally accessible parts are concerned, is formally described in the manifest file of the Model Representation. The Model Representation is "registered" in the "__content.xml" registry of the eFMU container.

```
1                      +--------------------+
2                      |    __content.xml   |
3                      +--------------------+
4                 ^                       ^
5                 | registers            |
6                 |                      |
7    +---------------+-+        +-+------------------------------------+
8    | Algorithm       |        | Production Code Model Representation |
9    | Code            | references |                                 |
10   | Container       |        +-----| manifest.xml                  |
11   |                 |        :     |   folder                      |
12   |   manifest.xml  |<------+      |      model.h                  |
13   +-----------------+              |      model.c                  |
14                                    +------------------------------------+
```
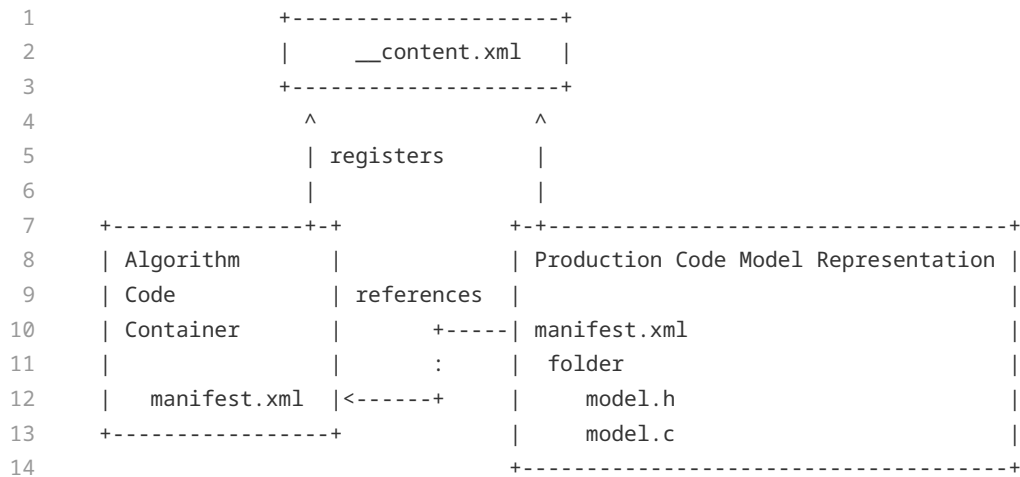
The manifest itself references to a manifest of a Algorithm Code Model Representation for more detailed information.

For each different target - the combination of compiler and processor - there exist a dedicated Production Code section inside an eFMU container. A special target is the generic one, where the included C code doesn't contain target specific parts, e.g. assember code sections or code assuming a certain hardware platform. Such a generic C code is therefore portable, i.e. compilable on an ARM architecture as well as on a i86 architecture. This flexibility allows for including an FMU into the Production Code Model Representation, that uses the generated Production Code and a FMI compatible interface.

| NOTE | An example use case for the FMU container is an early back-to-back test while already using the target datatypes: After modelling an controller, developers can easily check the resulting Production Code using FMI compatible tools. |

A generic target allows for testing and simulating the Production Code in an environment other than the target embedded device, which may require additional software parts to interface with the environment. These software parts can simulate parts of an operating system of the microcontroller, create stubs to represent other software functions that interact with the software-under-test or handle inputs, outputs and the execution.

| NOTE | Testing a Production Code Model Representation in a Processor-in-the-Loop scenario, tools using their own execution frame on the targed board. To support these use-cases this kind of code can be stored as Tool Specific Code inside the Production |

Code Model Representation. The name of the tool and its version have to be specified in the manifest file referencing the code.

5.1. Production Code Manifest

The Production Code manifest follows the general guidelines as pertaining to all manifests, including the listing of relevant manifests and files. In addition it describes the content of the "Code Files":

On the top level, the schema consists of the following elements:

Name	Description
attributes	The attributes of the top-level element are the same for all manifest kinds and are defined in section Section 2.3.1. Current kind-specific values: `kind = "ProductionCode"`, `xsdVersion` (value is the current xsd version of the schema for the Algorithm Code model manifest).
ManifestReferences	Reference to the manifest of the Algorithm Code on which this Production Code manifest is based on. This element is the same for all manifest kinds and is defined in section Section 2.3.4.3.
Files	List of files referenced in this model representation. This element is the same for all manifest kinds and is defined in section Section 2.3.3.
CodeContainer	Defines the details of the production code. For details see Section 5.1.2.
Annotations	Additional data that a vendor might want to store and that other vendors might ignore. For details see Section 2.3.4.5.

The Production Code manifest describes the structure of the contained "Production Code". Languages for the producion code include the "C" language and the "C++" language. The manifest will give more detailed information on the exact requirements on the Production Code language to integrate the code into an actual ECU software content.

NOTE   The Production Code manifest focusses on aspect directly tied to the Production Code itself in particular the technical aspects. Relevant aspect relating to the

algorithm or the "logical" concepts are referred to from the Algorithm Code manifest (e.g. whether an object is a state or calibration parameter, input or output etc.).

The Production Code manifest is an xml file with structured information about the Production Code. It contains two sections:

- Production code description section: This section contains all information directly pertaining to the code itself, i.e. the "technical realisation".

- Mapping section: this section contains all information relating to mapping the elements of the technical realistion (aka. the C-code) to the logical elements of the Algorithm Code.

This distinction into logical (as e.g. described in the Algorithm Code) and technical parts is crucial and is shown in one example here.

Example: Suppose a (logical) function *f* that computes outputs *y1* and *y2* from inputs *x1* and *x2* and a state *s1* using parameters *p1* and *p2*. This logical function could be implemented in several ways, e.g.:

- **f1** working on global variables only. In this case the (technical) function signature is that of a void void function and the expressions directly access the elements.

```
void f1() {
    ...
    s1 = ...  /* update of state s1 */
    y1 = ...  /* y1 expression */
    y2 = ...  /* y2 expression */
}
```

- **f2** that takes the inputs as arguments and returns output y1 as return value and y2 via a pointer. Access to state and parameters is through global variables

```
float f2(float x1, float x2, float *y2) {
    ...
    s1 = ...  /* update of state s1 */
    *y2 = ...  /* y2 expression */
    return ...;  /* y1 expression */
}
```

- **f3** that that works like **f2** but takes the states as a struct with two elements

... typedef struct { float s; float t; } states;

...

float f3(float x1, float x2, states myStates, float **y2) { ... **myStates.s = ... /** update of state s1 **/ *y2** = ... **/** y2 expression **/ return ...; /** y1 expression */ }

- **f4**: In this example the parameter and the state are coupled in a data structure (e.g. a spring with

parameter being the rigidity of the spring and the state being the deflection). As both are not in the same memory (one is in ROM the other in RAM), the one value is referenced per pointer. The C function itselfs takes as input an array with the two pairs.

```
1  ...
2  typedef struct {
3     float *deflection;
4     float rigidity;
5  } spring;
6
7  ....
8
9  float f4(float x1, float x2, spring[] springs, float *y2) {
10     ...
11     springs[0].deflection = ....  /* update of state s1 */
12     *y2 = ...  /* y2 expression */
13     return ...;  /* y1 expression */
14 }
```

As can be easily seen by these example, there is a big difference between the logical variables on which a function operates, and the representation of these in code. As the last two examples show, this can even go so far that the code structure contains elements that do not directly appear in the Algorithm Code.

Wheras the technical description part of the manifest relates solely to the technical (realisation) aspects of the C Code, the mapping section is dedicated to bridge the gap between the two levels of abstraction: the Algorithm Code and the Production Code.

5.1.1. Technical description of Production Code

The technical description part of the Production Code manifest specifies the following aspects of the code:

- the underlying language including detailed information on the version of the language
  - any restrictions / specification on the target (e.g. HW) for which the code is intended for
  - any restrictions / specification on the compilers to be used included specifics on compiler versions and configuration
- Definition of the type (numeric) type system on the target. This section maps the standardized (eFMI-) types onto the target types available on that specific target. These may depend on the compiler (e.g. some compilers use "int" for 32 bit and "long" for 64 bit, others use "long" for 32 bit and "long long" for 64 bit).
- Definition of the code itself. The code is thereby grouped in "Modules" which contain source files (for the language "C" normally a module contains a ".c" and a ".h" file).

For each file the content (as far as relevant and accessible) is described. This includes:

- references ("includes") to other files (defined in the Production Code manifest).
- defined types in that file (refering to the defined and standardized target types). Usually these are

specifically defined names for the type like e.g. "uint8" that are used in the actual Production Code. These defined types also contain definitions for structured types

- defined macros (if any)

- defined variables in the file

- defined functions in the file.

For Production Code Model Representations that contain e.g. AUTOSAR Classic or Adaptive code, there exist additional so-called description files, describing the technical aspects of the code. Those description files must be listed in the Code Container and are the alternative to the above mentioned details in the manifest and must be use instead.

5.1.2. Code Container

The code container groups the actual Production Code Model Representaion content, and gives specification for the following details:
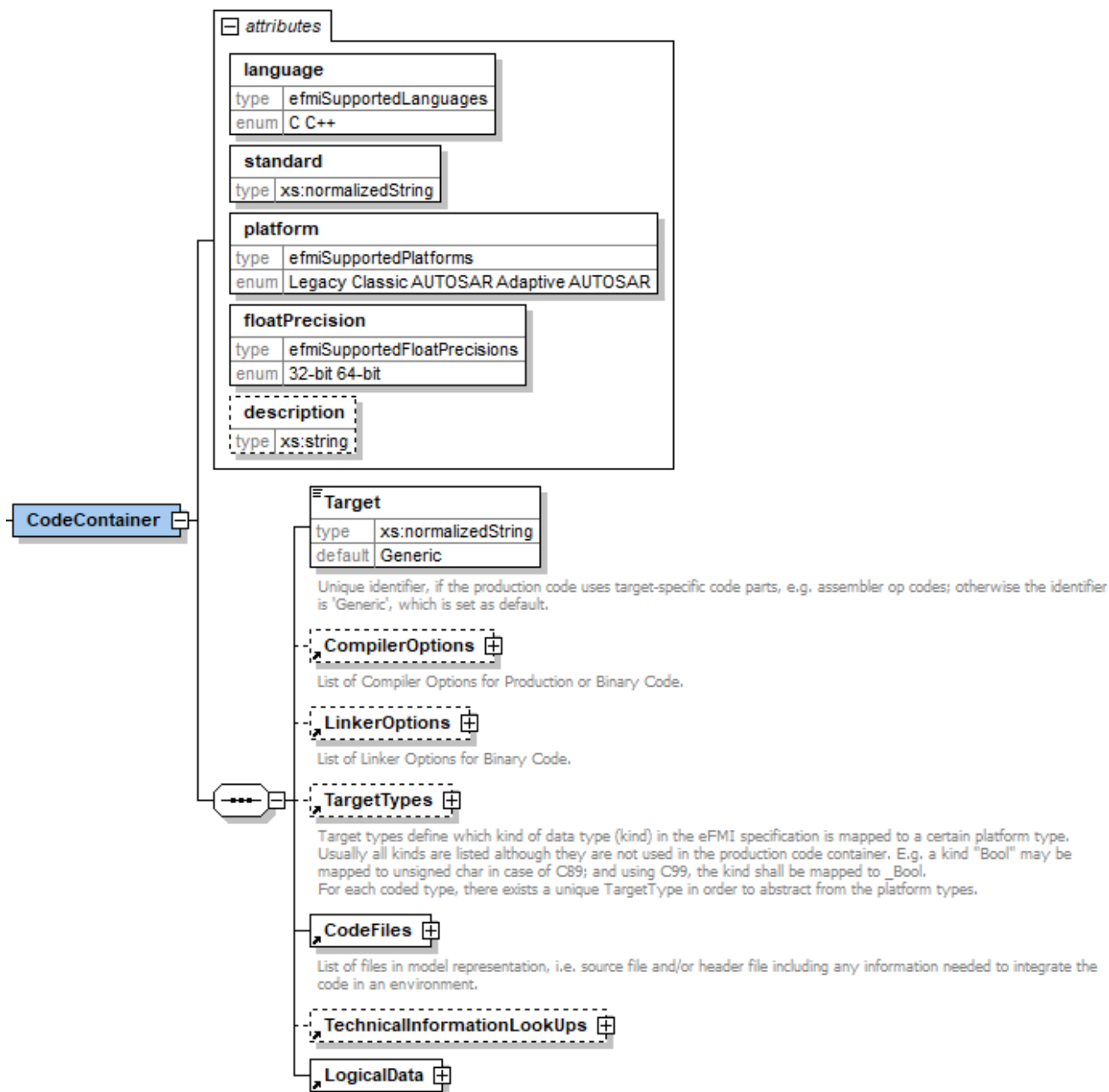
Name	Description
language	Language to be used. Currently, the following values are possible: "C" or "C++".
standard	Relevant language standard to be used.
platform	The target platform. Currently, the following values are possible: "Legacy" (= xxx) "Classic" (= xxx) "AUTOSAR" (= xxx) "Adaptive AUTOSAR" (= xxx)
floatPrecision	Floating point precision of the target platform. Currently, the following values are possible: "32-bit" or "64-bit".
description	Optional description
Target	Unique identifier, if the production code uses target-speciic code parts, for example assembler op codes; otherwise the identifier is the default Generic.

Name	Description
CompilerOptions	List of Compiler Options for Production or Binary Code. For more details, see section Section 5.1.2.1.
LinkerOptions	List of Linker Options for Production or Binary Code. For more details, see section Section 5.1.2.4.
TargetTypes	Defines which kind of data type (kind) in the eFMI specification is mapped to a certain platform type. Usually all kinds are listed although they are not used in the production code container. E.g. a kind "Bool" may be mapped to unsigned char in case of C89; and using C99, the kind shall be mapped to _Bool. For each coded type, there exists a unique TargetType in order to abstract from the platform types. For more details, see section Section 5.1.2.7.
CodeFiles	List of files in model representation, i.e. source file and/or header file including any information needed to integrate the code in an environment. For more details, see section Section 5.1.3.
TechnicalInformationLookUps	Facilitates a quick access to information in the manifest and the associated C files. For more details, see section Section 5.1.4.
LogicalData	Defines how the logical elements (variables, functions etc.) are mapped to the actual data structures and elements of functions and defined variables. For more details, see section Section 5.1.5.

Compiler Options
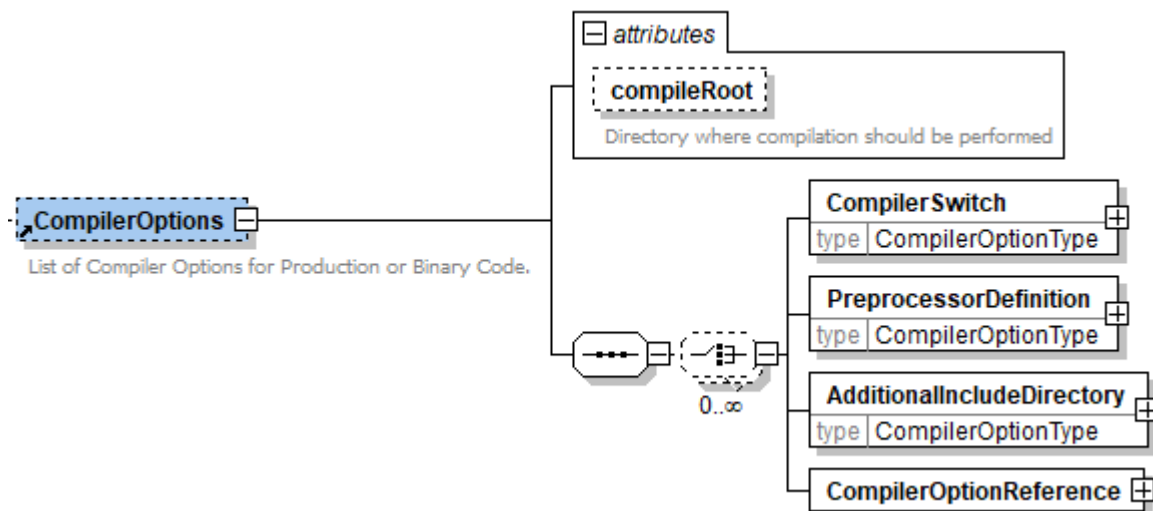
Name	Description
compileRoot	Directory where compilation should be performed.
CompilerSwitch	Compiler switch, see Section 5.1.2.2.
PreprocessorDefinition	Preprocessor definition, see Section 5.1.2.2.

Name	Description
AdditionalIncludeDirectory	Additional include directory, see Section 5.1.2.2.
CompilerOptionReference	Reference to option in another manifest file, see Section 5.1.2.3.

Compiler Option Type

Name	Description
id	Id of option.
name	Name of option.
value	Value of option.
description	Optional description of option.
optional	Definition of option is optional. Possible values: `"false"` (default) or `"true"`.

Compiler Option Reference

Name	Description
index	Index of the option reference in the list of option references.
id	Id of option reference.
manifestReferenceRefId	If of foreign manifest file.
foreignRefId	Id of option in foreign manifest file.

Linker Options

Name	Description
LinkerSwitch	The linker switches of type [LinkerOptionType].
Library	Library of type [LinkerOptionType].
AdditionalLibraryDirectory	Additional library directory of type [LinkerOptionType].

Name	Description
LinkerOption Reference	A list of option references, see Section 5.1.2.6.

Linker Option Type

Name	Description
id	Id of option.
name	Name of option.
value	Value of option.
description	Optional description of option.
optional	Definition of option is optional. Possible values: `"false"` (default) or `"true"`.

Linker Option Reference
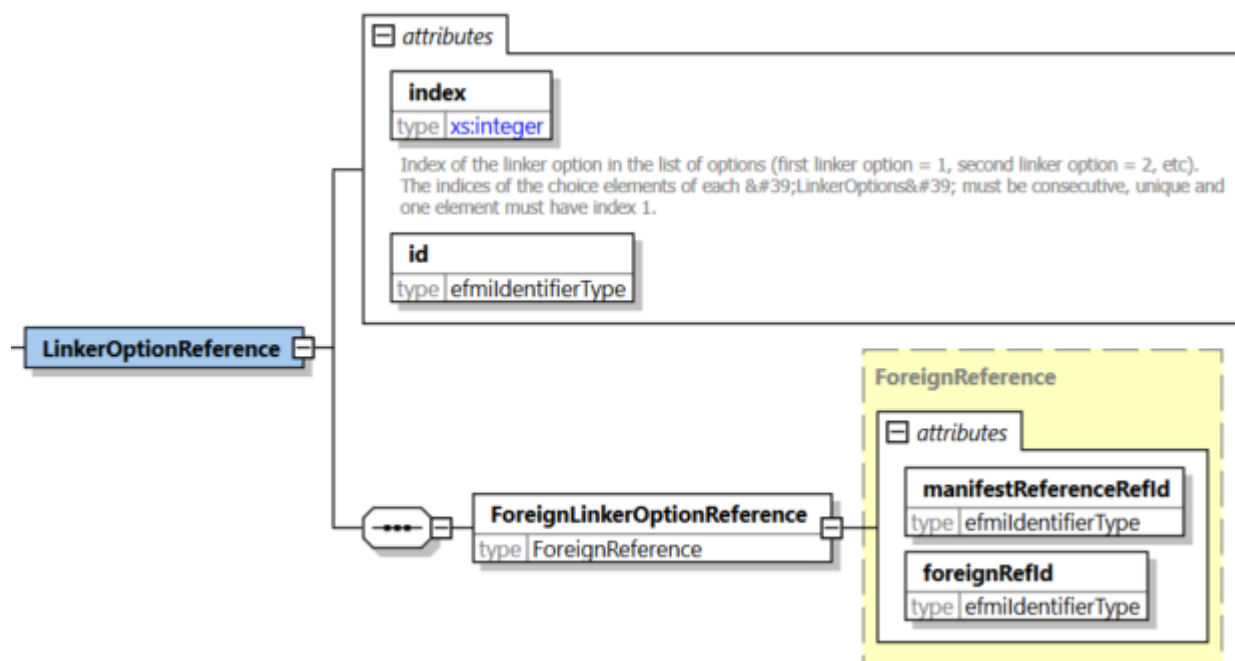
Name	Description
index	Index of the option reference in the list of option references.
id	Id of option reference.
manifestReferenceRefId	Id of foreign manifest file.
foreignRefId	Id of option in foreign manifest file.

Target Types

Target types define which kind of data type (kind) in the eFMI specification is mapped to a certain platform type. Usually all kinds are listed although they are not used in the production code container. E.g. a kind "Bool" may be mapped to unsigned char in case of C89; and using C99, the kind shall be mapped to _Bool. For each coded type, there exists a unique TargetType in order to abstract from the platform types.
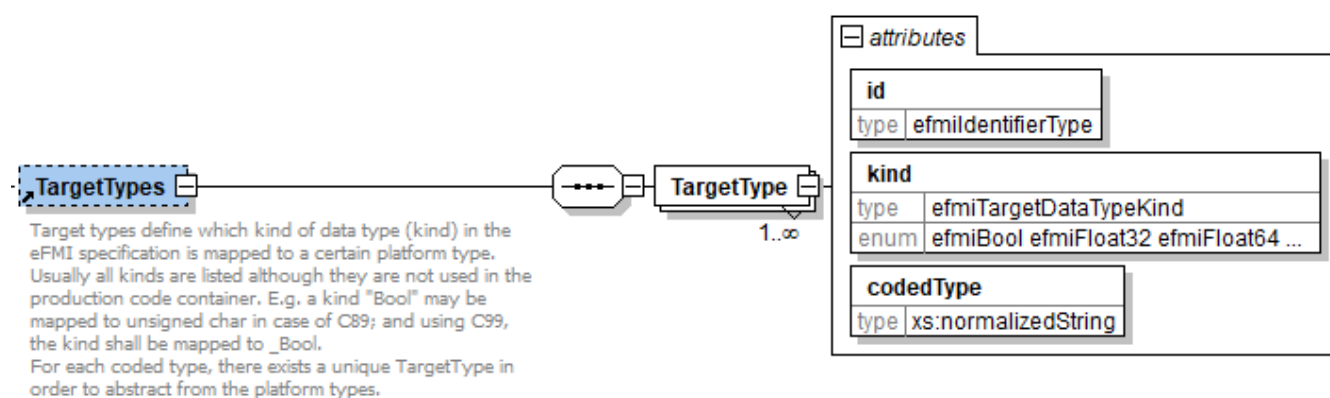
Name	Description
id	The unique id of the target type.

Name	Description
kind	The kind of the target type. The value must be one of the predefined kinds from the following list: `"efmiInteger8"`, `"efmiUnsignedInteger8"`, … , `"efmiUnsignedInteger64"`, `"efmiFloat32"`, `"efmiFloat64"`, `"efmiFloat128`, `"efmiBoolean"`, `"efmiVoid"`.
codedType	The actual Production Code type to be used, e.g. `"unsigned char"`.

Example:

```
1 <TargetType id="TT_float64" kind="efmiFloat64" codedType="double"/>
```

5.1.3. Code Files

The code file section describes the actual content of a (production) code file. It refers to one of the files listed in the "Files" section, so it is clear which file's content it actually species
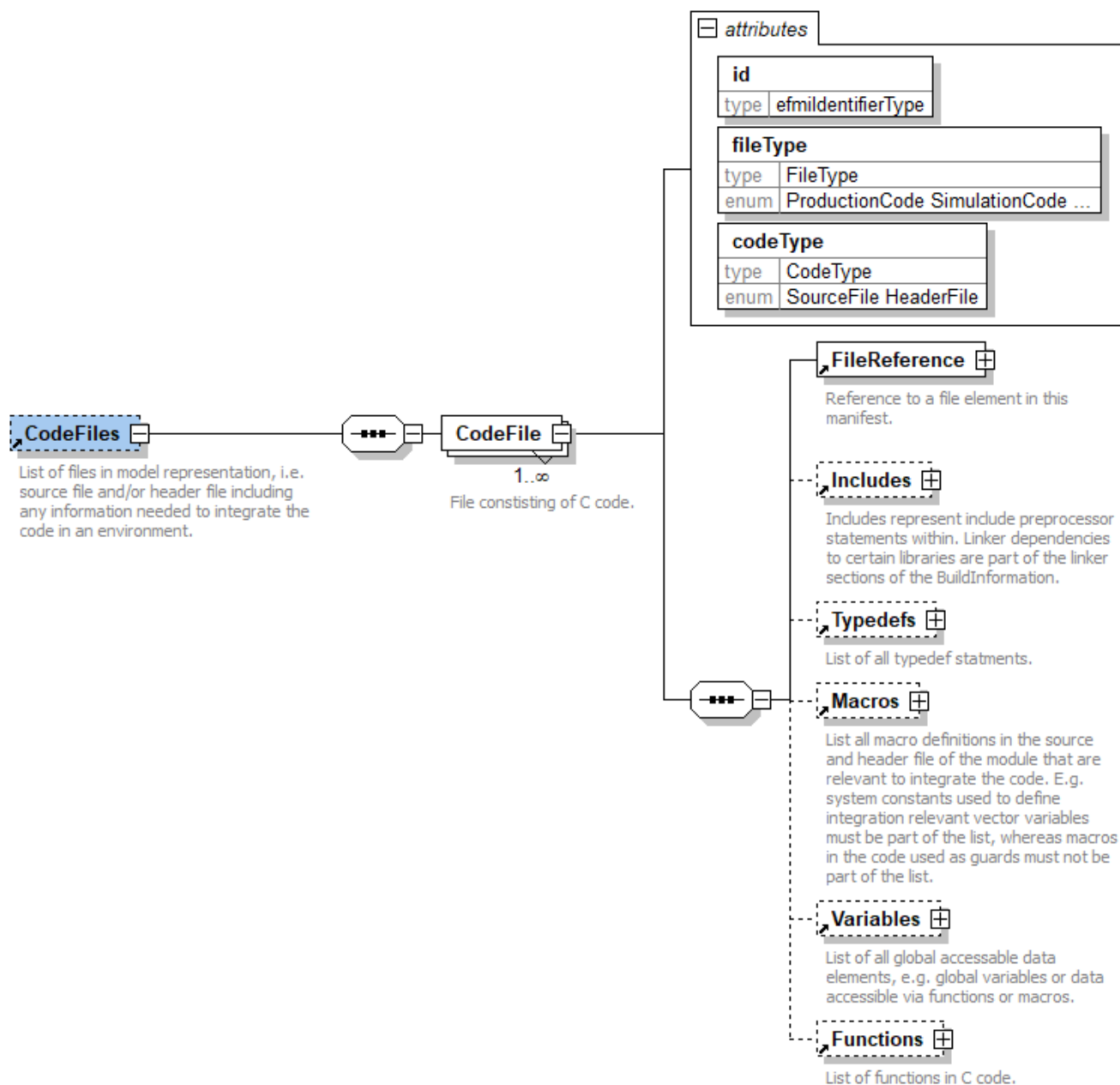
Name	Description
id	Unique id.
fileType	Type of the file. Allowed values: "ProductionCode", "SimulationCode", "ToolSpecificCode".
codeType	Type of the code. Allowed values: "SourceFile", "HeaderFile".
FileReference	Reference to a file element in this manifest file, see Section 2.3.4.2.
Includes	Definition of include files, see Section 5.1.3.1.
Typedefs	Definition of typedefs, see Section 5.1.3.2.
Macros	Definition of macros, see Section 5.1.3.2.3.
Variables	Definition of variables, see Section 5.1.3.2.4.
Functions	Definition of functions, see Section 5.1.3.2.5.

Example:

```
1 <CodeFile id="C_1" fileType="ProductionCode" codeType="SourceFile">
2    <FileReference fileRefId="F_22" kind="code"/>
3    .....
4 </CodeFile>
```

Includes

Includes represent include preprocessor statements. Linker dependencies to certain libraries are part of the linker sections of the BuildInformation.
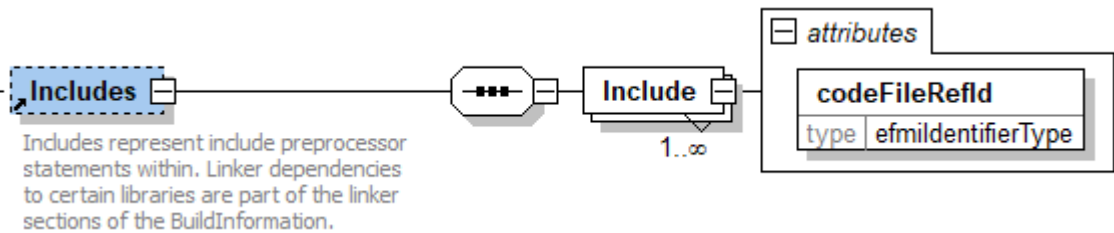
Name	Description
codeFileRefId	id of the included file. This attribute might be empty if the include is of a library.

Example:

```
1  <Include codeFileRefId="F_1"/>
```

Typedefs

Typdefs are used to either define structured types, array types or alias types (of predefined types).

Name	Description
id	Unique id of typedef.
name	name of the type
Alias	Alias means renaming of types, e.g. `"typedef unsigned char MyUint8_t;"`. Therefore the `targetTypeRefId` is always set and references the certain TargetType in the target type list and in cases of cascaded Typedefs, also the `typeDefRefId` is set. Usually, a TargetType is referenced by a most one Typedef statement. If a basetype is renamed (e.g. Int16 → MyInt16) or a user type based on an existing type is defined, two or more Typedef statements may point to a single TargetType.
Pointer	Declares a type that is a pointer to another type. This type can be any other defined type.
Components	Definition of a struct. Structs in structs are allowed but Dimensions have to be specified at variable definitions only. For details see Section 5.1.3.2.1.
EnumerationItems	Definition of an enum. For details see Section 5.1.3.2.2.

The following is an example of a simple alias declaration

Example:

```
1   <Typedef name="Float32" id="TD_F32">
2       <Alias targetTypeRefId="TT_float32" />
```

```
3    </Typedef>
```

The more complex data structure of function **spring** of the fourth example would be described by the following snipppet:

```
1    <Typedef name="spring" id="TD_spring">
2        <Components>
3          <Component id="C_1" name="deflection" typeRefId="TD_F32" pointer="true">
4          <Component id="C_2" name="rigidity" typeRefId="TD_F32">
5        <Components>
6        <Alias targetTypeRefId="TT_float64" />
7    </Typedef>
```

Components (struct)

Components declare a structure and are a list of Component:
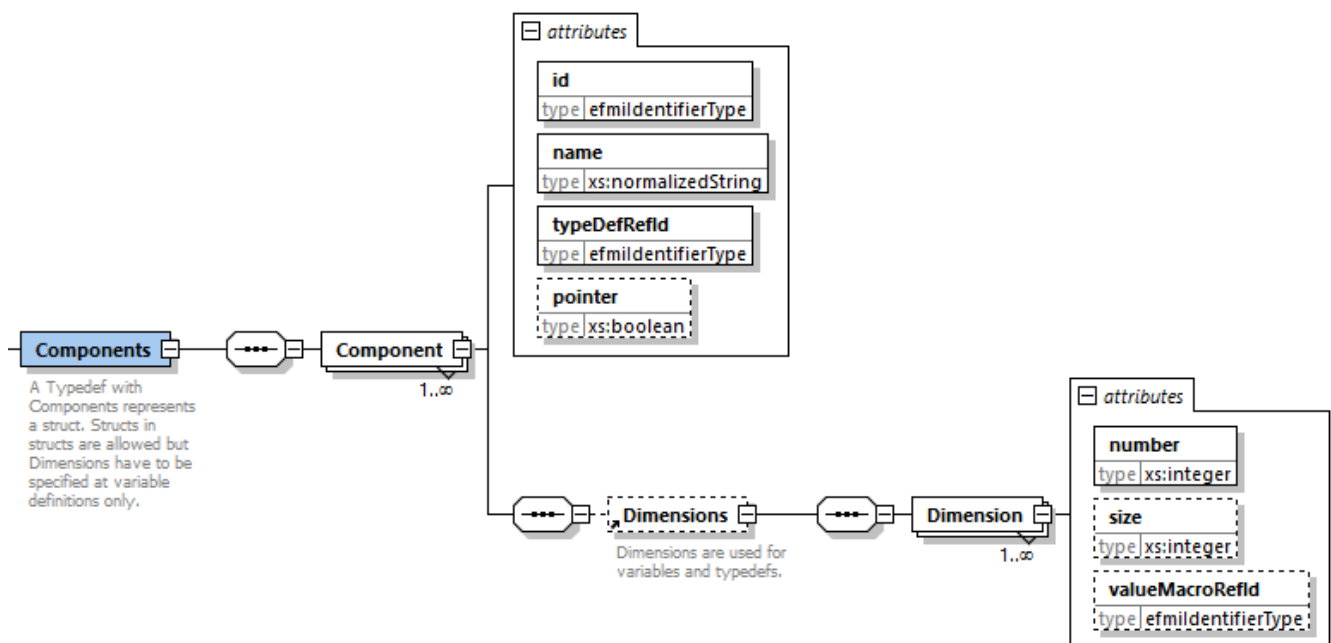
Name	Description
id	Unique id.
Name	Name of the field. Must be unique within one `<Components>` tag.
typeDefRefId	Reference of the type of the field.
pointer	Boolean flag on whether the field is a reference or not (optional field).

Each field can be an array. This is indicated with the subelement `<Dimensions>` that contains a list of `<Dimension>` elements, each with the following attributes:

Name	Description
number	The index of the dimension.

Name	Description
size	The size (number of elements) of that dimension.
valueMacroRefId	Instead of the size a reference to the value macro defining the size.

Enumeration Items (enum)
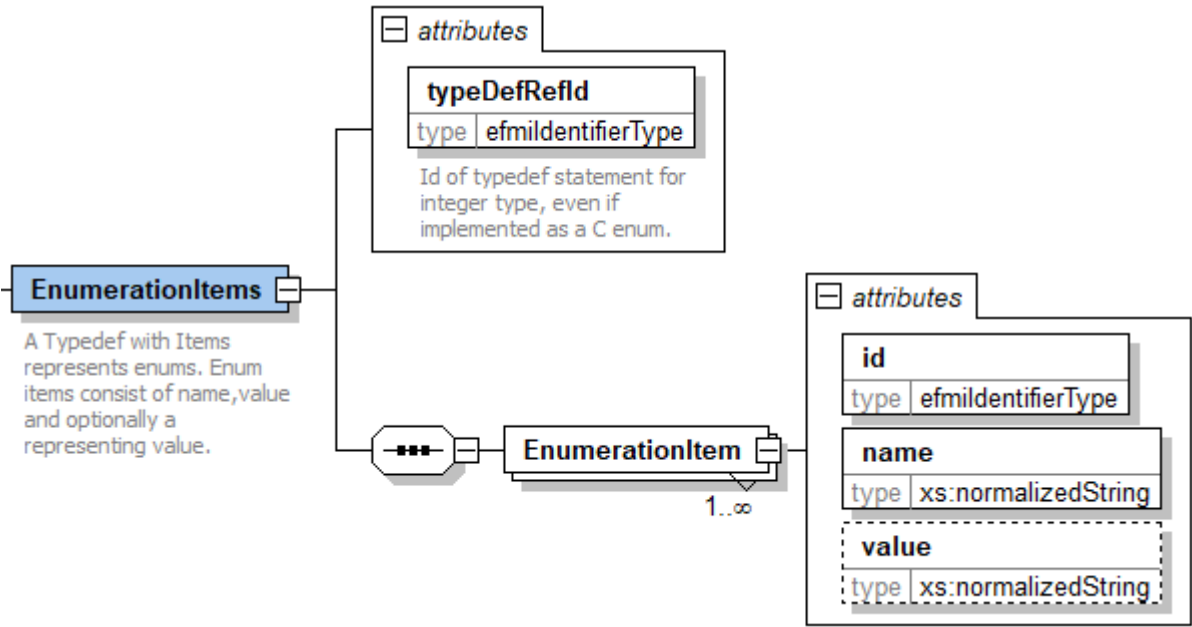
`<EnumerationItems>` declares an enumeration type with the list of enumeration items. Each `<EnumerationItem>` has the following fields
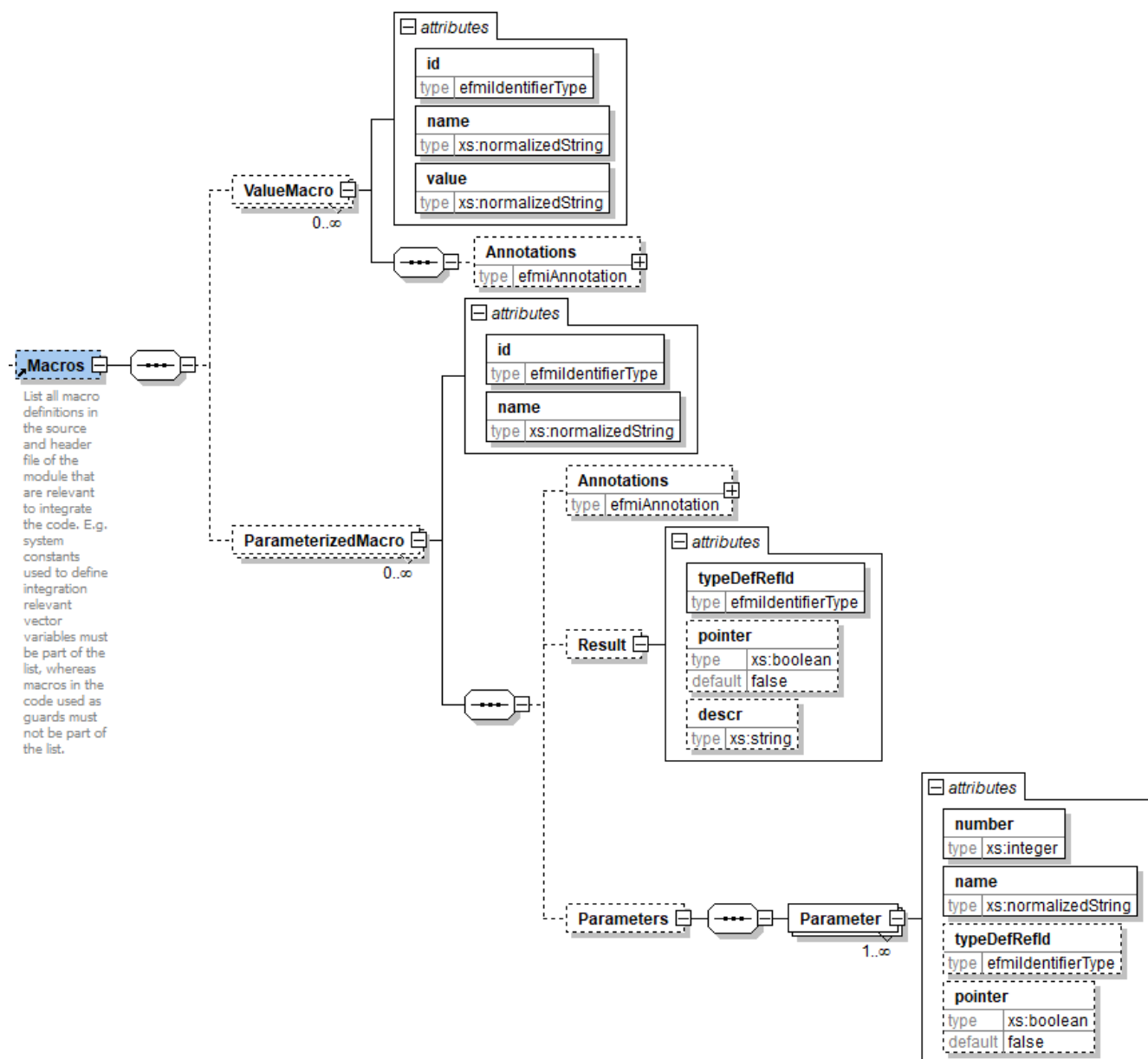
Name	Description
id	Unique id.
name	Name of the enumeration literal. This name must be unique within an enumeration definition (`<EnumerationItems>`)
value	Encoded value (this field is optional).

Macros

Here all macro definitions in the source and header file of the module are listed that are relevant to integrate the code. For example system constants used to define integration relevant vector variables must be part of the list, whereas macros in the code used as guards must not be part of the list.

There are two kind of macros "ValueMacro" and "ParameterizedMacro". Both are contained as children in the "Macros" tag.

A value macros defines a symbol and assigns a value to it. The value must be a number

Name	Description
id	Unique id.
name	Name of the macro variable.
value	Concrete value of the macro variable.
Annotations	Additional data that a vendor might want to store and that other vendors might ignore. For details see Section 2.3.4.5.

A parameterized macro defines however only the signature of a macro with parameters. Thereby each parameter is given as a "Parameter" element with attrubtes for its name and its position (since xml is not guaranteed to be order-preserving). The positions must be the values 0 ... n-1 where n is the number of parameters.

Name	Description
name	Name of the macro argument.

Name	Description
Number	Position of the macro argument.

The following example shows the declaration of a value and a parametrized macro

```
1    <Macros>
2        <ValueMacro id="VM_1" name="num_Cyl" value="4"/>
3        <ParameterizedMacro id="PM_1" name="myMax">
4            <Parameter name="a" number="0">
5            <Parameter name="b" number="1">
6        </ParameterizedMacro>
7    </Macros>
```
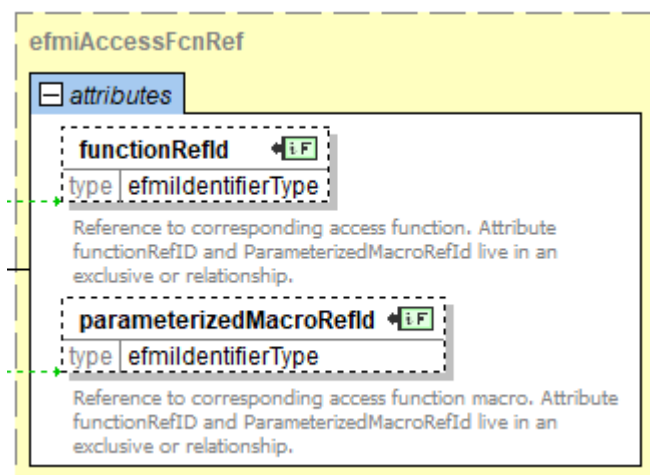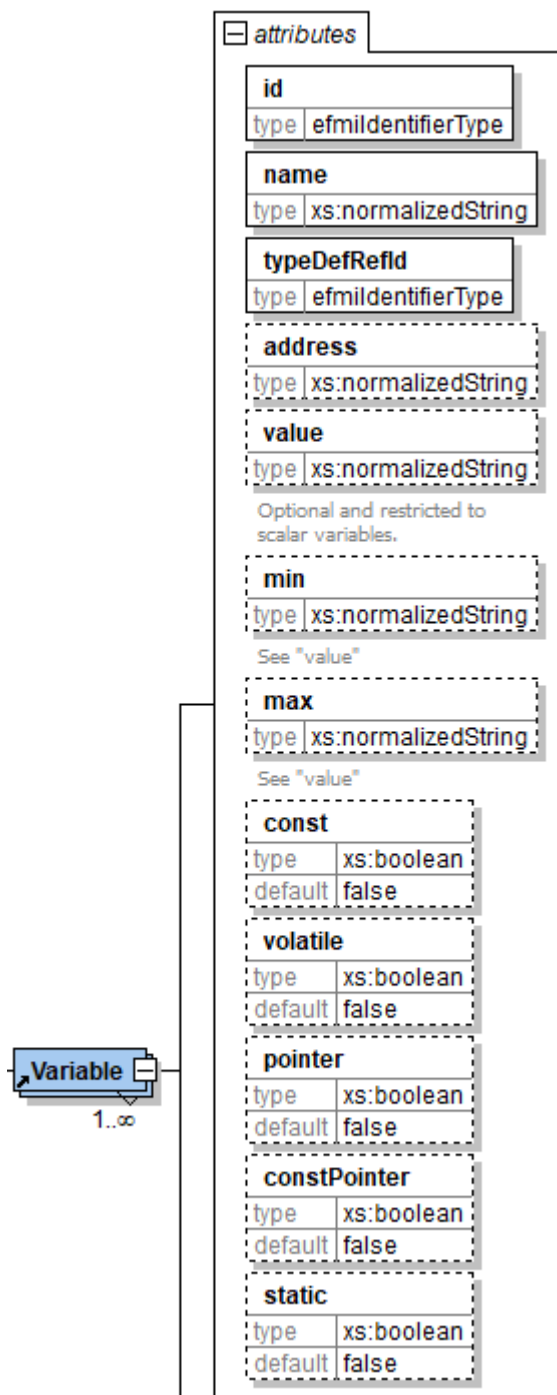
Variables

<Variable> elements are grouped in the <Variables> element.
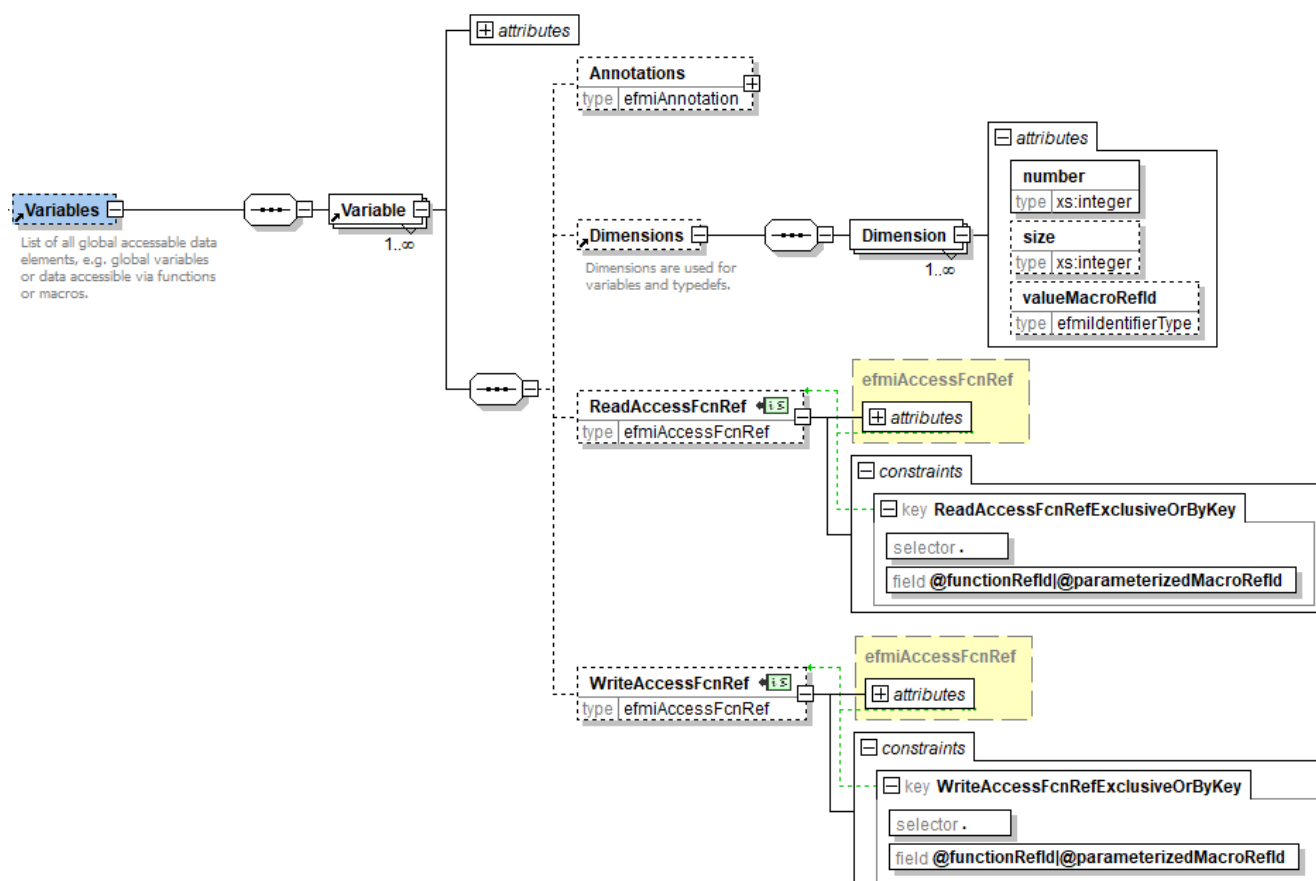
Each variable has the following attributes:

Name	Description
id	Unique id of the variable.
name	Name of the variable.
typedefRefId	id of the defined type of the variable.
address	Optional address.
value	Optional initial value of that variable that must be consistent which the initial value in Algorithm Code. Value might be different because of a decision to implement the Algorithm Code variable in a different datatype, for example Algorithm Code variable is `Float64` and Production Code variable is `Float32`.
min	Optional minimum value (see `value`).
max	Optional maximum value (see `value`).
const	Optional Boolean value on whether the variable is constant.
volatile	Optional Boolean value on whether the variable is volatile.
pointer	Optional Boolean value whether the variable is a pointer of the type or a variable of that type.
constPointer	Optional Boolean value whether the variable is a const pointer.
static	Optional Boolean value on whether the variable is static.

Similar like a field in a `<Component>` a `<Variable>` can also be multidimensional by adding the `<Dimensions>` element. The following example defines a 2x2 array of variables with name `"T"`.

```
1    <Variable id="V_33" name="T" typeDefRefId="TD_F64" pointer="false" value="0.1" const="false"
    volatile="true" static="false">
2        <Dimensions>
3            <Dimension number="0" size="2">
4            <Dimension number="1" size="2">
5        </Dimensions>
6    </Functions>
```
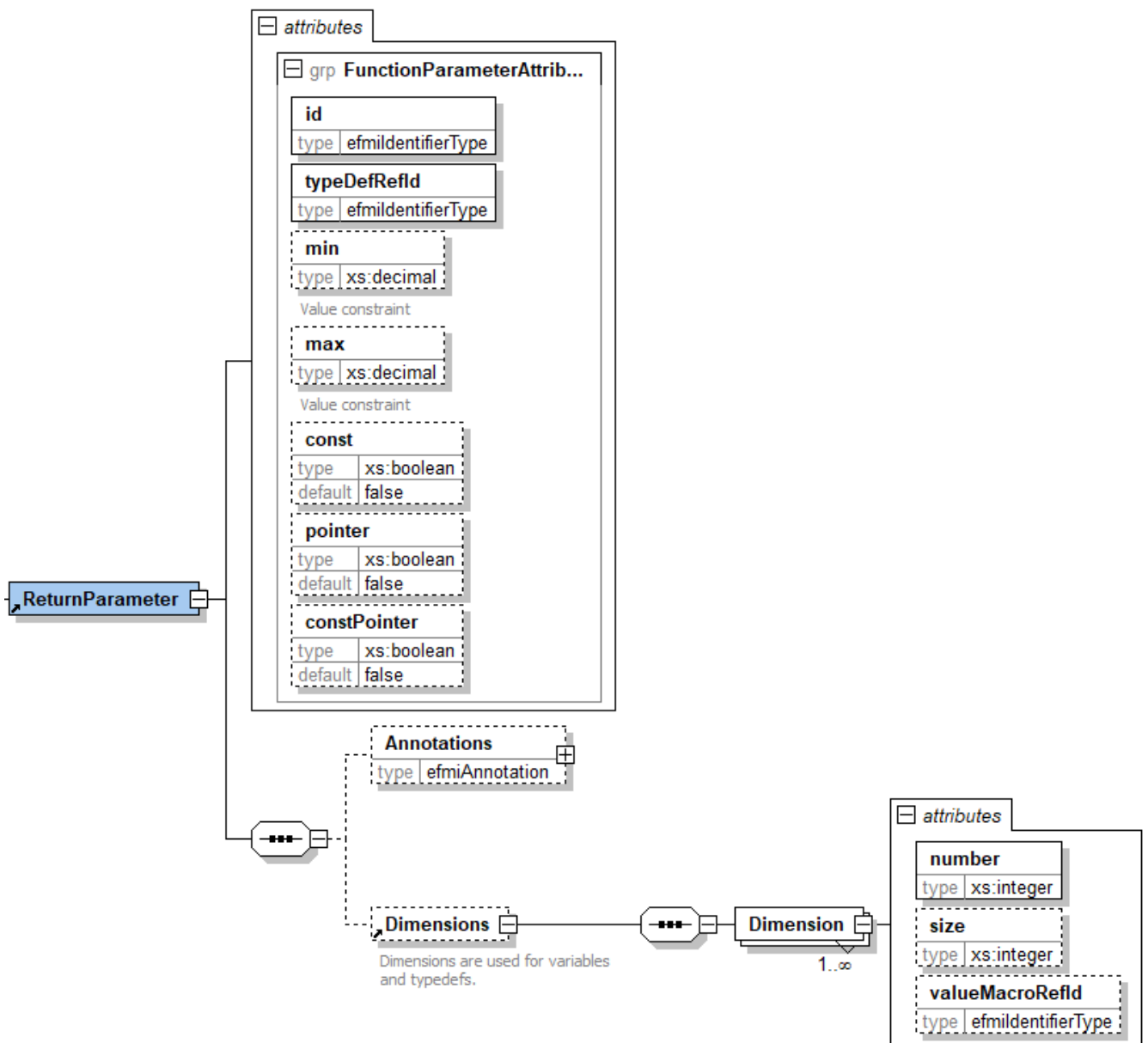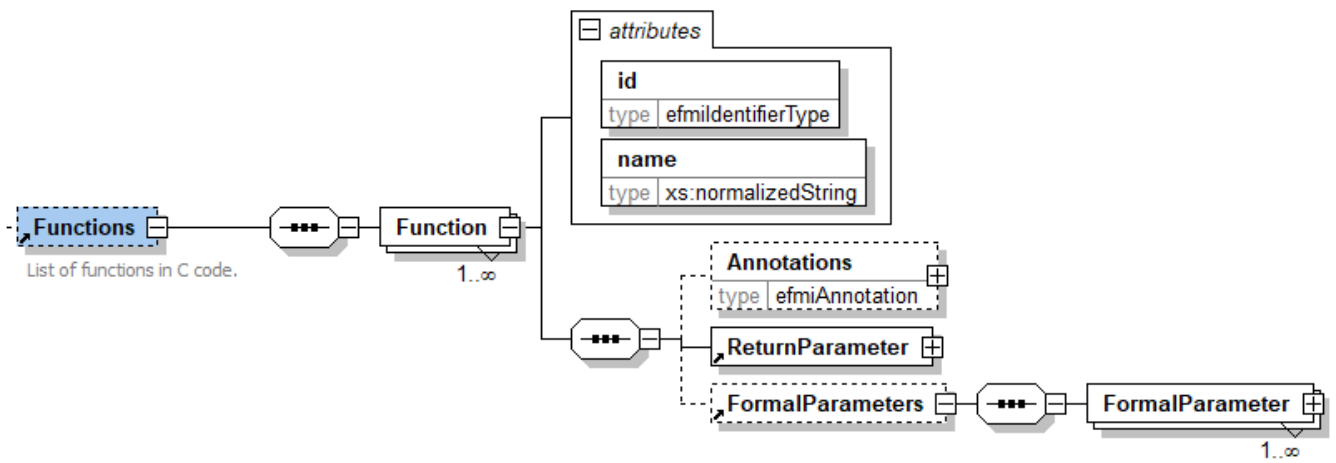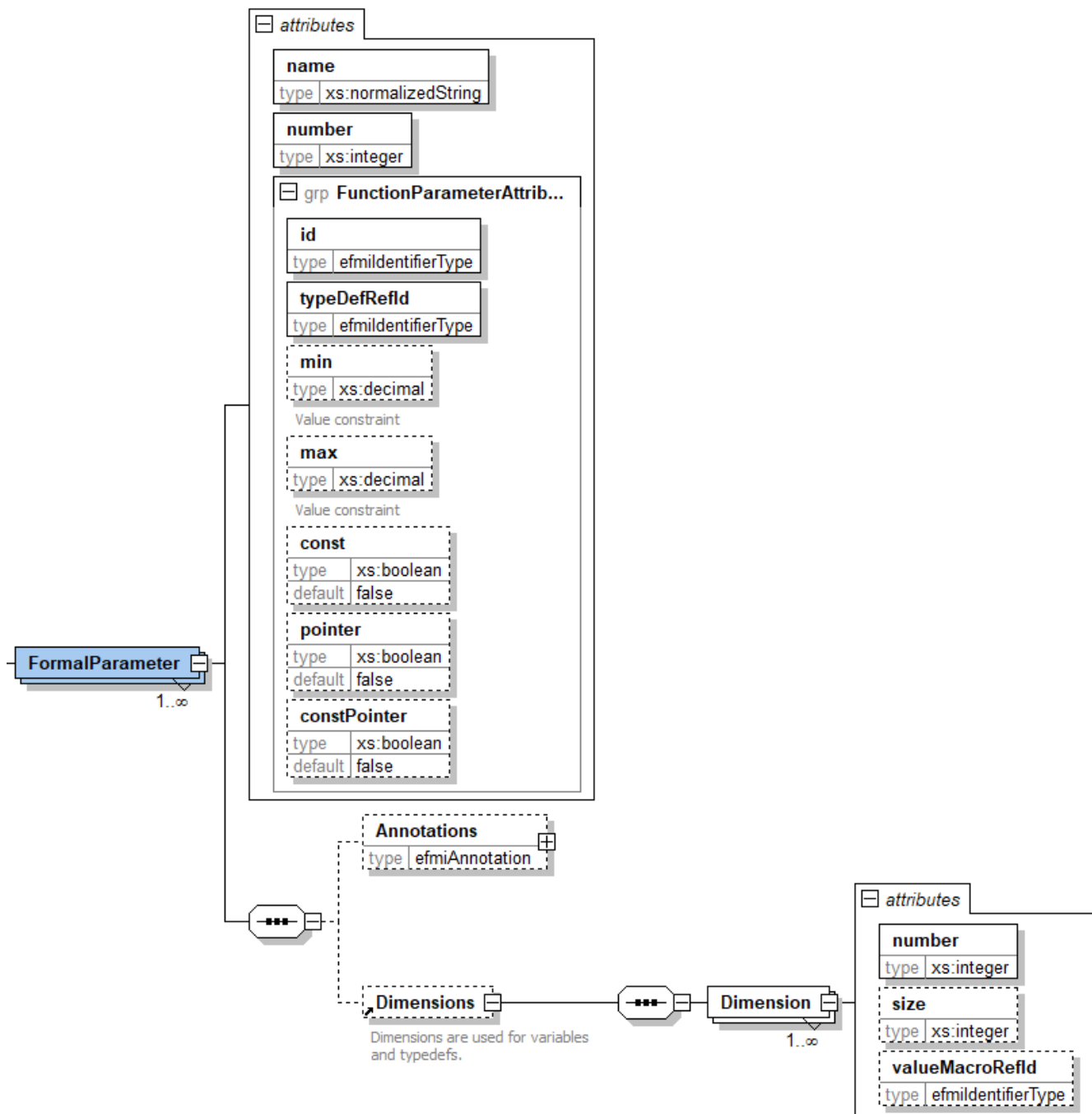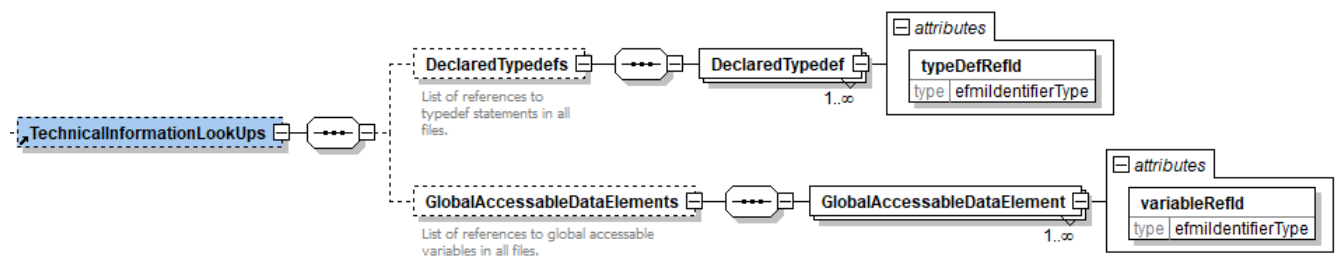
Functions

The described functions of (production) code files are grouped in the "Functions" tag. Each function has an `"id"` and a `"name"`. In addition it has a subelement for the return parameter (if the function is void, the subelement is not present) and a list of "formal parameter". The return parameter (if present) and the formal parameters list.

Functions

List of functions in C code.

Function 1..∞

attributes

id
type | efmiIdentifierType

name
type | xs:normalizedString

Annotations
type | efmiAnnotation

ReturnParameter

FormalParameters — **FormalParameter** 1..∞

ReturnParameter

attributes

grp **FunctionParameterAttrib...**

id
type | efmiIdentifierType

typeDefRefId
type | efmiIdentifierType

min
type | xs:decimal
Value constraint

max
type | xs:decimal
Value constraint

const
type | xs:boolean
default | false

pointer
type | xs:boolean
default | false

constPointer
type | xs:boolean
default | false

Annotations
type | efmiAnnotation

Dimensions
Dimensions are used for variables and typedefs.

Dimension 1..∞

attributes

number
type | xs:integer

size
type | xs:integer

valueMacroRefId
type | efmiIdentifierType

Example:

```
 1  <Functions>
 2      <Function id="Func_1" name="doStep">
 3          <FormalParameters>
 4              <FormalParameter id="V_33" name="T" number="0" typeDefRefId="TD_F64">
 5          </FormalParameters>
 6      <Function/>
 7      <Function id="Func_2" name="doStep2">
 8          <ReturnParameter id="Func_2_ret" typeDefRefId="TD_F64" pointer="false">
 9      <Function/>
10  <Functions/>
```

5.1.4. Technical Information Lookups

Facilitates a quick access to information in the manifest and the associated C files.

Name	Description
DeclaredTypeDefs	List of all typedef statements in C code
GlobalAccessableDataElements	List of all global variables and global available access functions

Both lists consist of elements, `DeclaredTypedef` and `GlobalAccessableDataElement` respectively, that only have a reference attribute to a certain kind of element.

Attribute of `DeclaredTypedef`:

Name	Description
typeDefRefId	Reference to a TypeDef element in the manifest.

Attribute of `GlobalAccessableDataElement`:

Name	Description
variableRefId	Reference to a Variable element in the manifest.

5.1.5. Logical Data

Defines how the logical elements (variables, functions etc.) are mapped to the actual data structures and elements of functions and defined variables.

The description in the code files basically describes only Production Code parts. As shown in the beginning of this section the mapping to the Algorithm Code is sometimes not obvious, for example because variables in the Algorithm Code do only appear as arguments or are may be part of structures or arrays. Therefore we describe this mapping explicitely.

The mapping is given in the element `LogicalData` which contains the `DataReferences` and the `FunctionReferences`.
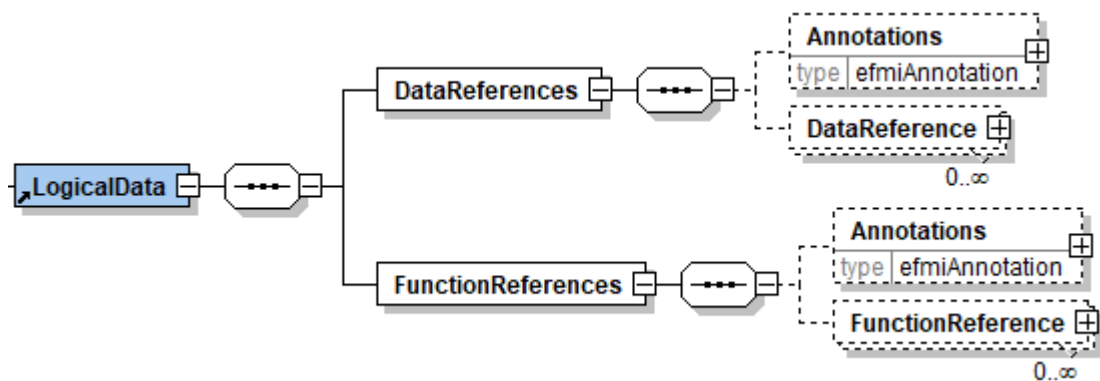
A `DataReference` itself contains the following attributes and elements to identify the variable in the Production Code and the mapped variable in the Algorithm Code
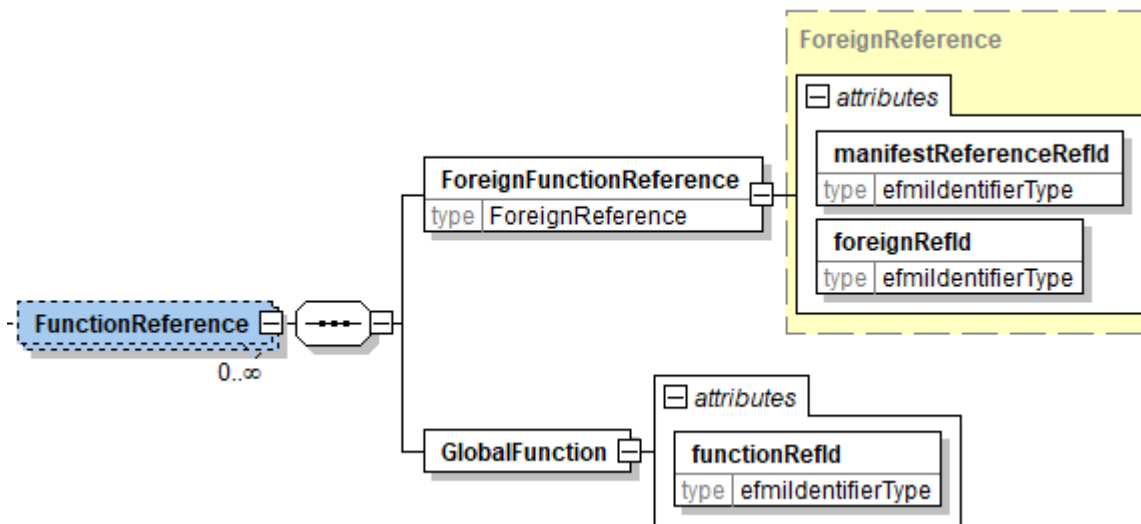

Name	Description
ForeignVariableReference	Subelement of type `ForeignReference` to the element in the Algorithm Code.
GlobalVariable	Reference to a declared global accessible variable in the current manifest. If the referenced variable is of a complex type, the `componentIdentifier` gives the "path" within that complex variable. The "." is used as component separator, brackets are used for array index, e.g. "a.b[3].c" means that the refered variable has a field "a" that itself contains a field "b" which is an array of a complex type that contains a field "c".

Name	Description
FormalParameter	Reference to a formal parameter of a global accessible function by the `formalParameterRefId` attribute in the current manifest. If the referenced parameter is of a complex type, the `componentIdentifier` gives the "path" within that complex parameter. The "." is used as component separator, brackets are used for array index, e.g. "a.b[3].c" means that the referred parameter has a field "a" that itself contains a field "b" which is an array of a complex type that contains a field "c".

A `FunctionReference` is similar to the DataReferences mapping Algorithm Code functions, mainly the block interface functions, to functions in the Production Code.

Name	Description
ForeignFunctionReference	Subelement of type `ForeignReference` to the element in the Algorithm Code.
GlobalFunction	Reference to a declared global accessible function in the current manifest by `functionRefId` attribute.

5.2. Production Code Language

A Production Code Model Representation includes code files that are modules in terms of the C or C++ programming language.

The C programming language is described in [KR78] and in a destilled version in [CLangWiki]. A similar description of the C++ programming language gives [Str13] or as a destilled version [CPPLangWiki].

For both programming languages, the Motor Industry Software Reliability Association (MISRA) has published a set of guidelines to facilitate code safety, security, portability and reliability in the context of embedded software systems, see [MISRA12], [MISRA08]. In cases where the C code is not hand-coded but generated by a tool different guidelines [MISRA04] shall be fulfilled.

An example is the calling of an algorithm to solve a scalar nonlinear function, where a function

pointer and a void pointer for the context is passed. (This is necessary, as the function depends on the internal state of the model.)

```
int solveOneNonlinearEquation (Real_t (*f_Nonlinear)(Real_t u, void* data), Real_t u_min, Real_t
u_max,
                         Real_t tolerance, Real_t *u, void *data)
```

This could be called from C Code, e.g., by

```
err = solveOneNonlinearEquation(my_f_Nonlinear, 1.0, 8.0, tol, &u, &mydata);
```

where the function 'my_f_Nonlinear' is defined by

```
Real_t f_Nonlinear_3(Real_t u, void *data) {
    myDataType *mydata = (myDataType*)data;

    return  mydata->p[0] + log(mydata->p[1]*u) - u;
}
```

This is considered safe for the usage for auto-generated code, where the void pointer is passed together with a function pointer to the function that uses this void pointer as one of its arguments.

For individual Production Code sections, compliance with Coding Guidelines like MISRA:2012 is annotated in the manifest xml-File.

Common for both languages is that especially for resource limited embedded systems a number of language features are limited or at least not available. For example:

- dynamic memory handling
- only compile-time fixed array sizes
- functions typically offered by operating system
- availability of mathematical functions
- no runtime type information
- ...

Both languages are standardized by the International Organization for Standardization (ISO) and the following table lists an excerpt of different standards and their informal name(s):

Reference	Name(s)
ISO/IEC 9899:1990	ANSI C, ISO C, C89, C90
ISO/IEC 9899/AMD1:1995	C95
ISO/IEC 9899:1999	C99
ISO/IEC 9899:2011	C11

ISO/IEC 9899:2018	C18
ISO/IEC 14882:1998	C++98
ISO/IEC 14882:2003	C++03
ISO/IEC 14882:2011	C++11, C++0x
ISO/IEC 14882:2014	C++14, C++1x
ISO/IEC 14882:2017	C++17, C++1z

A Production Code Model Representation must indicate the actually used language and standard of the modules in the manifest file.

Chapter 6. Binary Code Model Representation

The Binary Code Model Representation is intended to be a container to exchange software artifacts in binary form. Such binaries can be directly integrated with other embedded software running on an ECU. The main purpose of this format is the protection of intellectual property. Shareholders can exchange a software solution without revealing crucial implementation or algorithm details to the user of a particular solution. Beside the protection of intellectual property, the Binary Code Model Representation also provides protection of integrity of the solution. The software solution cannot be altered except for the intended interface such as calibration parameters. Furthermore the binary representation unitizes separate functionalities into dedicated binary files. These binary files can be used independently in different contexts.

An eFMU container might consist of multiple Binary Model Representations which may originate from the same Production Code Model Represention.
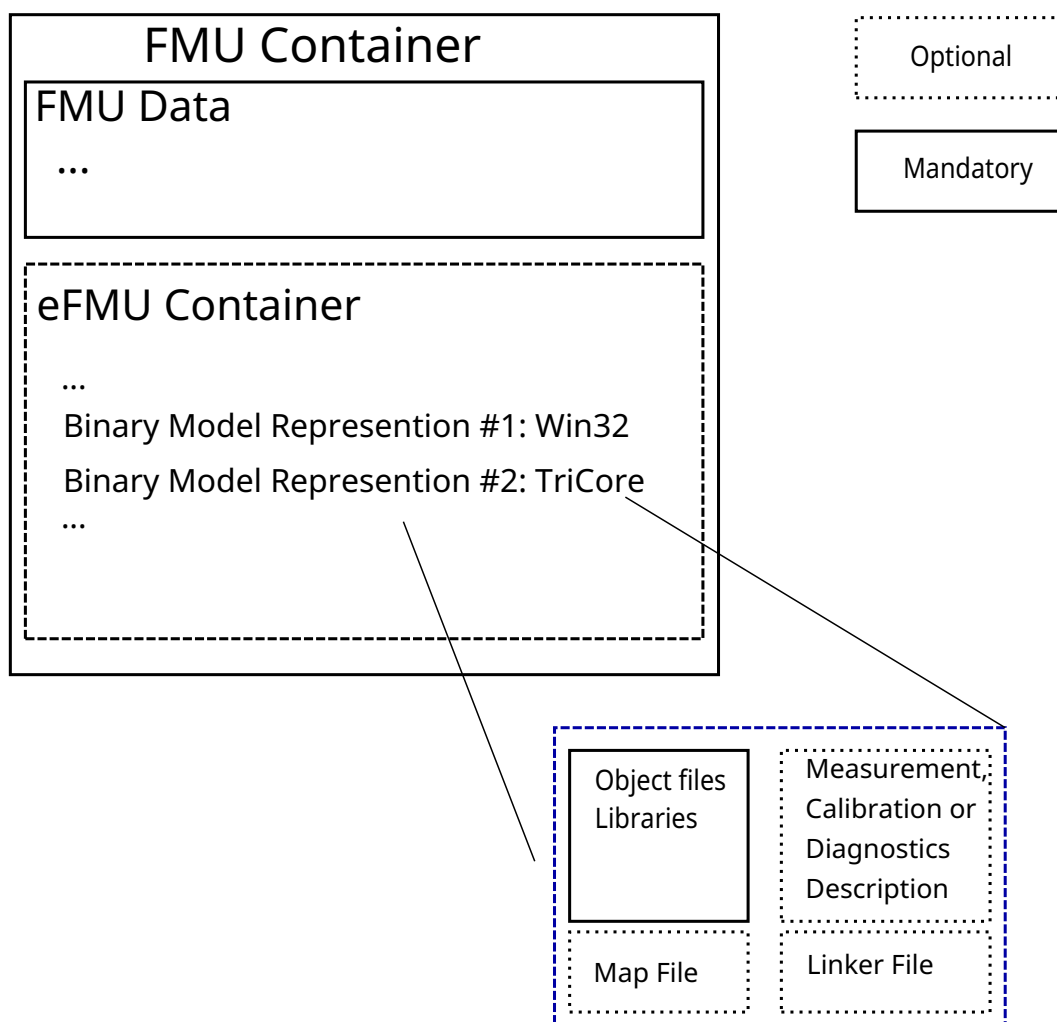

Figure 1. Structure of Binary Model Represention

A Binary Code Model Representation consists at least of the following items:

- Object files or static libraries in Executable and Linking Format (ELF) for the use for embedded devices or dynamic linked libraries for co-simulation purposes in Windows environments
- Container manifest

Furthermore, it might include a file containing information necessary for calibration, measurement and diagnosis purposes and a linker script that contains the necessary information in order to link the software for a particular target.

6.1. Manifest

Since a binary container is subject to an integration on a particular target ECU, its manifest has to provide any necessary information about

- the components interface,
- the compiler and its configuration,
- the linker and its configuration,
- the target

Optionally, there might exists

- information about the run time behavior
- meta information regarding the source code (e.g. MISRA Compliance, Code Quality reports, etc.)
- Calibration

The Binary Code manifest is an XML file with structured information about the Binary Model Representation.

NOTE	Some of the above points are already available in the Production Code Model Representation. Such information (interface, MISRA Compliance) will be referenced by the Binary Code manifest from the Production Code manifest.

6.1.1. Structure of the Manifest

The Binary Code manifest:
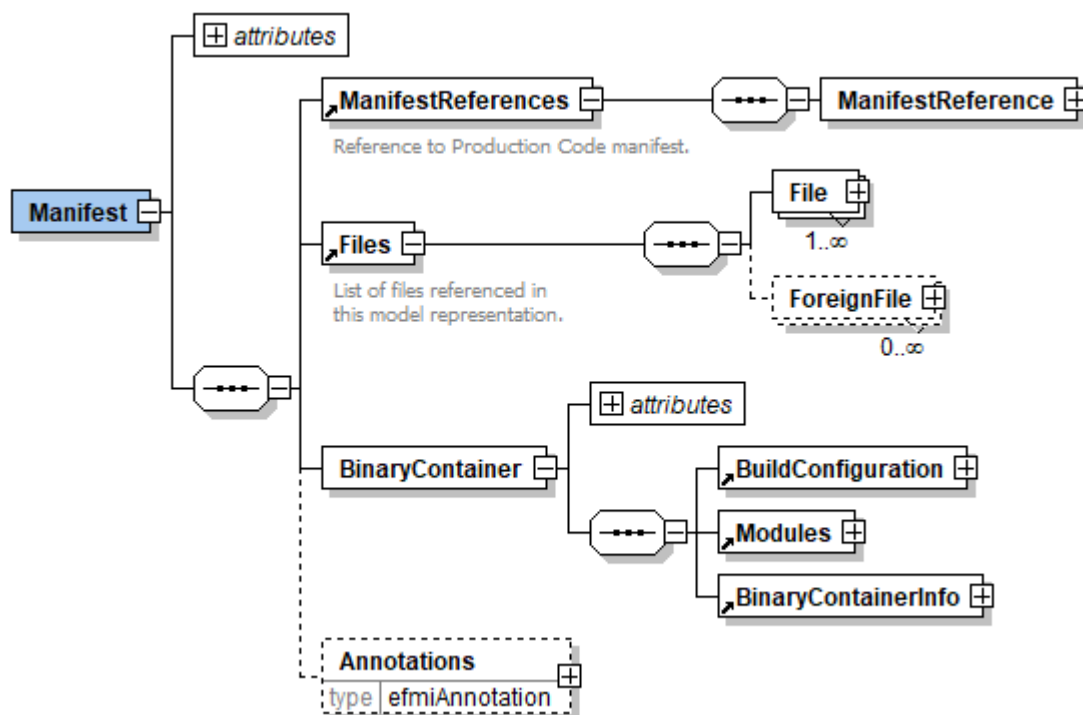
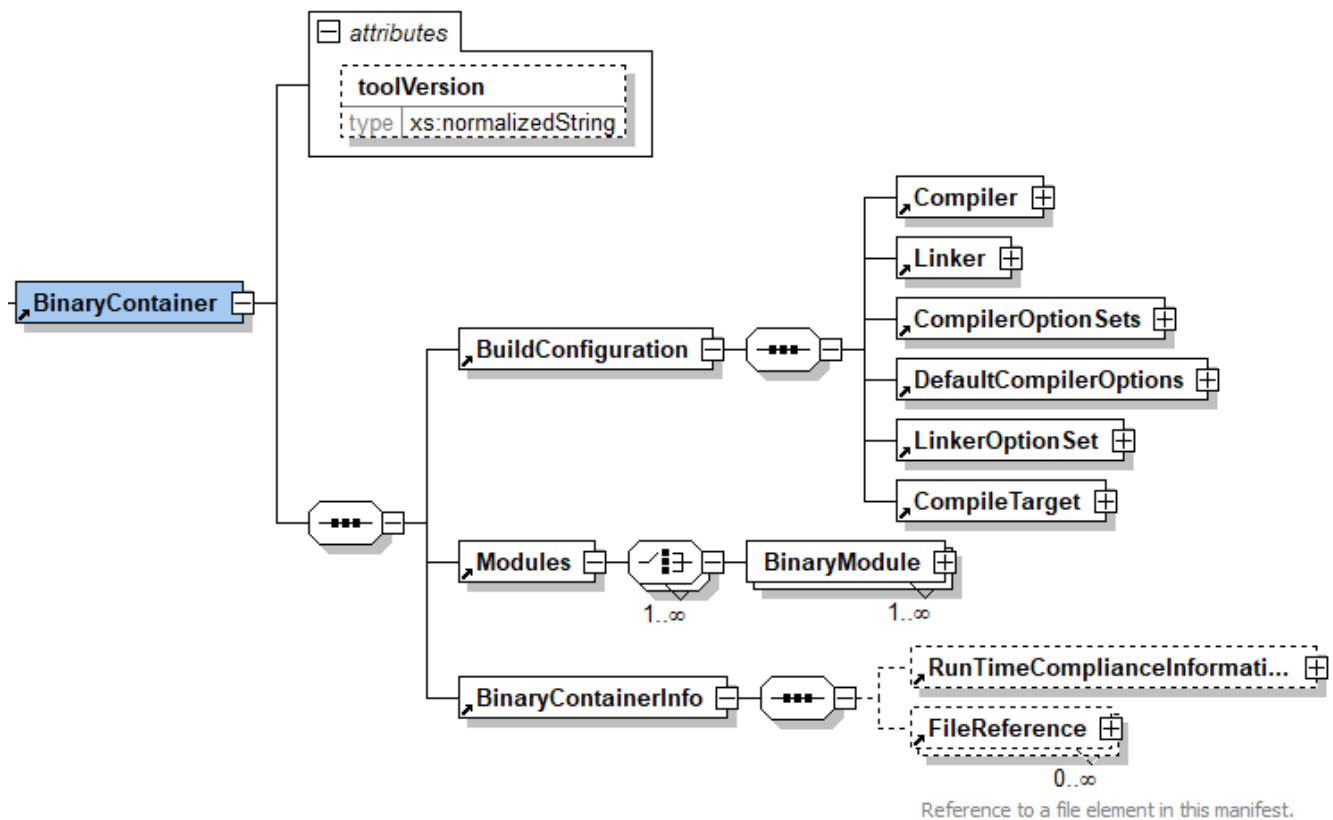consists of the following elements:

On the top level, the schema consists of the following elements:

Name	Description
attributes	The attributes of the top-level element are the same for all manifest kinds and are defined in section Section 2.3.1. Current kind-specific values: `kind = "BinaryCode"`, `xsdVersion` (value is the current xsd version of the schema for the Binary Code model manifest).
ManifestReferences	Reference to the manifest of the Production Code on which this Binary Code manifest is based on. This element is the same for all manifest kinds and is defined in section Section 2.3.4.3.
Files	List of files referenced in this model representation. This element is the same for all manifest kinds and is defined in section Section 2.3.3.
BinaryContainer	Defines the essential content of the actual container. For details see Section 6.1.2.
Annotations	Additional data that a vendor might want to store and that other vendors might ignore. For details see Section 2.3.4.5.

The following subsections focus on the `BinaryContainer` element which represents the actual Binary Model Representation.

6.1.2. Binary Container

Element `BinaryContainer`

consists of the following elements:

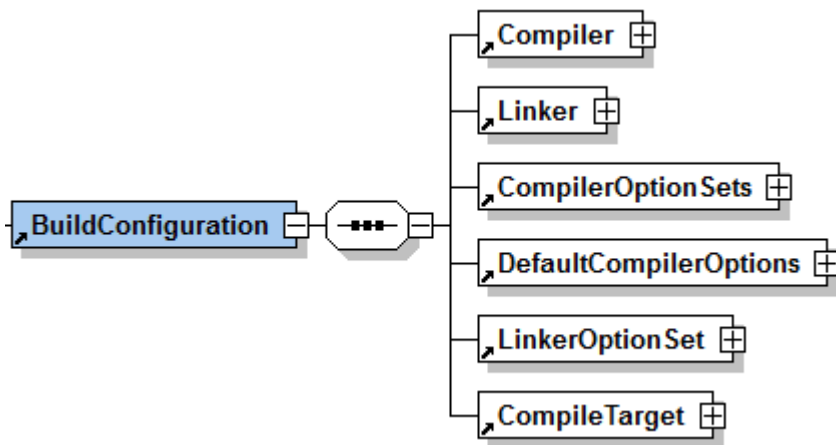| Name	Description
BuildConfiguration	The BuildConfiguration describes the actual build environment used to create the binary objects in the container. For more details see Section 6.1.2.1.
Modules	The Modules section describes all relevant binaries and source code references required or available for the binary model representation container. For more details see Section 6.1.3.
BinaryContainerInfo	The BinaryContainerInfo element contains additional and optional information relevant to the end user. For more details see Section 6.1.4.

Each of the above listed elements has to exist exactly once in a BinaryContainer. Additionally, the the BinaryContainer has the following Attributes:

Name	Description
toolVersion	This attribute is used by the the generating tool to store its Name and Version.

BuildConfiguration

Element BuildConfiguration consists of all information related to the compilation and linking of the model representation:

This element contains *exactly one* of each of the following elements:

Name	Description
Compiler	This element unambigously describes the compiler that has been used to create the binary artifacts. For details see Section 6.1.2.2.
Linker	This element unambigously describes the linker that has been used to create the binary artifacts. For details see Section 6.1.2.3.
CompilerOptionSets	This element stores all possible compiler settings used to create any binary element in the container. For details see Section 6.1.2.4.
DefaultCompilerOptions	This element refers to a CompilerOptionSet that has to be used to create the binary. For details see Section 6.1.2.5.
LinkerOptionSet	This element describes the relevant linker option for the above linker that has been used to create the binary object. For details see Section 6.1.2.6.
CompileTarget	This element describes the target platform, the binary has been compiled for. For details see Section 6.1.2.7.

NOTE	It is possible that a Binary Code Model Representation needs to be combined with some source from the Simulation Code, Tool-specific code of the Production Code model or even from external generators in order to analyze, integrate or test the model. In such cases additional sources need to be compiled and linked together. To support such a use case, the BuildConfiguration of a Binary Model Representation needs to provide all required information to be able to compile and link additional sources with the binary artifacts.

Compiler

In order to integrate the object code, it is required to have all relevant information about the compile process of a binary specified. Hence, the compiler is to be specified in the manifest as follows:
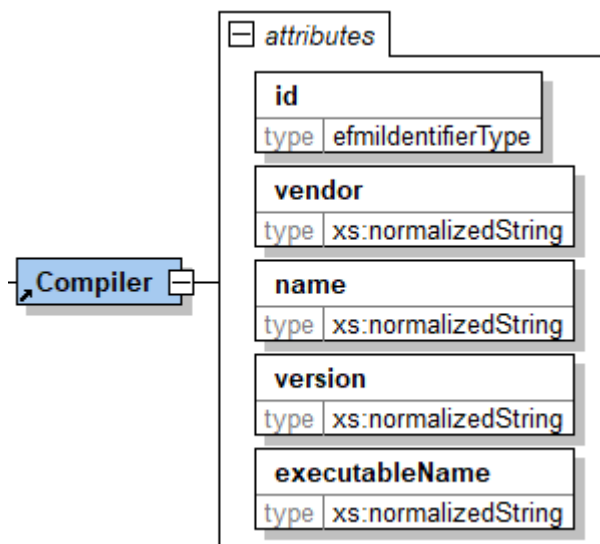
All attributes are mandatory and are defined as follows:

Name	Description
id	A unique `id` that has to be referenced by any corresponding `CompilerOptionSet`.
vendor	The name of the Company/Vendor that has created or issued the compiler.
name	A unique, unambiguous name of the compiler or compiler suite.
version	The specific version of the above compiler that has been used to create the binary.
executableName	The name of the actual executable of the compiler (suite).

The attributes `vendor`, `name` and `version` must clearly identify a particular compiler. Furthermore, it should be possible to use the value `executableName` together with a matching `CompilerOptionSet` to automatically compile a source file.

NOTE	The following example depicts a compiler configuration for a target compiler for the TriCore processor archtecture.

```
1  <Compiler id="ID_1000001" vendor="Altium"
2          name="TASKING VX-toolset for TriCore: C compiler"  version="v4.2r2"
   executableName="ctc"/>
```

Linker

Similar to the definition of the compiler infrastructure and options, the linker and link options have to be declared to be known to the integration engineer.
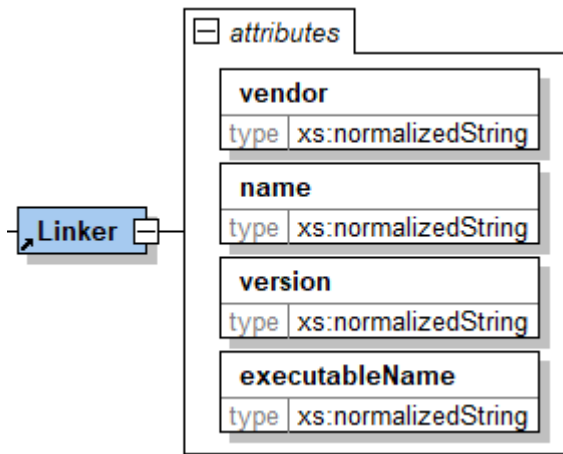
All attributes are mandatory and defined as follows:

Name	Description
vendor	The name of the Company/Vendor that have created or issued the linker.
name	Unique, unambiguous name of the linker .
version	The specific version of the above linker that have been used to create the binary.
executable Name	The name of the actual executable of the linker (suite).

The attributes vendor, name and version must clearly identify a particular linker. Furthermore, it should be possible to use the value executableName together with the below defined LinkerOptionSet to automatically link object files together.

| NOTE	The following example depicts an linker configuration for the TriCore processor architecture.

```
1  <Linker id="ID_1000002" vendor="Altium" name="TASKING VX-toolset for TriCore: object linker"
   version="v4.2r2" executableName="ltc"/>
```
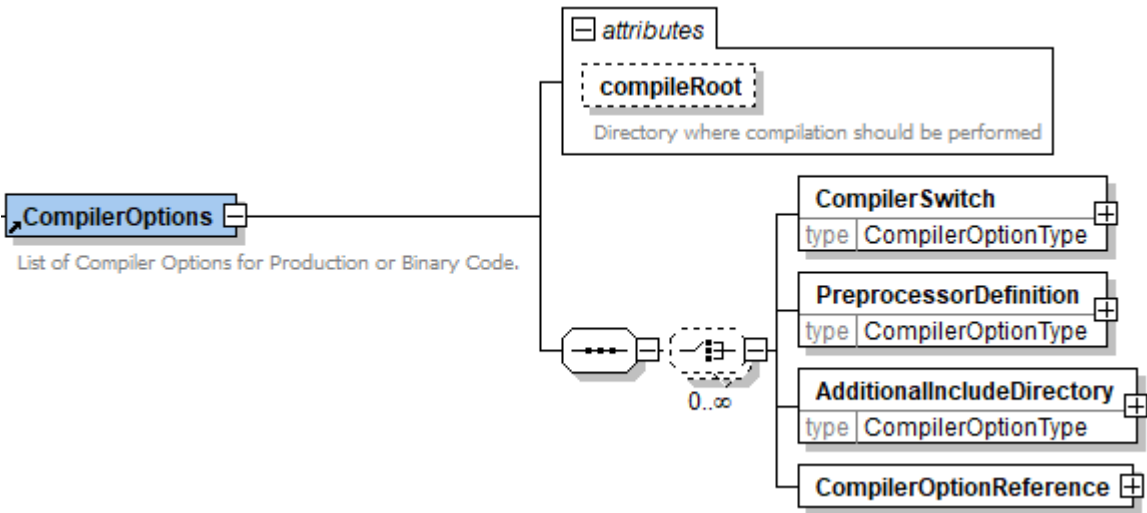
CompilerOptionSets

The CompilerOptionSets contains one or more CompilerOptionSet which defines settings and switches used to create at least one of the contained binary artifacts.
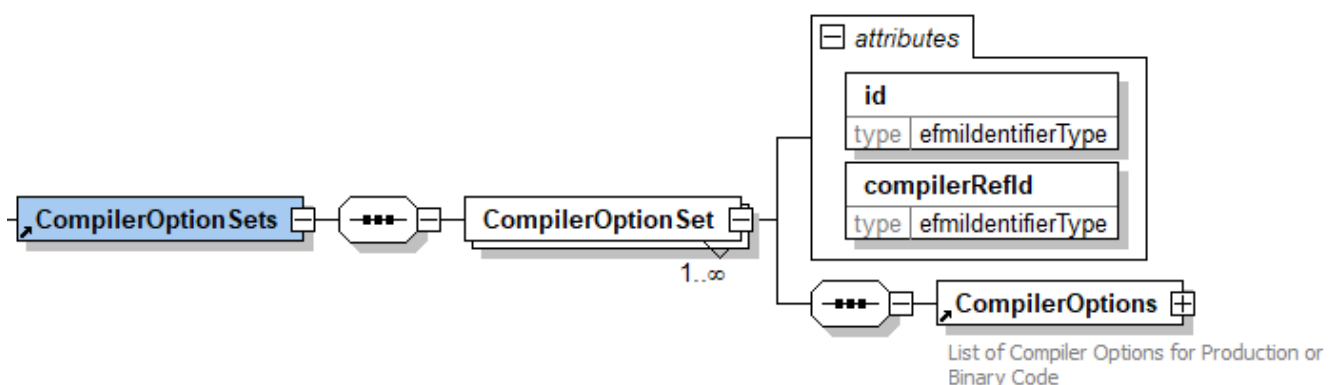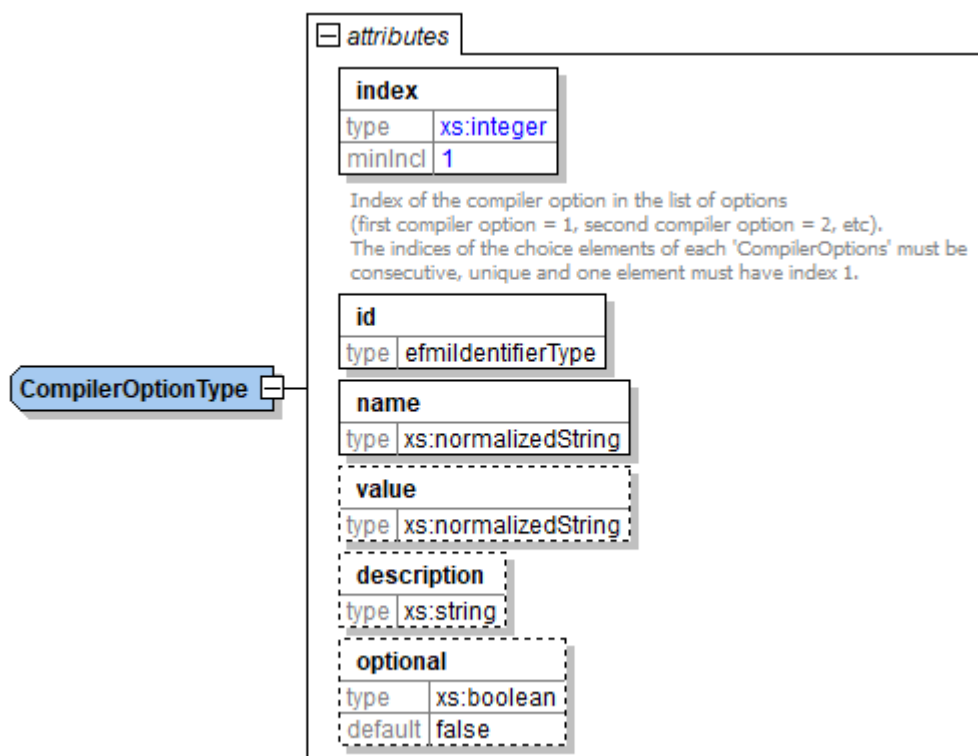
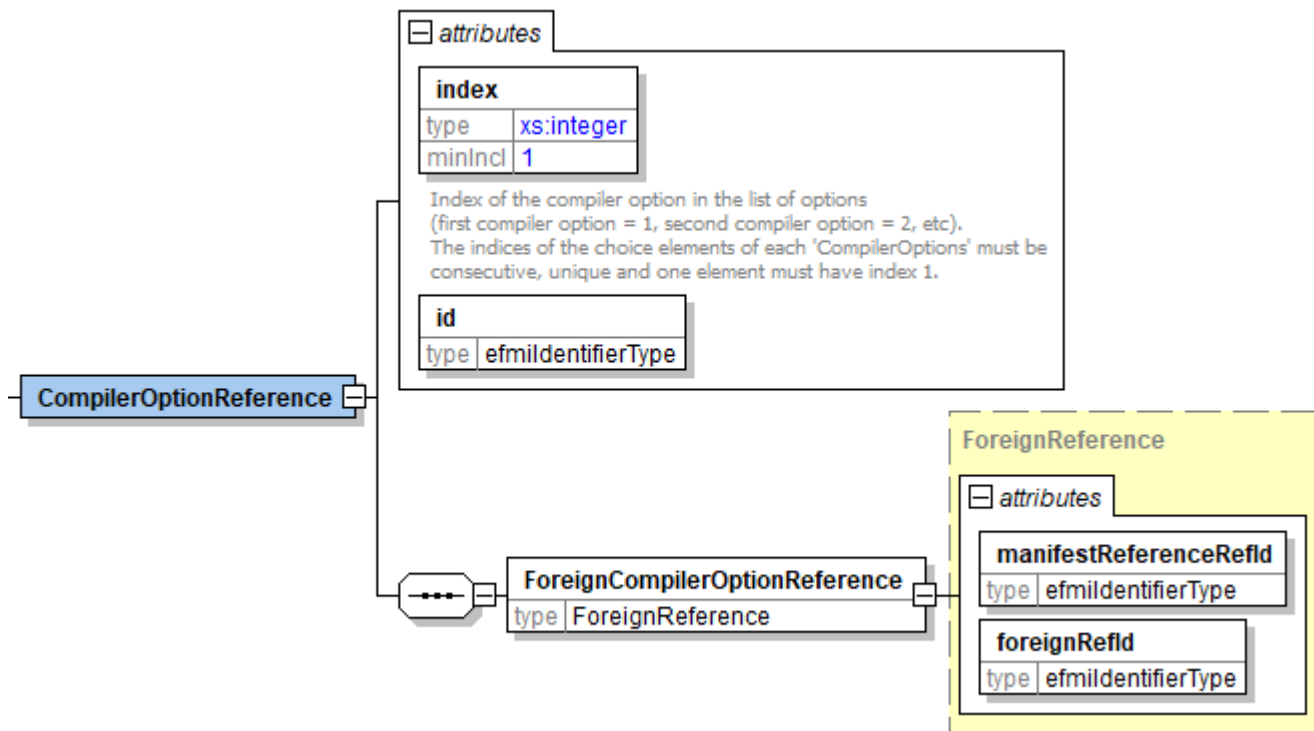| Name	Description
id	The unique identifier of the the `CompilerOptionSet` within the manifest.
compilerRefId	A reference to a configured compiler for the Compilers Section.
CompilerOptions	List of compiler options for Production or Binary Code, see [CompilerOptions]

The `CompilerOptions` list is defined as:

Name	Description
compileRoot	Directory where compilation should be performed.
CompilerSwitch	The compiler switches of type [CompilerOptionType].
PreprocessorDefinition	Preprocessor definitions of type [CompilerOptionType].
AdditionalIncludeDirectory	Additional include directory of type [CompilerOptionType].
CompilerOptionReference	A list of option references, see [CompilerOptionReference].

The `CompilerOptionType` attributes are defined as:

Name	Description
index	Index of the compiler option in the list of options (first compiler option = 1, second compiler option = 2, etc). The indices of the choice elements of each 'CompilerOptions' must be consecutive, unique and one element must have index 1.
id	Unique id of compiler option.
name	Name of option.
value	Optional value of option.
description	Optional description of option.
optional	Optional Boolean with default `false`, defining whether the option is optional.

The `CompilerOptionReference` list is defined as:

Name	Description
index	Index of the compiler option in the list of options (first compiler option = 1, second compiler option = 2, etc). The indices of the choice elements of each 'CompilerOptions' must be consecutive, unique and one element must have index 1.
id	Unique id of option reference.
ForeignOptionReference	Reference to another manifest file of type ForeignReference. For details see Section 2.3.4.3.

NOTE	The following example depicts some of the options that have to be provided in order to compile code for the Infineon Tricore TC27x family. Most options are special to this compiler family.

```xml
1  <CompilerOptionSets>
2    <CompilerOptionSet id="ID_1001" compilerRefId="ID_1000001">
3      <CompilerOptions>
4        <CompilerSwitch>
5          <id>ID_100010</id>
6          <name>--iso</name>
7          <value>90</value>
8        </CompilerSwitch>
9        <CompilerSwitch>
10         <id>ID_100011</id>
11         <name>--align</name>
12         <value>4</value>
13       </CompilerSwitch>
14       <CompilerSwitch>
15         <id>ID_100012</id>
16         <name>--optimize</name>
17         <value>3</value>
```

```
18          </CompilerSwitch>
19          <CompilerSwitch>
20              <id>ID_100013</id>
21              <name>--tradeoff</name>
22              <value>4</value>
23          </CompilerSwitch>
24          <CompilerSwitch>
25              <id>ID_100014</id>
26              <name>--source</name>
27          </CompilerSwitch>
28          <CompilerSwitch>
29              <id>ID_100015</id>
30              <name>--error-file</name>
31          </CompilerSwitch>
32          <CompilerSwitch>
33              <id>ID_100016</id>
34              <name>--rename-sections=sect</name>
35          </CompilerSwitch>
36          <CompilerSwitch>
37              <id>ID_100017</id>
38              <name>--core</name>
39              <value>tc1.6.x</value>
40          </CompilerSwitch>
41          <CompilerSwitch>
42              <id>ID_100018</id>
43              <name>-Hsfr/regtc27x.sfr</name>
44          </CompilerSwitch>
45          <CompilerSwitch>
46              <id>ID_100019</id>
47              <name>--default-near-size</name>
48              <value>0</value>
49          </CompilerSwitch>
50  <CompilerOptionSets>
```

Default Compiler Options

While each module might have its own compiler options referenced from the `CompilerOptionsSets` of the `BinaryContainer`, a default option set for the container can be defined. The default compiler options are used in any case where no other `CompilerOptionsSet` is provided.

The `DefaultCompilerOptions` are specified as follows:
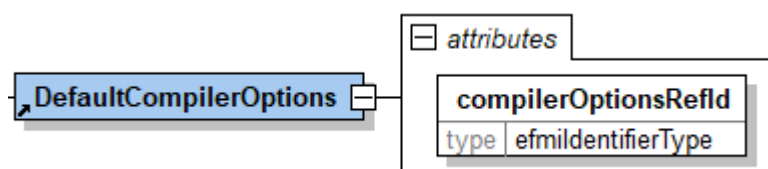
Name	Description
compilerOptionsRefId	Reference to a previously defined `CompilerOptionSet` to be used as default.

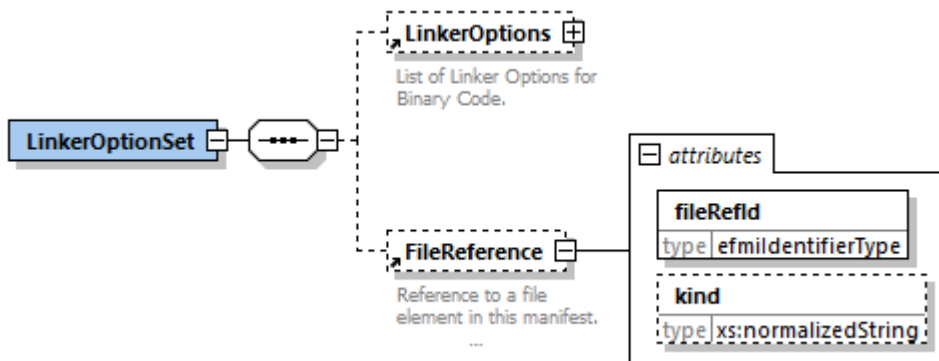| NOTE	The following example depicts an default option set that refers to the

`CompilerOptionSet` defined in the parent `BinaryContainer` element.

```
1  <DefaultCompilerOptions compilerOptionsRefId="ID_1001" />
```

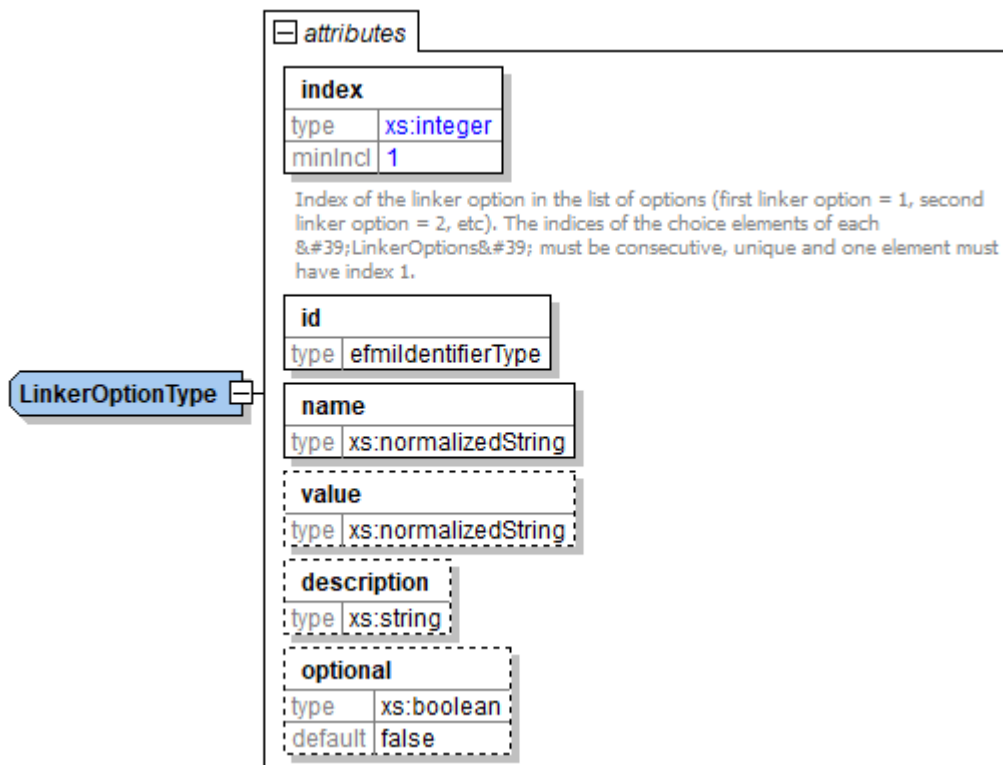**LinkerOptionSet**

The `LinkerOptionSet` contains one `LinkerOptions` which defines linker settings and switches.

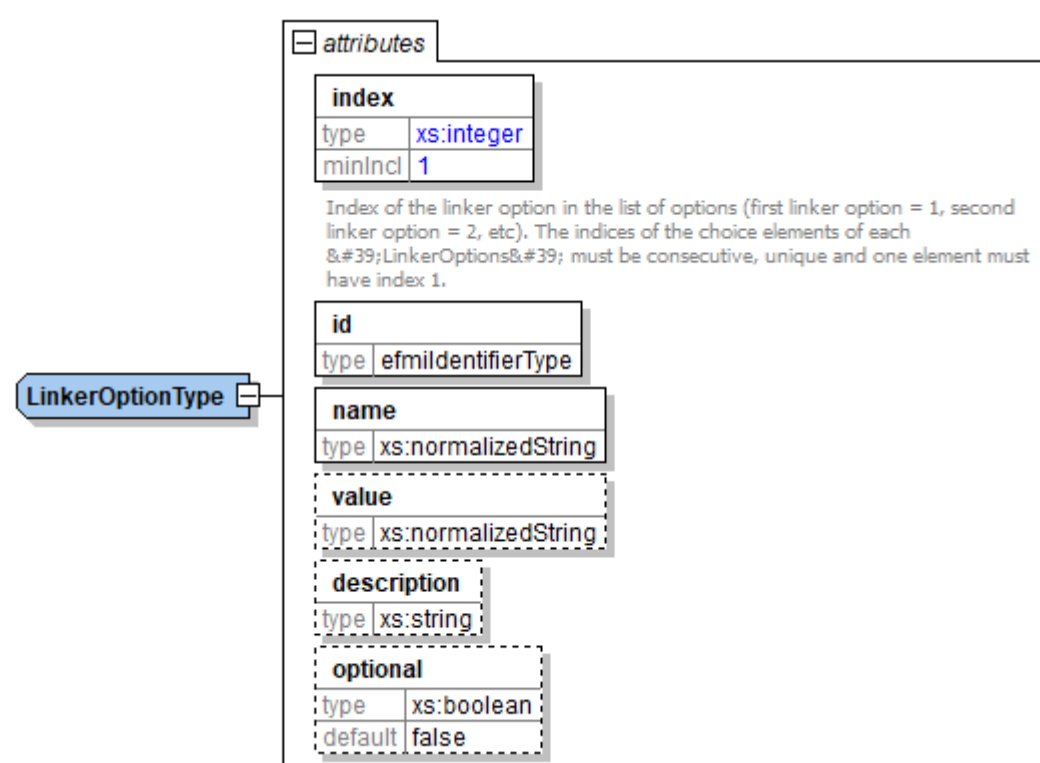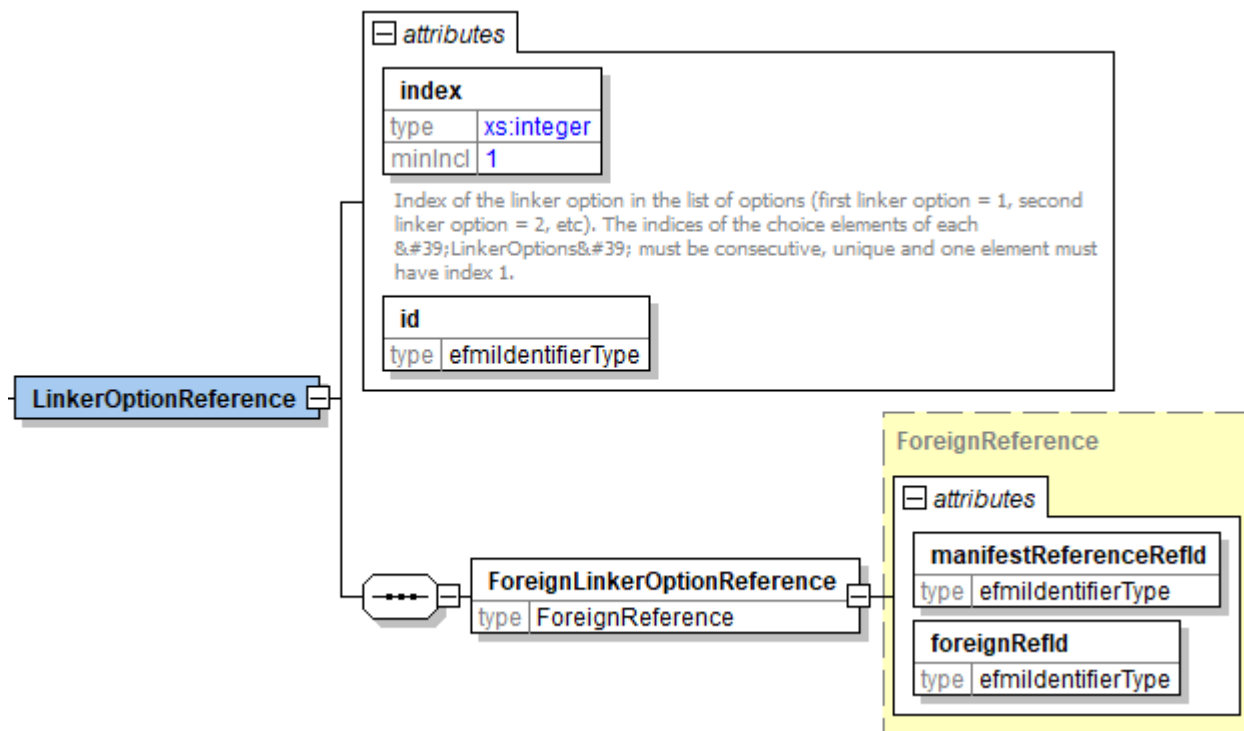| Name	Description
`LinkerOptions`	List of linker options for Production or Binary Code, see [LinkerOptions]
`FileReference`	The linker script is referenced with a `FileReference` element.

The `LinkerOptions` list is defined as:

Name	Description
LinkerSwitch	The linker switches of type [LinkerOptionType].
Library	Library of type [LinkerOptionType].
AdditionalLibraryDirectory	Additional library directory of type [LinkerOptionType].
LinkerOptionReference	A list of option references, see [LinkerOptionReference].

The `LinkerOptionType` attributes are defined as:

Name	Description
index	Index of the option in the linker command line.
id	Unique id of linker option.
name	Name of option.
value	Optional value of option.
description	Optional description of option.
optional	Optional Boolean with default `false`, defining whether the option is optional.

The `LinkerOptionReference` list is defined as:

Name	Description
index	Index of the option in the linker command line.
id	Unique id of option reference.
ForeignOptionReference	Reference to another manifest file of type ForeignReference. For details see Section 2.3.4.3.

NOTE	The following example depicts some of the options that have to be provided in order to compile code for the Infineon Tricore TC27x family. Most options are special to this linker family.

```
1  <LinkerOptionSet>
2    <LinkerOptions>
3      <LinkerSwitch>
4        <id>ID_100010</id>
5        <name>output</name>
6        <value>dummy.elf:ELF</value>
7      </LinkerSwitch>
8      <LinkerSwitch>
9        <id>ID_100011</id>
10       <name>no-warnings</name>
11     </LinkerSwitch>
12     <LinkerSwitch>
13       <id>ID_100012</id>
14       <name>incremental</name>
15     </LinkerSwitch>
16     <LinkerSwitch>
17       <id>ID_100013</id>
18       <name>lsl-file</name>
19       <value>TC277.lsl</value>
20     </LinkerSwitch>
```

```
21      <LinkerSwitch>
22        <id>ID_100014</id>
23        <name>map-file</name>
24        <value>mapfile.map</value>
25      </LinkerSwitch>
26    <LinkerOptions>
27    <FileReference fileRefId="ID_999915" kind="LinkerScript" />
28 </LinkerOptionSet>
```
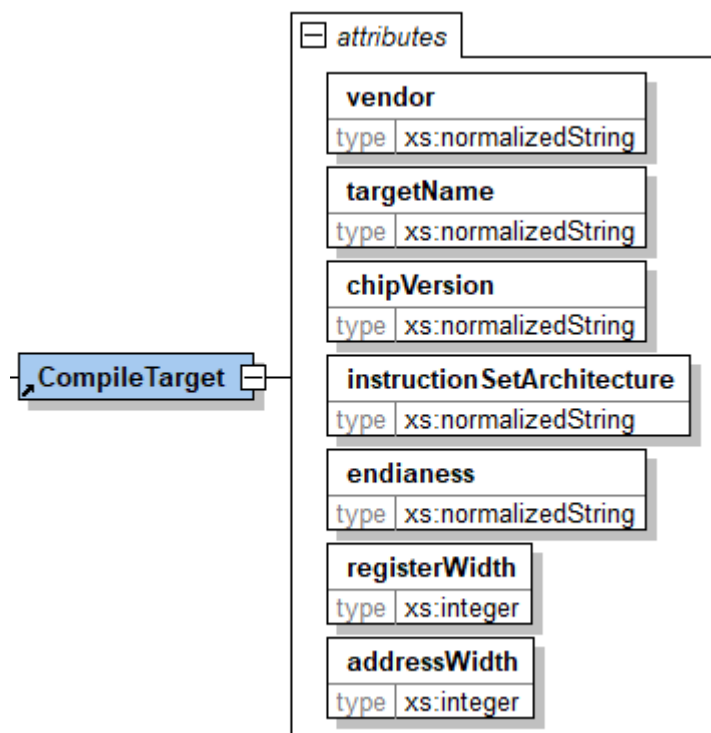
Target

In order to decide whether a target ECU is (technically) suitable for a particular binary with respect to target optimization and assumptions done during Production Code generation regarding hardware, the manifest has to specify the following items:

To define the target ECU the binary representation is compiled for, this section defines the following attributes:

Name	Description
vendor	The manufacturer of the the target platform/processor.
targetName	The name of the architecture.
chipVersion	The exact version of processor used in the architecture.
instructionSetArchitecture	A unique identifier for the instruction set used by the chip.
endianess	Describes whether the target uses Big-Endian or Little-Endian byte order.
registerWidth	Declares the bit width of the registers of the chip.
addressWidth	Declares the bit width of a memory address in the target.

```
1  <CompileTarget id="ID_100001" vendor="Infineon" targetName="TriCore" chipVersion="TC277 C-Step"
   instructionSetArchitecture="TC1.6E" endianess="LITTLE" registerWidth="32" addressWidth="32"/>
```

6.1.3. Modules

The `Modules` section lists and describes all relevant binaries contained in the Binary Model Representation. Furthermore, it lists all source code references to the Production Code container that are provided with the binary files.

The `Modules` section consist of a list of one or more `BinaryModule` items.
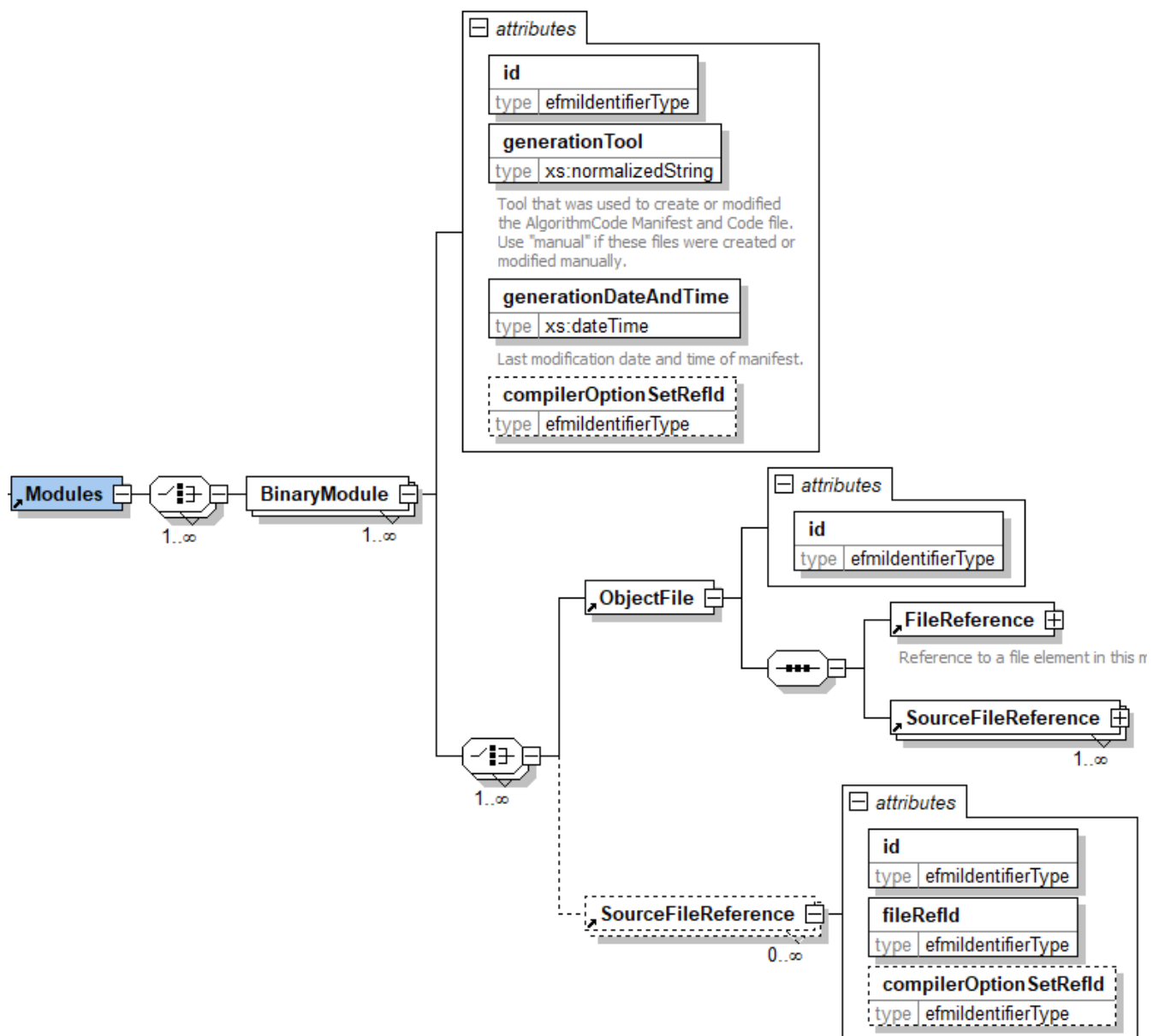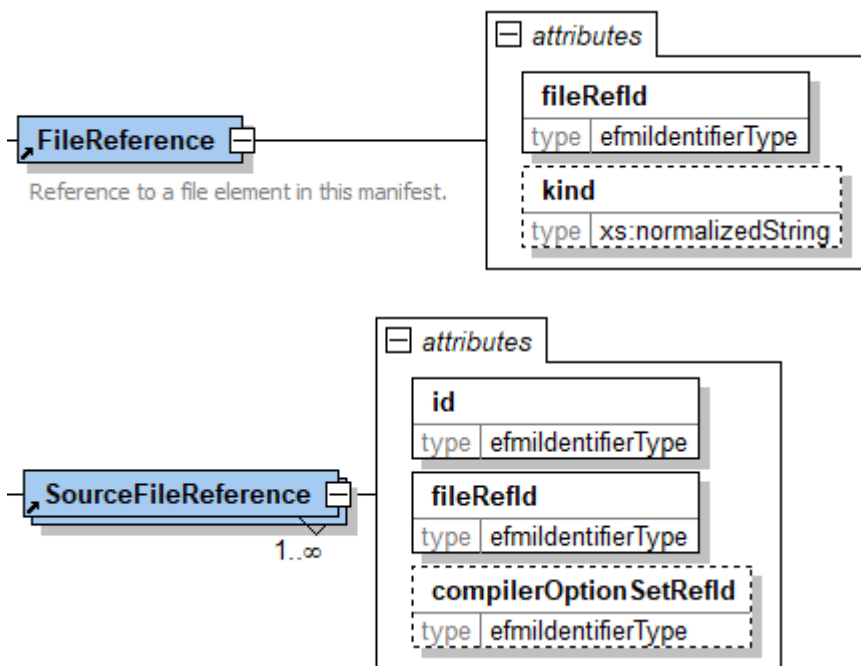
A `BinaryModule` describes a binary object in the Binary Code Model Representation. It has the following attributes:

Name	Description
id	A unique identifier for further referencing.
creator	The creating tool or person.
creationDate	The date, the particular binary moduel has been created.
compilerOpti onSetRefId	A reference to the `CompilerOptionSet` used for generation of the object file.

A `BinaryModule` contains one `ObjectFile` element and zero or more `SourceFileReference`:

Name	Description
ObjectFile	The actual binary object in the container. There can be only one object file per Binary module.
SourceFileRe ference	Each element refers to a code file in production Code manifest.

NOTE	`SourceFileReference` elements refer to possibly required `CodeFile` elements from the Production Code Model. Those files are not part of the object file but might be necessary for further processing steps, e.g., a PiL simulation of th object file.

The `SourceFileReference` element has the following attributes:

Name	Description
id	A unique identifier for further referencing.
fileRefId	Reference to the code Files in the Production Code manifest via a `ForeignFile` reference in the manifest `Files` section.

Name	Description
CompilerOpti onSetId	If a `CompilerOptionSetId` is specified, it must be used for compiling this code artifact. Otherwise, the `DefaultCompilerOptions` must be used.

Each `ObjectFile` has the following attributes:

Name	Description
id	A unique identifier for further referencing.

Additionally, it consists of the following elements:

Name	Description
FileReferenc e	Reference to the actual binary object file. The `kind` of the `FileReference` is either *"RelocatableObjectFile"* or *"ExecutableObjectFile"*. This element is mandatory.
ForeignSourc eFileReferen ce	The `ForeignSourceFileReference` elements refer to `CodeFile` elements of the Production Code Model Representation which have been used to generate the binary object file. The presence of the actual source files in the Production code container is not required. The manifest information, however, needs to be available.

NOTE	The following example shows a snippet for a very simple model. It consists of one non-executable object file that have been generated from two ("Production Code") source files.

```xml
 1  <ForeignFile id="ID_999920">
 2    <ForeignReference foreignRefId="ID_9" manifestReferenceRefId="ID_0000001" />
 3  </ForeignFile>
 4  <ForeignFile id="ID_999921">
 5    <ForeignReference foreignRefId="ID_10" manifestReferenceRefId="ID_0000001" />
 6  </ForeignFile>
 7  <ForeignFile id="ID_999922">
 8    <ForeignReference foreignRefId="ID_5" manifestReferenceRefId="ID_0000001"/>
 9  </ForeignFile>
10  <ForeignFile id="ID_999923">
11    <ForeignReference foreignRefId="ID_1" manifestReferenceRefId="ID_0000001"  />
12  </ForeignFile>
13  <ForeignFile id="ID_999924">
14    <ForeignReference foreignRefId="ID_3" manifestReferenceRefId="ID_0000001"  />
15  </ForeignFile>
16  [...]
17  <Modules>
18    <BinaryModule id="ID_4" creator="JDoe" creationDate="2018-08-09">
19      <ObjectFile id="ID_10">
20        <FileReference fileRefId="ID_01" kind="RelocatableObjectFile" />
21        <SourceFileReference id="ID_02" fileRefId="ID_999920" />
22        <SourceFileReference id="ID_03" fileRefId="ID_999921" />
23      </ObjectFile>
24      <SourceFileReference id="ID_5"fileRefId="ID_999922" />
25      <SourceFileReference id="ID_1" compilerOptionSetRefId="ID_46" fileRefId="ID_999923" />
26      <SourceFileReference id="ID_3" compilerOptionSetRefId="ID_46" fileRefId="ID_999924" />
```
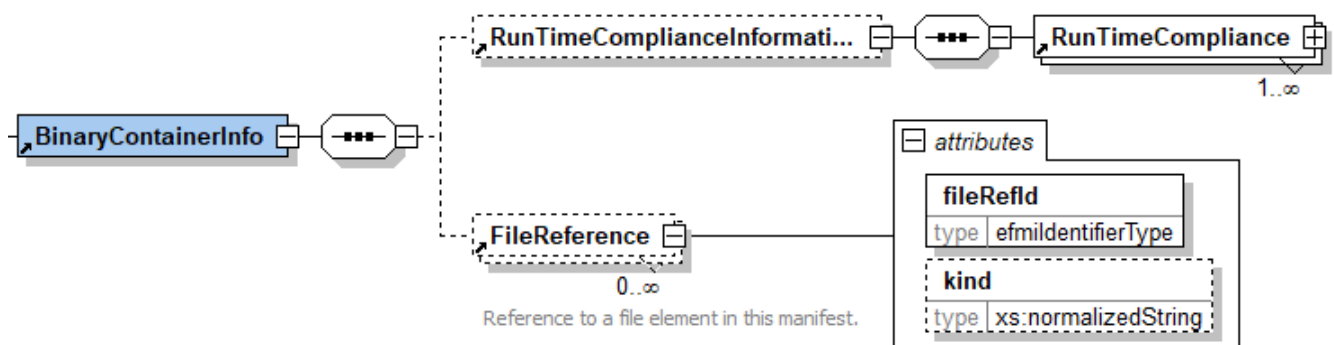
```
27    </BinaryModule>
28  </Modules>
```

6.1.4. Binary Container Info (optional)

The previously described elements of the manifest for the Binary Code Model Representation are mandatory. However, there is also information that might not be necessary to describe a binary but very helpful in the actual use cases for the Binary Code Model Representation such as integration or validation.

To store and provide this information, the manifest contains the BinaryContainerInfo section. A BinaryContainerInfo element might contain a description for each of the following topics

- mapping information (memory, registers, etc.)

- run time behavior

- calibration information

- measurement information

- information about the diagnosis interface

The BinaryContainerInfo element is defined as follows:

It contains the following elements:

Name	Description
RunTimeComplianceInformation	Information regarding run time behavior of the different functions provided by the Binary Code model representation.
FileReference	In addition to the run time information, it is also possible to provide reference to files that give further information regarding the above mentioned topics. The kind of the FileReference indicates which topic is tackled. Possible kinds are: *MapFile*, *CalibrationInformationFile*, *MeasurementInformationFile*, *DiagnosisInformationFile*, *ValidationAndVerificationFile*, *ComplianceInformationFile*, *LicenseFile*, *ConfigurationFile*.

Mapping Information

In order to provide the integration engineer with additional information about a binary file that has already has been linked, a map file can be specified in the `MapFileReference` element.

NOTE	The following example shows, how a map file can be provided using the combination of the `File` element declared for the Manifest and the actual `FileReference` with the `kind="MapFile"`.

```
1  <File id="ID_999913" path="/objects/" name="SpeedController.map" role="other"
   needsChecksum="true"
2         checksum="A43C0994FAD1247988C2AA8A90CCA2E241CF5687" />
3  [...]
4  <BinaryContainerInfo>
5    <FileReference fileRefId="ID_999913" kind="MapFile" />
6  </BinaryContainerInfo>
```

NOTE	The map file can be used to easily inspect information about the memory mapping and, memory usage. Furthermore general information about estimated stack size and the overall link process can be provided here.

Run Time Behavior

In order to integrate a function defined in an eFMI into a binary for the target ECU, it is required to have information about the run time behavior to decide whether there are enough resources available in order to coexist with additional functions or tasks running on the same ECU.
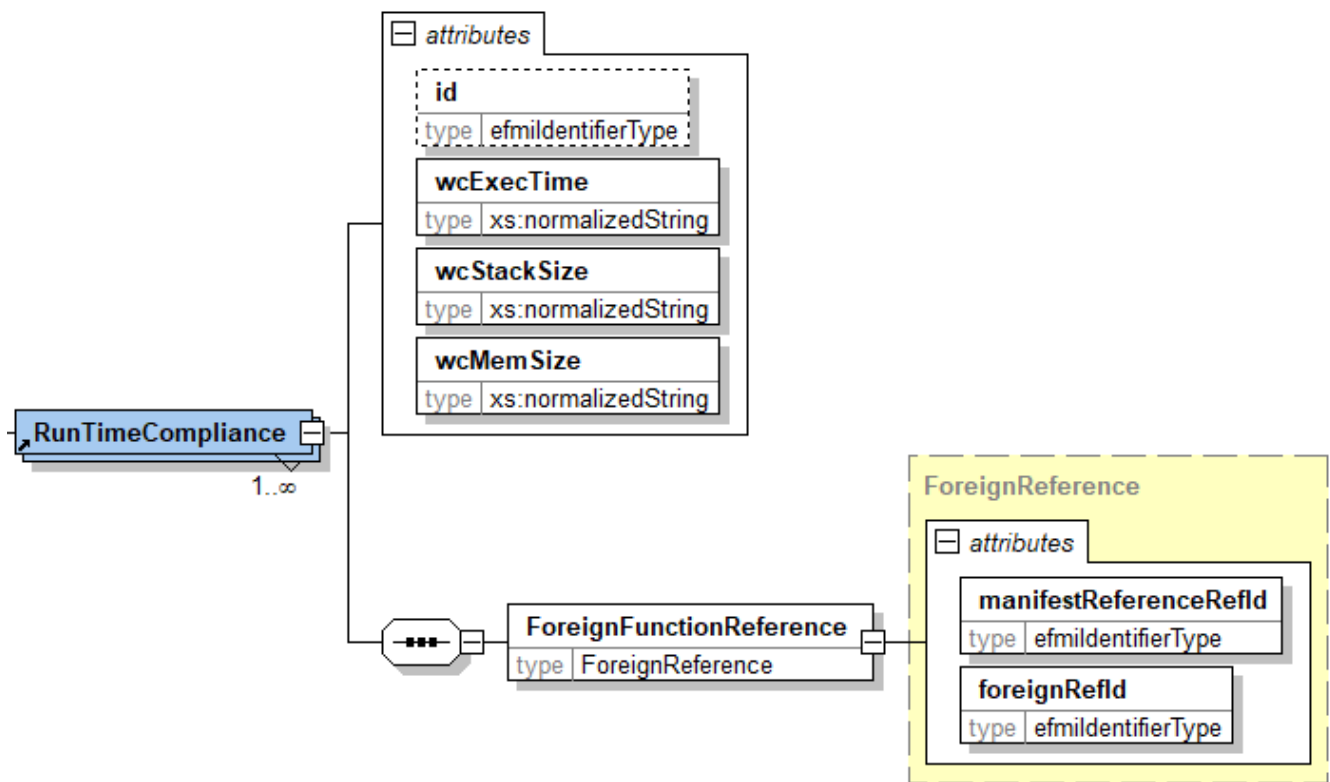
NOTE	This information might help the integration engineer to identify possible bottlenecks before he starts the actual integration.

Hence, the manifest can specify `RunTimeComplianceInformation` as additional, optional information.

If `RunTimeComplianceInformation` is provided, it can specify the run time behavior for one or more functions as follows:

It consists of one `ForeignFunctionReference` that refers to the function in the manifest of the Production Code model representation. The information about the run time behavior is described by the following attributes:

Name	Description
id	A unique identifier for further referencing.
wcExecTime	The maximum time consumed by the function in the worst case.
wcStackSize	The maximum stack size required by the function in the worst cas.
wcMemSize	The maximum memory consumed by the function in the worst case.

Note that valid units have to be used for each attribute by the author.

NOTE	The following example shows how the `RunTimeComplianceInformation` can be defined for some function.

```
1 <BinaryContainerInfo>
2   <RunTimeComplianceInformation>
3     <RunTimeCompliance id="ID_100301" wcExecTime="8.4ms" wcStackSize="70kb" wcMemSize="840kb">
4       <ForeignFunctionReference foreignRefId="ID_41" manifestReferenceRefId="ID_0000001" />
5     </RunTimeCompliance>
6   </RunTimeComplianceInformation>
7 </BinaryContainerInfo>
```

Calibration

In order to be able to calibrate the binary object provided by the Binary Code Model Representation with common, widely used calibration tools, the manifest might specify one or more files containing calibration information. Calibration information is given using `FileReference` elements with the `kind="CalibrationInformationFile"`.

NOTE	The following code snippet shows how a calibration file can be provided.

```
1 <File id="ID_999912" path="/" name="myFunction.a2l" role="other"
  checksum="0DC09613F414FFCE10865AF3AD3EC31D3ED61EA8" needsChecksum="true" />
2 [...]
3 <BinaryContainerInfo>
4   <FileReference fileRefId="ID_999912" kind="CalibrationInformationFile" />
5 </BinaryContainerInfo>
```

NOTE	An incomplete and optional A2L file provides the symbols used for calibration purposes. When the integrator performs the final linking, the memory addresses of all A2L files of the used software functions are updated. The resulting A2L files can be used by calibration tools to dynamically change parameters for example.

Measurement

In order to measure internal values of the controller software during the testing and validation phase, the manifest might specify one or more file containing measurement information. Measurement information is given using `FileReference` elements with the `kind="MeasurmentInformationFile"`.

NOTE	The following code snippet shows how a measurement information file can be provided. Note that in this example, in case of an *A2L*-File, the same file might be used for calibration and measurement.

```
1 <File id="ID_999912" path="/" name="myFunction.a2l" role="other"
  checksum="0DC09613F414FFCE10865AF3AD3EC31D3ED61EA8" needsChecksum="true" />
2 [...]
3 <BinaryContainerInfo>
4   <FileReference fileRefId="ID_999912" kind="MeasurmentInformationFile" />
5 </BinaryContainerInfo>
```

Diagnosis

ECU software often provides some subroutines for diagnosis that is used for testing and maintenance. Hence, the manifest of a Binary Model representation can contain one or more files that provide information for diagnosis tools. Diagnosis information is given using `FileReference` elements with the `kind="DiagnosisInformationFile"`.

NOTE	The following code snippet shows how a diagnosis information file can be provided.

```
1 <File id="ID_999914" path="/" name="myFunction.cdd" role="other"
    checksum="E7A58CD816076EE26DE1D6BF2F13630000675FB2" needsChecksum="true" />
2 [...]
3 <BinaryContainerInfo>
4   <FileReference fileRefId="ID_999914" kind="DiagnosisInformationFile" />
5 </BinaryContainerInfo>
```

Compliance

Since the main intention of the Binary Code container is the protection of intellectual property, the source code usually cannot be checked according to compliance to relevant standards. However, since this information might be of interest for the integrating company, an eFMI binary container shall have an optional section to define one or more files describing the components compliance. Diagnosis information is provided using `FileReference` elements with the `kind="ComplianceInformationFile"`.

NOTE   The following code snippet shows how a compliance information file can be provided.

```
1 <File id="ID_999910" path="/doc/" name="MISRA.doc" role="other"
    checksum="27D8D7BB69E1D7E98C7A278C5A48199CE7B65399" needsChecksum="true" />
2 [...]
3 <BinaryContainerInfo>
4   <FileReference fileRefId="ID_999910" kind="ComplianceInformationFile" />
5 </BinaryContainerInfo>
```

NOTE   A `FileReference` can also point to a `ForeignFile` element and, hence, to an arbitrary file in the eFMU container. This means it can also point to a compliance information file from Production Code container.

NOTE   Note that the eFMI standard does not define how the integrity of the compliance information can be ensured. It is up to the software provider and the integrating company to ensure the validity and integrity of this compliance information.

License Information

In case that any third party licenses have to be shipped with the binary or to provide license information is provided using `FileReference` elements with the `kind="LicenseFile"`.

NOTE   The following code snippet shows how a licenese file can be provided.

```
1 <File id="ID_999911" path="/license/" name="BSD.TXT" role="other"
    checksum="A7549D084CFD2F9C6DEFA940B9BD5DA402B8341D" needsChecksum="true" />
2 [...]
3 <BinaryContainerInfo>
4   <FileReference fileRefId="ID_999910" kind="LicenseFile" />
5 </BinaryContainerInfo>
```

Validation & Verification

For Verification and Validation, additional files can be provide using one or more `FileReference` elements with the `kind="ValidationAndVerificationFile"`.

NOTE	The following code snippet shows how some simulation results (e.g., ASAM MDF format) from a use case for back to back testing as well as some description of equivalence classes (e.g., properitary XML format) can be specified for th container.

```
1 <File id="ID_999920" path="/v_n_v/" name="scenario1.mdf" role="other"
    checksum="DB1A8489D88604A5C896BAB2B35631314B257036" needsChecksum="true" />
2 <File id="ID_999921" path="/v_n_v/" name="equivalenceclasses.xml" role="other"
    checksum="F61E2D36002DD140653334E4871DEBE6EE3B721A" needsChecksum="true" />
3 [...]
4 <BinaryContainerInfo>
5   <FileReference fileRefId="ID_999910" kind="ValidationAndVerificationFile" />
6 </BinaryContainerInfo>
```

Configuration of Runtime

Certain binary files require additional information on runtime. The Binary Code container provides the possibility to link such information via `FileReference` elements with the `kind="ConfigurationFile"`.

NOTE	The following code snippet shows how a SOME/IP stack configuration for Adaptive AUTOSAR application is referenced.

```
1 <File id="ID_999910" path="/adaptive/" name="someip.json" role="other"
    checksum="DB1A8489D88604A5C896BAB2B35631314B257036" needsChecksum="true" />
2 [...]
3 <BinaryContainerInfo>
4   <FileReference fileRefId="ID_999910" kind="ConfigurationFile" />
5 </BinaryContainerInfo>
```

6.2. Binary Format

The Binary Code Model Representation contains object files and libraries in binary format.

For deployment on a target architecture the object file or library must be provided as a binary file ELF format [ELFLinux].

NOTE	Hence, an ELF file should be be target specific (e.g., for a specific ECU) and, optionally, may be executable. Executable ELF files will be used in PiL Simulation and can contain dedicated frame code. PiL-simulation tools may also create their own harness for PiL simulation. Non-executable ELF files (relocatable ELF) can be used for the integration on the embedded target.

For Windows-based co-simulation a Binary Code Model Representation might also contain Windows-

compatible object files or dynamic link libraries [DLLWin].

NOTE	For the (co-)simulation use case the binary artifacts support multiple use cases. On the one hand, it may be a DLL, shared library or object file for general purpose code for a general purpose platform (e.g., Windows or Linux) that can be used in a Software-in-the-Loop simulation.

Additionally, the Binary Code Model Representation can refer to the following Production Code Model Represention items:

- Simulation Code that might be necessary/used for a standalone SiL or PiL simulation of the eFMU.

- Tool specific code that might be required to use simulation features of a particular tool.

NOTE	An example for the tool specific code might be a TargetLink S-Function frame used for a SiL Simulation or an TargetLink TSM-Frame used for PiL simulation. Another example migth be a minimal stub for debugging purposes on the target architecture.

Beside the actual binary format the Binary Code Model Representation might contain also files including information for calibration, measurement and diagnosis purposes.

NOTE	An example format for the description of calibration, measurement and diagnosis is the ASAM A2L format. This might be an incomplete A2L since the absolute memory addresses will be updated after the final link process is completed.

An eFMI Binary Model Represention might make use of service functions which do not necessarily have to be contained in the binary files. Especially for the use case of ECU integration these service functions might be provided by the ECU environment.

[] **Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification**. http://refspecs.linuxfoundation.org/elf/elf.pdf, last visited 2019-03-28.

[] **Dynamic-Link Libraries**. https://docs.microsoft.com/en-us/windows/desktop/Dlls/dynamic-link-libraries, last visited 2019-03-29.

Chapter 7. Acronyms

Name	Description
AA	Adaptive AUTOSAR Application
AlgC	Algorithm Code
AlgCL	Algorithm Code Language
ARXML	Classic AUTOSAR interface description file
AST	Abstract Syntax Tree
Bin Code	Binary Code
DAE	Differential Algebraic Equation system
ECU	Embedded Control Unit
eFMI	FMI for embedded systems
eFMU	FMU for embedded systems
ELF	Executable and Linking Format
EqC	Equation Code
EqCL	Equation Code Language
FFT	Fast Fourier Transform
FMI	Functional Mock-Up interface
FMI-CS	FMI for Co-Simulation
FMU	Functional Mock-Up unit
GPL	GNU General Public License
LPV	Linear Parameter-Varying (control / controller)
LTI	Linear Time-Invariant
LTV	Linear Time-Varying
ML	Machine Learning
MPC	Model Predictive Control
NMPC	Nonlinear Model Predictive Control
NN	Neural Network
ODE	Ordinary Differential Equations
PID	Proportional-Integral-Derivative (control / controller)
PiL	Processor-in-the-Loop
Prod Code	Production Code
SiL	Software-in-the-Loop
SOA	Service-oriented Architecture

Name	Description
SW	Software
SWC	Classic AUTOSAR Software Component
V&V	Validation & Verification

Chapter 8. Glossary

- Calibration Parameter - Value equals the start value and can be changed anytime during evaluation of the system by an external source [Req_4.1.09, Req_5.1.13].

- Calibration Variables - Constant for all execution steps, but changeable by eeprom-update [Req_6.2.05].

- Code - Formal specification of the model behavior.

    ◦ Production Code - Code intended for the execution on an embedded system.

    ◦ Target Specific Code - Production Code with specific instructions for a certain target.

- ECU software content - Pre-existing software into which the Production Code has to be integrated.

- eFMU - Container of model representations and other artefacts according to the eFMI standard.

- Manifest - Meta information in an extendable form describing an associated artefact.

    ◦ eFMU Manifest - Manifest describing the available model representations of the eFMU container and how to get access to them, plus other general meta information.

    ◦ Code Manifest - Manifest describing the model interface of the associated code and providing additional meta information on how to access and utilize the code.

- Model Representation - Compound of Code + Code Manifest representing the model in one particular standardized form.

- Parameter - Value equals the start value and can be changed only before initialization of the system.

- State Machine - A (finite) state machine is used to model a system fluctuating between a fixed number of states. Transitions rules between one state to another are defined through entry and exit actions.

- State-Space Representation - A mathematical model describing the dynamics of a system with a set of first order differential equations. Inputs, outputs and internal state variables are related by A, B, C, D matrices.

- System constants - Values that are constant for a specific configuration of a software system under test (a specific variant of software and hardware components), but might be changed if the component is used for a slightly different configuration (e.g. number of battery cells available).

- Target - The intended productive execution environment of the software function that is encapsulated in the eFMU. The eFMU target is characterized by the controller hardware (processor, …) and software (compiler, runtime environment, software architecture).

References

- [BOA11] Blochwitz T., Otter M., Arnold M., Bausch C., Clauß C., Elmqvist H., Junghanns A., Mauss J., Monteiro M., Neidhold T., Neumerkel D., Olsson H., Peetz J.-V., Wolf S. (2011): **The Functional Mockup Interface for Tool independent Exchange of Simulation Models**. 8th International Modelica Conference, Dresden 2011. http://www.ep.liu.se/ecp/063/013/ecp11063013.pdf

- [BOA12] Blochwitz T., Otter M., Akesson J., Arnold M., Clauß C., Elmqvist H., Friedrich M., Junghanns A., Mauss J,, Neumerkel D., Olsson H., Viel A. (2012): **Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models**. 9th International Modelica Conference, Munich, 2012. http://www.ep.liu.se/ecp/076/017/ecp12076017.pdf

- [CLangWiki] **The C Programming Language**. https://en.wikipedia.org/wiki/C_(programming_language), last visited 2019-02-06.

- [CPPLangWiki] **C++** Programming Language. https://en.wikipedia.org/wiki/C%2B%2B, last visited 2019-02-06.

- [DLLWin] **Dynamic-Link Libraries**. https://docs.microsoft.com/en-us/windows/desktop/Dlls/dynamic-link-libraries, last visited 2019-03-29.

- [ELFLinux] **Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification**. http://refspecs.linuxfoundation.org/elf/elf.pdf, last visited 2019-03-28.

- [KR78] Kernighan Brian W., Ritchie Dennis M. (1978): **The C Programming Language** (1st ed.), Englewood Cliffs, NJ: Prentice Hall. ISBN 0-13-110163-3.

- [MISRA12] MISRA C:2012: **Guidelines for the use of the C language in critical systems**. ISBN 978-1-906400-10-1, MIRA Limited, Nuneaton, March 2013

- [MISRA08] MISRA C++:2008: **Guidelines for the use of the C++ language in critical systems**. ISBN 978-906400-04-0, MIRA Limited, Nuneaton, March 2013

- [MISRA04] MISRA AC AGC: **Guidelines for the application of MISRA-C:2004 in the context of automatic code generation**. ISBN ISBN 978-906400-02-6, MIRA Limited, Nuneaton, March 2004

- [SHA1Wiki] **Secure Hash Algorithm**. https://en.wikipedia.org/wiki/Secure_Hash_Algorithms, last visited 2019-02-08.

- [Str13] Stroustrup Bjarne (1997), **The C++ Programming Language** (Forth ed.), Addison-Wesley, ISBN 0-32-156384-0.

Appendix A: Legal information

Copyright

The eFMI Standard consists of (1) the actual specification text and (2) accompanying software artifacts the specification text refers to and leverages on for defining how valid eFMUs must look like and what their meaning is (the XML Schema Definitions (XSD) for the manifests of eFMUs); both are copyrighted by the Modelica Association [https://modelica.org/]:

© 2021-2024, Modelica Association and contributors.

Licensing

The specification text of the eFMI Standard is released under Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) [https://creativecommons.org/licenses/by-sa/4.0/] license:
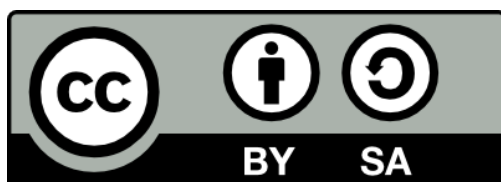
This work is licensed under a CC BY-SA 4.0 license.

The XML Schema Definitions of the eFMI Standard are released under 3-Clause BSD License [https://opensource.org/licenses/BSD-3-Clause]:

Registered trademarks

Modelica® is a registered trademark of the Modelica Association.

eFMI® is a registered trademark of the Modelica Association.

FMI® is a registered trademark of the Modelica Association.

SSP® is a registered trademark of the Modelica Association.

DCP® is a registered trademark of the Modelica Association.

Third party marks and brands are the property of their respective holders.

Colophon

This document has been typeset in AsciiDoc® [https://asciidoc.org/] using Asciidoctor [https://asciidoctor.org/] and Asciidoctor PDF [https://docs.asciidoctor.org/pdf-converter/latest/] to generate HyperText Markup Language [https://html.spec.whatwg.org/] and Portable Document Format (ISO 32000-2:2020) distributions. AsciiDoc® is licensed under Eclipse Public License 2.0 [https://www.eclipse.org/legal/epl-2.0/], and Asciidoctor and Asciidoctor PDF are licensed under MIT License [https://opensource.org/license/mit].

The Portable Document Format distribution of this document uses the Sans and Mono font of the Kurinto Font Folio [https://www.kurinto.com/] licensed under the SIL Open Font License (OFL) [https://openfontlicense.org/] version 1.1.

Disclaimer

This release is a candidate-draft for the eFMI Standard 1.0.0 (the first stable version of the eFMI Standard); its version is *eFMI Standard 1.0.0 Beta 1*. It succeeds the *eFMI Standard 1.0.0 Alpha 4* candidate-draft. Candidate-drafts provide no guarantees or implications (including neither extent, features, structure nor wording) for future eFMI Standard releases including the final stable release they are a candidate of.

Appendix B: Contributors

The eFMI Standard 1.0.0 development has been mostly done within the ITEA 3 project EMPHYSIS [https://itea3.org/project/emphysis.html] — which run from September 2017 until February 2021, lead by Oliver Lenord (Robert Bosch GmbH) and was initiated and organized by Oliver Lenord, Christian Bertsch (Robert Bosch GmbH), Pacôme Magnin (Siemens NV) and Martin Otter (German Aerospace Center DLR) — and finished from 2021-2024 in the succeeding Modelica Association Project eFMI [https://efmi-standard.org] whose project leader was Christoff Bürger (Dassault Systèmes).

Specification

The essential design and actual writing of the specification has been done by the following core developers:

Algorithm Code

- Main author: Christoff Bürger (Dassault Systèmes)
- Other contributors:
  - Martin Otter (German Aerospace Center DLR)
  - Andreas Pfeiffer (German Aerospace Center DLR)

Behavorial Model

- Main author: Christoff Bürger (Dassault Systèmes)
- Other contributors:
  - Andreas Pfeiffer (German Aerospace Center DLR)
  - Robert Reicherdt (PikeTec GmbH, part of Synopsys, Inc.)

Production Code

- Main author: Jörg Niere (dSPACE GmbH)
- Other contributors:
  - Kai Werther (ETAS GmbH)
  - Michael Hussmann (dSPACE GmbH)
  - Robert Reicherdt (PikeTec GmbH, part of Synopsys, Inc.)

Binary Code

- Main author: David Brenken (e:fs TechHub GmbH, former name: Elektronische Fahrwerksysteme GmbH)
- Other contributors:
  - Robert Reicherdt (PikeTec GmbH, part of Synopsys, Inc.)
  - Pierre Le Bihan (Dassault Systèmes)
  - Jörg Niere (dSPACE GmbH)

Introduction

- Main author: Martin Otter (German Aerospace Center DLR)

- Other contributors:
    - Christoff Bürger (Dassault Systèmes)

Appendix

- Main author: Christoff Bürger (Dassault Systèmes)

- Other contributors:

    - Martin Otter (German Aerospace Center DLR)

Tool support

The eFMI specification has been assessed by implementing eFMI support in various tools covering the whole eFMI workflow. More than a hundred test models and variants of the benchmark test cases (see Section Benchmark test cases) have been used to evaluate the toolchain, validating tool interoperability and correctness.

The developed and benchmarked tools are (in alphabetic order):

AUTOSAR Builder (Dassault Systèmes)

- Generation of AUTOSAR Platform components (AUTOSAR Adaptive and Classic) from eFMI Production Code containers. Generated AUTOSAR components can be stored as eFMI Production or Binary Code containers referencing their adapted source containers.

- Developers: Fabien Aillerie

Astrée (AbsInt Angewandte Informatik GmbH)

- Verification of eFMI Production Code containers for safety and code quality according to common embedded coding standards like MISRA C:2012.

- Developers: Reinhold Heckmann

Behavioral Model Scripts (German Aerospace Center DLR)

- Generation of eFMI Behavioral Model containers from annotated Modelica models.

- Developers: Andreas Pfeiffer

CSD (Siemens NV)

- Test of eFMI Production Code containers using eFMI Behavioral Model containers and system integration of production code into existing embedded code.

- Developers: Jishnu Jayaram

Dymola (Dassault Systèmes)

- Generation of eFMI Algorithm Code containers from Modelica models and generation of respective eFMI Behavioral Model containers from Modelica experiments using the model. Software-in-the-loop (SiL) test of Software Production Engineering generated eFMI Production Code containers using eFMI Behavioral Model containers.

- Developers: Christoff Bürger

SCODE CONGRA (ETAS GmbH)

- Generation of eFMI Production Code containers from eFMI Algorithm Code containers. Software-in-the-loop (SiL) test of eFMI Production Code containers using eFMI Behavioral

Model containers.

- Developers: Kai Werther

Simcenter Amesim (Siemens Digital Industries Software)

- Generation of neural network approximation as eFMI Algorithm Code container from Amesim models.

- Developers: Jérôme André

SimulationX (ESI Group)

- Generation of eFMI Algorithm Code containers from Modelica models.

- Developers: Gerd Kurzbach

Software Production Engineering (Dassault Systèmes)

- Generation of eFMI Production Code containers from eFMI Algorithm Code containers. Generation of eFMI Binary Code containers from eFMI Production Code containers.

- Developers: Samuel Devulder, Pierre Le Bihan, Fabien Aillerie, Laurent Le Goff

TargetLink (dSPACE GmbH)

- Generation of eFMI Production Code containers from eFMI Algorithm Code containers and system integration of production code into the MATLAB/Simulink® ecosystem and dSPACE hardware. Target system simulation of eFMI Production Code containers for software- and hardware-in-the-loop simulation.

- Developers: Michael Hussmann, Jörg Niere

TPT (PikeTec GmbH, part of Synopsys, Inc.)

- Test of eFMI Production Code containers using eFMI Behavioral Model containers. Support for various target platforms for hardware-in-the-loop testing.

- Developers: Robert Reicherdt

Benchmark test cases

The specification has been assessed with benchmark test cases provided in the Modelica library eFMI_TestCases [https://github.com/modelica/efmi-testcases] and as Simcenter Amesim models. The eFMI_TestCases library has been managed by Andreas Pfeiffer (German Aerospace Center DLR) and Christoff Bürger (Dassault Systèmes).

The individual benchmark test cases have been developed by (in alphabetic order):

German Aerospace Center DLR

- Jonathan Brembeck

- Ricardo de Castro

- Michael Fleps-Dezasse

- Martin Otter

- Andreas Pfeiffer

- Jakub Tobolar

Robert Bosch GmbH

- Siva Sankar Armugham

- Christian Bertsch

- Oliver Lenord

- Naresh Mandipalli

- Jonathan Neudorfer

- Christian Potthast

- Vishnupriya Veeraragavan

Siemens Digital Industries Software

- Jérôme André

Industrial demonstrators

The eFMI specification and the developed tools have been assessed with the following industrial demonstrators:

Performance assessment (Robert Bosch GmbH)

Comparing generated Production Code of nine benchmark test cases of the `eFMI_TestCases` library with manually developed code. This includes comparison of execution performance on the Bosch ECU MDG1.

- Tooling: Performance Test Environment

- Contributors:

  - Vishnupriya Veeraragavan

  - Oliver Lenord

Powertrain vibration reduction (Robert Bosch GmbH)

Generate a controller with a nonlinear inverse model on the Bosch ECU MDG1 to reduce vibrations in a powertrain.

- Tooling: Dymola, SCODE-CONGRA, TPT, Astrée and eFMI2AUTOSAR (Robert Bosch GmbH)

- Contributors:

  - Oliver Lenord

  - Kai Werther

  - Siva Sankar Armugham

Model-based diagnosis of thermo systems (Robert Bosch GmbH)

Generate diagnosis functions on the Bosch ECU MDG1.

- Tooling: OpenModelica, SCODE-CONGRA, ECU Test Environment

- Contributors:

  - Oliver Lenord

- Christian Potthast

Virtual sensor for hybrid drivetrain (Siemens NV, Siemens Digital Industries Software)

Generate virtual sensor by approximating a dynamic model by means of a neural network.

- Tooling: Simcenter Amesim and TargetLink
- Contributors:
  - Jérôme André (Siemens Digital Industries Software)
  - Alexander Van Bellinghen (Siemens NV)
  - Yuri Durodié (Siemens NV)
  - Jishnu Jayaram (Siemens NV)
  - Jorg Niere (dSPACE GmbH)

Semi-active damping controller and observer (German Aerospace Center DLR)

Generate a controller (with a nonlinear inverse model) and a prediction model (nonlinear extended Kalman Filter or nonlinear unscented Kalman Filter) on a pre-development ECU from e:fs TechHub GmbH (former name: Elektronische Fahrwerksysteme GmbH) and on an ECU of KW automotive. The implementation with the KW automotive ECU has been tested in real driving tests.

- Tooling: Dymola and TargetLink
- Contributors:
  - Florian Bitter (e:fs TechHub GmbH, former name: Elektronische Fahrwerksysteme GmbH)
  - Jonathan Brembeck (German Aerospace Center DLR)
  - Daniel Baumgartner (German Aerospace Center DLR)
  - Christoff Bürger (Dassault Systèmes)
  - David Brenken (e:fs TechHub GmbH, former name: Elektronische Fahrwerksysteme GmbH)
  - Dario Celan (e:fs TechHub GmbH, former name: Elektronische Fahrwerksysteme GmbH)
  - Georg Hofstetter (e:fs TechHub GmbH, former name: Elektronische Fahrwerksysteme GmbH)
  - Michael Hussmann (dSPACE GmbH)
  - Konrad Krauter (e:fs TechHub GmbH, former name: Elektronische Fahrwerksysteme GmbH)
  - Severin Kirpal (e:fs TechHub GmbH, former name: Elektronische Fahrwerksysteme GmbH)
  - Jorg Niere (dSPACE GmbH)
  - Andreas Pfeiffer (German Aerospace Center DLR)
  - Raik Ritter (e:fs TechHub GmbH, former name: Elektronische Fahrwerksysteme GmbH)
  - Julian Ruggaber (German Aerospace Center DLR)
  - Christina Schreppel (German Aerospace Center DLR)
  - Jakub Tobolar (German Aerospace Center DLR)

- Johannes Ultsch (German Aerospace Center DLR)

  - Christoph Winter (German Aerospace Center DLR)

Dual-clutch use case (Mercedes-Benz AG)

Standardized, parameterized, reusable module for a simplified dual clutch transmission model with state events. The model extensively uses typically stiff components of the Modelica Standard Library (modelica.org) like clutches with friction and non-linear springs, resulting in a stiff, mixed eqution system with discontinous states due to gear shifts. The objective is to demonstrate the portability of the generated module to hardware-in-the-loop (HiL) systems and to a pre-development transmission controller unit.

- Tooling:

  - Model development and eFMU generation: Dymola and TargetLink

  - Software-in-the-loop tests: Dymola, Software Production Engineering

  - Hardware-in-the-loop tests: TargetLink, ConfigurationDesk (dSPACE GmbH) and PROVEtech (Akka Technologies)

- Contributors:

  - Zdenek Husar (Mercedes-Benz AG)

  - Jan Röper (Mercedes-Benz AG)

  - Emmanuel Chrisofakis (Mercedes-Benz AG)

  - Klaus Riedl (Mercedes-Benz AG)

  - Christoff Bürger (Dassault Systèmes)

  - Hans Olsson (Dassault Systèmes)

Transmission model as virtual sensor (Volvo Cars)

Virtual sensor for electric machine control based on a Modelica transmission model. The virtual sensor provides vehicle state estimation used to mitigate, e.g., backlash in the electric driveline, and thereby increase the overall performance of the whole electric driveline.

- Tooling: Dymola and TargetLink

- Contributors:

  - Sarah Bellis (Volvo Cars)

  - Martin Johnsson (Volvo Cars)

  - Jart Hageman (Volvo Cars)

  - Sabina Linderoth (Volvo Cars)

  - Edvin Eriksson Johannsson (Volvo Cars)

  - David Kastö (Volvo Cars)

  - Aditya Naronikar (Volvo Cars)

  - Ottilia Wahlgren (Volvo Cars)

  - Emma Kroon (Volvo Cars)

- Johannes Emilsson (Volvo Cars)

- Joachim Härsjö (Volvo Cars)

- Per Jacobsson (Volvo Cars)

- Johan Bergeld (Volvo Cars)

- Christoff Bürger (Dassault Systèmes)

AEBS: Advanced Emergency Braking System (Dassault Systèmes)

Advanced emergency braking controller derived from industrial Simulink (MathWorks) model with enabled subsystems and signal locks. For correct handling of the side-effects of enabled subsystems Modelica state machines are used; the signal locks are modeled using `previous` of Modelica synchronous. The final objective is the generation and validation of an AUTOSAR Adaptive Platform component starting from the Modelica model via a seamless tool chain based on eFMI.

- Tooling:

    - Model development and Algorithm Code generation: Dymola

    - Production and Binary Code generation: Software Production Engineering

    - AUTOSAR Adaptive Platform component generation: AUTOSAR Builder

- Contributors:

    - Christoff Bürger (Dassault Systèmes)

    - Samuel Devulder (Dassault Systèmes)

    - Fabien Aillerie (Dassault Systèmes)

pNMPC controller for semi-active suspension (GIPSA-lab)

Model-based controller for semi-active suspension regulation with hardware-in-the-loop (HiL) test via the INOVE vehicle suspension test rig. The controller is a parameterized nonlinear model predictive controller (pNMPC) from GIPSA-lab using a neural network model to predict the future behavior of the car like the response of chassis and wheel to a given road profile and suspension parameter. The suspension control is realized by means of this simulated prediction. A Simcenter Amesim physics model of the whole car including suspension, chassis and wheels is used to derive and train the neural network model, for which in turn an implementation as eFMI GALEC code is generated (all within Simcenter Amesim). Respective eFMI production code is generated using TargetLink. The final solution is deployed on a dSPACE MicroAutoBox II ECU, based on GIPSA-lab's pNMPC module and a S-function block wrapping the production code.

- Tooling: Simcenter Amesim and TargetLink

- Contributors:

    - Olivier Sename (Gipsa Lab)

    - Rattena Tang (Gipsa Lab)

    - Suzanne De Conti (Gipsa Lab)

    - Karthik Murali Madhavan Rathai (Gipsa Lab)

    - Thanh-Phong Pham (Gipsa Lab)

- Manh-Hung Do (Gipsa Lab)

- Marc Alirand (Siemens Digital Industries Software)

- Jérôme André (Siemens Digital Industries Software)

- Joerg Niere (dSPACE GmbH)

Appendix C: Reserved Built-in Functions

This section lists already designed built-in functions that are not yet part of the efmi standard but might be added to it in the future. Therefore, the names and functionality of these functions are reserved:

Overview of the reserved built-in functions

Function-Name	Description
Round Real r to an Integer	
roundTowardsZero(r)	Round towards zero (also known as truncation).
roundAwayZero(r)	Round towards infinity.
roundHalfDown(r)	Round half towards negative infinity.
roundHalfUp(r)	Round half towards positive infinity.
roundHalfTowardsZero(r)	Round half towards zero (also knowns as: round half aways from infinity).
roundHalfAwayZero(r)	Round half away zero (also known as: round half towards infinity)
roundHalfToOdd(r)	Round half towards odd number.
Division of Integer variables i1, i2 with rounding to an integer	
divisionDown(i1,i2)	integer(roundDown(i1/i2)).
divisionUp(i1,i2)	integer(roundUp(i1/i2)).
divisionAwayZero(i1,i2)	integer(roundAwayZero(i1/i2)).
divisionHalfDown(i1,i2)	integer(roundHalfDown(i1/i2)).
divisionHalfUp(i1,i2)	integer(roundHalfUp(i1/i2)).
divisionHalfTowardsZero(i1,i2)	integer(roundHalfTowardsZero(i1/i2)).
divisionHalfAwayZero(i1,i2)	integer(roundHalfAwayZero(i1/i2)).
divisionHalfToEven(i1,i2)	integer(roundHalfToEven(i1/i2)).
divisionHalfToOdd(i1,i2)	integer(roundHalfToOdd(i1/i2)).
divisionEuclidean(i1,i2)	Euclidean division of two integers.
Integer remainder of division of Integer variables i1, i2	
remainderDown(i1,i2)	Integer remainder of roundDown(i1/i2).

Function-Name	Description
`remainderUp(i1,i2)`	Integer remainder of roundUp(i1/i2).
`remainderAwayZero(i1, i2)`	Integer remainder of roundAwayZero(i1/i2).
`remainderHalfDown(i1, i2)`	Integer remainder of roundHalfDown(i1/i2).
`remainderHalfUp(i1,i2 )`	Integer remainder of roundHalfUp(i1/i2).
`remainderHalfTowardsZ ero(i1,i2)`	Integer remainder of roundHalfTowardsZero(i1/i2).
`remainderHalfAwayZero (i1,i2)`	Integer remainder of roundHalfAwayZero(i1/i2).
`remainderHalfToEven(i 1,i2)`	Integer remainder of roundHalfToEven(i1/i2).
`remainderHalfToOdd(i1 ,i2)`	Integer remainder of roundHalfToOdd(i1/i2).
`remainderEuclidean(i1 ,i2)`	Integer remainder of Euclidean division.
Remainder of division of Real variables `r1, r2`	
`realRemainderDown(r1, r2)`	Real remainder of roundDown(r1/r2).
`realRemainderUp(r1,r2 )`	Real remainder of roundUp(r1/r2).
`realRemainderAwayZero (r1,r2)`	Real remainder of roundAwayZero(r1/r2).
`realRemainderHalfDown (r1,r2)`	Real remainder of roundHalfDown(r1/r2).
`realRemainderHalfUp(r 1,r2)`	Real remainder of roundHalfUp(r1/r2).
`realRemainderHalfTowa rdsZero(r1,r2)`	Real remainder of roundHalfTowardsZero(r1/r2)
`realRemainderHalfAway Zero(r1,r2)`	Real remainder of roundHalfAwayZero(r1/r2)
`realRemainderHalfToEv en(r1,r2)`	Real remainder of roundHalfToEven(r1/r2)
`realRemainderHalfToOd d(r1,r2)`	Real remainder of roundHalfToOdd(r1/r2)

Definition of the reserved built-in functions

The following functions are appended to $C_{builtin1}$:

```
1  /*********************************************************************************
2      Direct rounding to an integer:
3  *********************************************************************************/
4
5  function roundTowardsZero
6      input  Real r;
7      output Real i;
8  algorithm /*
9      Also known as: truncation, round away from infinity.
10     i := (if r >= 0.0 then roundDown(r) else roundUp(r));
11 */ end roundTowardsZero;
12
13 function roundAwayZero
14     input  Real r;
15     output Real i;
16 algorithm /*
17     Also known as: round towards infinity.
18     i := (if r <= 0.0 then roundDown(r) else roundUp(r));
19 */ end roundAwayZero;
20
21 /*********************************************************************************
22     Rounding to the nearest integer (using a tie-breaking rule):
23 *********************************************************************************/
24
25 function roundHalfDown
26     input  Real r;
27     output Real i;
28 algorithm /*
29     Also known as: round half towards negative infinity.
30     i := roundUp(r - 0.5);
31 */ end roundHalfDown;
32
33 function roundHalfUp
34     input  Real r;
35     output Real i;
36 algorithm /*
37     Also known as: round half towards positive infinity.
38     i := roundDown(r + 0.5);
39 */ end roundHalfUp;
40
41 function roundHalfTowardsZero
42     input  Real r;
43     output Real i;
44 algorithm /*
45     Also known as: round half away from infinity.
46     i := roundAwayZero(r - sign(r) * 0.5);
47 */ end roundHalfTowardsZero;
48
49 function roundHalfAwayZero
50     input  Real r;
51     output Real i;
52 algorithm /*
```

```
53     Also known as: round half towards infinity.
54     i := roundTowardsZero(r + sign(r) * 0.5);
55 */ end roundHalfAwayZero;
56
57 function roundHalfToOdd
58     input  Real r;
59     output Real i;
60 algorithm /*
61     i := (if roundHalfDown(r) < roundHalfUp(r)
62           then (if integer(remainder(r + 0.5, 2.0)) == 0 then r - 0.5 else r + 0.5)
63           else roundHalfDown(r));
64 */ end roundHalfToOdd;
65
66 /*********************************** END OF LISTING ***********************************/
```

The following functions redefine $C_{builtin2}$, which defines builtin functions for *Integer* division. For every function named rounda of $C_{builtin1}$ with a an arbitrary sequence of characters, $C_{builtin2}$ contains the character sequence:

```
 1 /*********************************** BEGIN OF LISTING ***********************************/
 2
 3 function divisiona
 4     input Integer dividend;
 5     input Integer divisor;
 6     output Integer quotient;
 7 algorithm /*
 8     quotient := integer(rounda(real(dividend) / real(divisor)));
 9 */ end divisiona;
10
11 function remaindera
12     input Integer dividend;
13     input Integer divisor;
14     output Integer remainder;
15 algorithm /*
16     remainder := dividend - divisor * divisiona(dividend, divisor);
17 */ end remaindera;
18
19 /*********************************** END OF LISTING ***********************************/
```

Further, $C_{builtin2}$ contains the following character sequence:

```
 1 /*********************************** BEGIN OF LISTING ***********************************/
 2
 3 function divisionEuclidean
 4     input Integer dividend;
 5     input Integer divisor;
 6     output Integer quotient;
 7 algorithm /*
 8     quotient := integer((if divisor > 0
 9         then roundDown(real(dividend) / real(divisor))
10         else roundUp(real(dividend) / real(divisor))));
11 */ end divisionEuclidean;
12
13 function remainderEuclidean
```

```
14    input Integer dividend;
15    input Integer divisor;
16    output Integer remainder;
17 algorithm /*
18    remainder := dividend - divisor * divisionEuclidean(dividend, divisor);
19 */ end remainderEuclidean;
20
21 /******************************* END OF LISTING *********************************/
```

Above functions are in lexical order w.r.t. their names; they constitute $C_{builtin2}$ in its entirety.

The following functions redefine $C_{builtin3}$, which defines builtin functions for *Real* division, where the quotient is forced to be an integer according to a rounding strategy. For every function named roundα of $C_{builtin1}$ with α an arbitrary sequence of characters, $C_{builtin3}$ contains the character sequence:

```
 1 /************************************ BEGIN OF LISTING ************************************/
 2
 3 function realRemainderα
 4    input Real dividend;
 5    input Real divisor;
 6    output Real remainder;
 7 algorithm /*
 8    remainder := dividend - divisor * round_α(dividend / divisor);
 9 */ end realRemainderα;
10
11 /************************************ END OF LISTING *************************************/
```

Above functions are in lexical order w.r.t. their names; they constitute $C_{builtin3}$ in its entirety.