Functional Mock-Up Interface for embedded systems (eFMI)

Version 1.0.0-alpha.4 (Draft), February 22, 2021

Contents

Preamble

.1. CopyRight and License

.2. Release Notes

.2.1. Version 1.0.0-alpha.4

Disclaimer

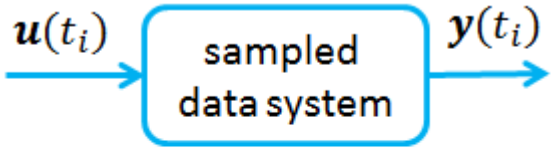
This alpha release is a draft version of the eFMI standard (= Functional Mock.Up Interface for embedded systems). It is planned to standardize a potentially improved version by the Modelica Association.

.3. 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:



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

- 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.

.4. 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 .5 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.

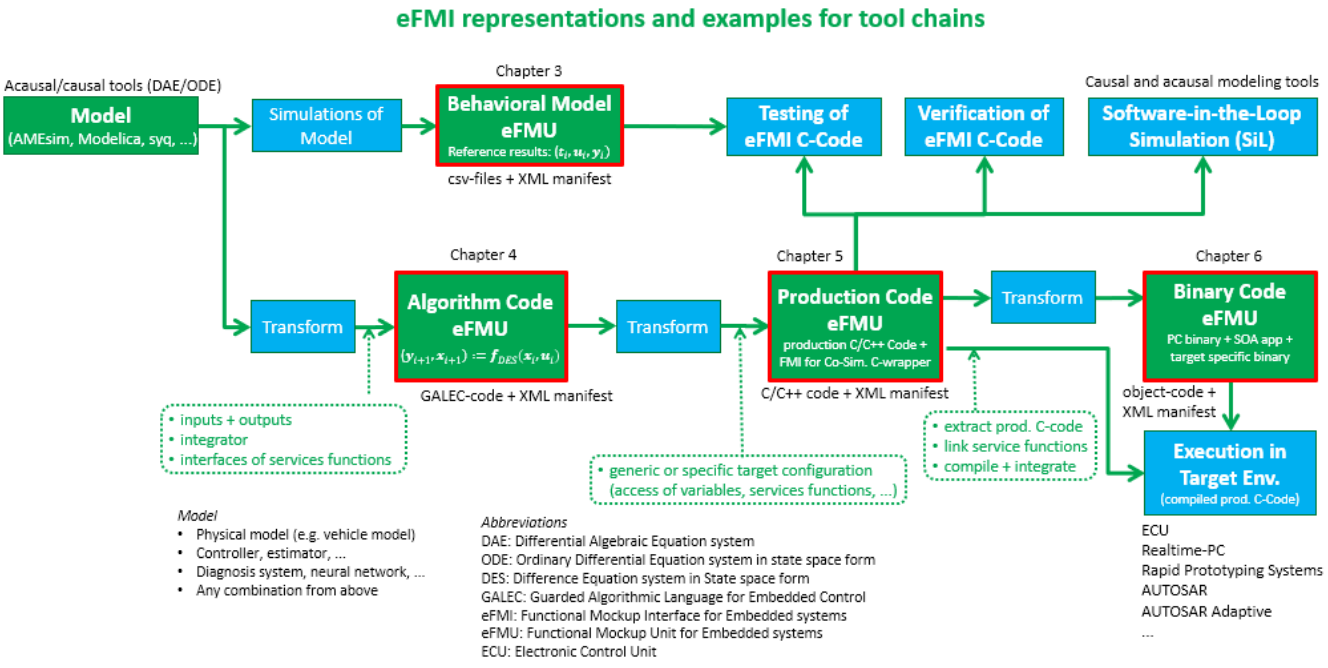
Thereafter the eFMI standard is specified starting with the description of the overall container

structure of an eFMU (Functional Mock-Up Unit for embedded systems) in section Chapter 2.

The following sections Chapter 3, Chapter 4, 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:



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 Mockup 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.

.5. 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 5.2). 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, ...)	not fixed, but description of the actual interface
Implementation	one implementation (one source code, one binary)	any number of implementations (target, vendor and "architecture" 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

```
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 4.2.6. 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 [GALEC Language]. 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:

```
function add
  input  Real u1;
  input  Real u2;
  output Real y;
algorithm
  y := u1 + u2;
end add
```

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

```
// GALEC function wrapper
function dot  // scalar product
  input  Real v1[:];
  input  Real v2[size(v1,1)];
  output Real y;
  external "C" y = dot(size(v1,1), v1, v2)
end dot

// C Code signature
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 Mockup 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
```

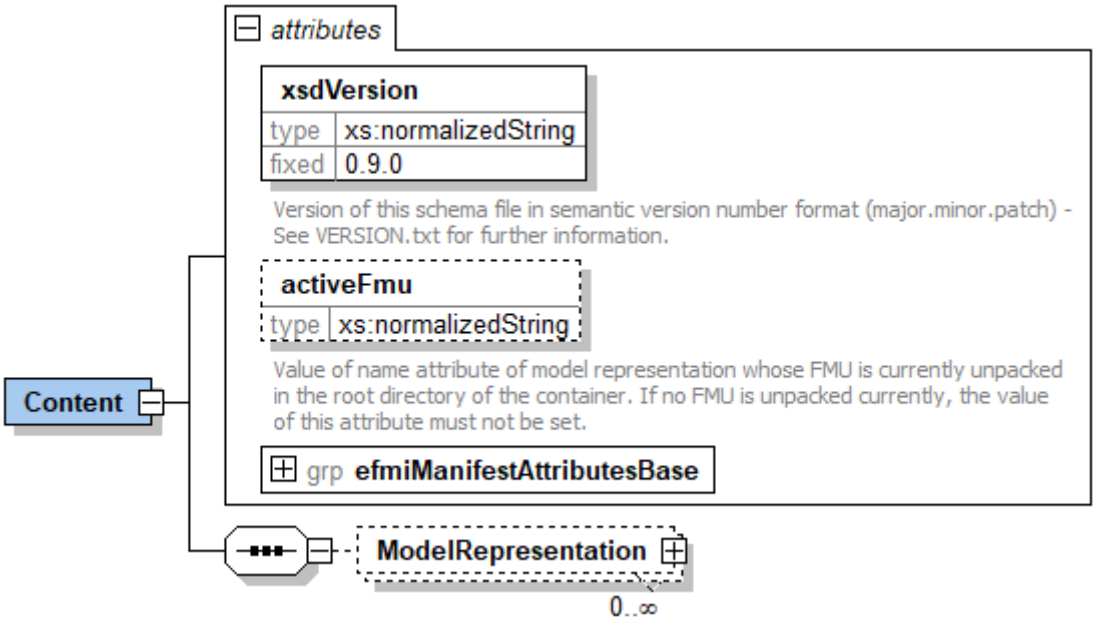
An eFMU may contain any number of additional subfolders below the `<eFMU root directory>` with one subfolder for each model representation. An eFMU container can contain only one Behavioral Model Representation, one Algorithm Code Model Representation, but can contain multiple Production Code Model Representations and also multiple Binary Code Model Representations. Each Model Representation itself can be organized in subfolders. It must have a dedicated manifest file. Other files describing the model representation such as code, an FMU, documentation, or license files may be organized in this subfolder.

The following diagram sketches the eFMU containers visually (details are given in the next sub-section):

[efmi container architecture] | *images/efmi-container-architecture.png*

2.1. Content description (efmiContainerManifest.xsd)

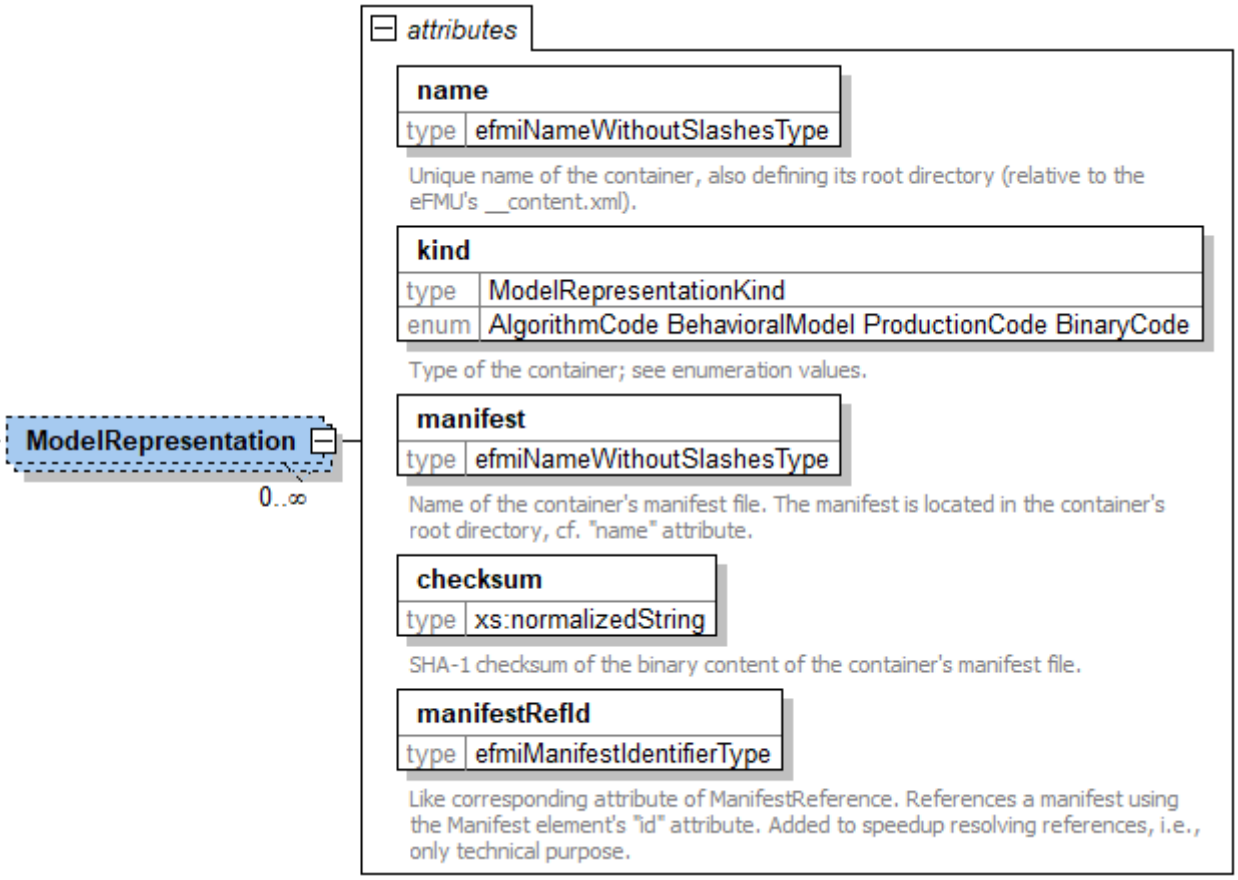
The `__content.xml` file is the registry for all model representations in the eFMU container. It has the following schema definition:



Name	Description
`xsdVersion`	Version of the `__content.xml` schema file in *semantic version number* format (https://semver.org).

Name	Description
`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.
`efmiManifestAttributesBase`	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 checksum of the binary content of the manifest file. A checksum of the whole subfolder is not required, because the files belonging to a model representation and their checksums are listed in the manifest file itself.

Name	Description
`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 version="1.1" encoding="utf-8"?>
<Content xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="schemas/efmiContainerManifest.xsd"
         xsdVersion ="0.9.0"
         efmiVersion="1.0.0"
         id         ="{92b7edbe-e77d-419a-8457-bf8d452a98f6}"
         name       ="MyModel"
         generationDateAndTime="2021-02-27T15:43:25Z"
>
    <ModelRepresentation kind         ="ProductionCode"
                         name         ="TLGeneratedCode_v1"
                         manifest     ="mark.xml"
                         checksum     ="e29810938a2a535dc8f6f9b8f51c5febe834ee01"
                         manifestRefId="63f8c810-f008-47f0-a4b6-7a243f83e46b" />
    <ModelRepresentation kind         ="AlgorithmCode"
                         name         ="algoCode_v1"
                         manifest     ="luke.xml"
                         checksum     ="e29810938a2a535dc8f6f9b8f51c5febe834ee05"
                         manifestRefId="63f8c810-f008-47f0-a4b6-7a243f85e46b" />
    <ModelRepresentation kind         ="BinaryCode"
                         name         ="binCode_v1"
                         manifest     ="matthew.xml"
                         checksum     ="e29810938a2a535dc8f6f9b8f51c5febe834ee08>"
                         manifestRefId="63f8c810-f008-47f0-a4b6-7a243f85e47b" />
</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 that is specific to the corresponding manifest file. For example, these two attributes are defined for the AlgorithmCode manifest file in the following way:

xsdVersion	
type	xs:normalizedString
fixed	0.13.0

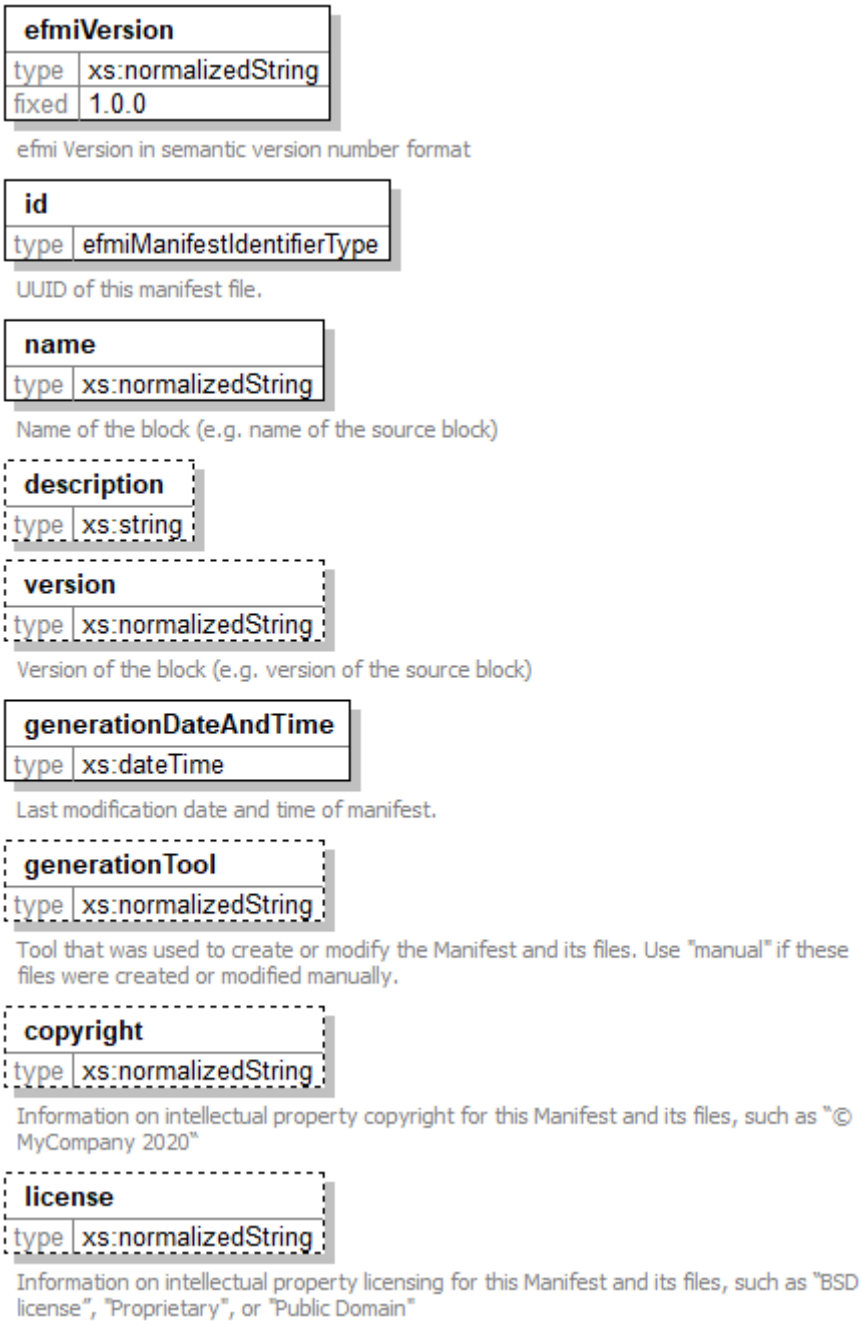
Version of this schema file in semantic version number format (major.minor.patch) - See VERSION.txt for further information.

kind	
type	ModelRepresentationKind
fixed	AlgorithmCode
enum	AlgorithmCode BehavioralModel ProductionCode BinaryCode

The attributes have the following meaning:

AName	Description
`xsdVersion`	The version of this manifest schema file in *semantic version number* format (https://semver.org).
`kind`	The type of this manifest file. The allowed values are `AlgorithmCode, ProductionCode, BinaryCode, BehavioralModel`.

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



The attributes have the following meaning:

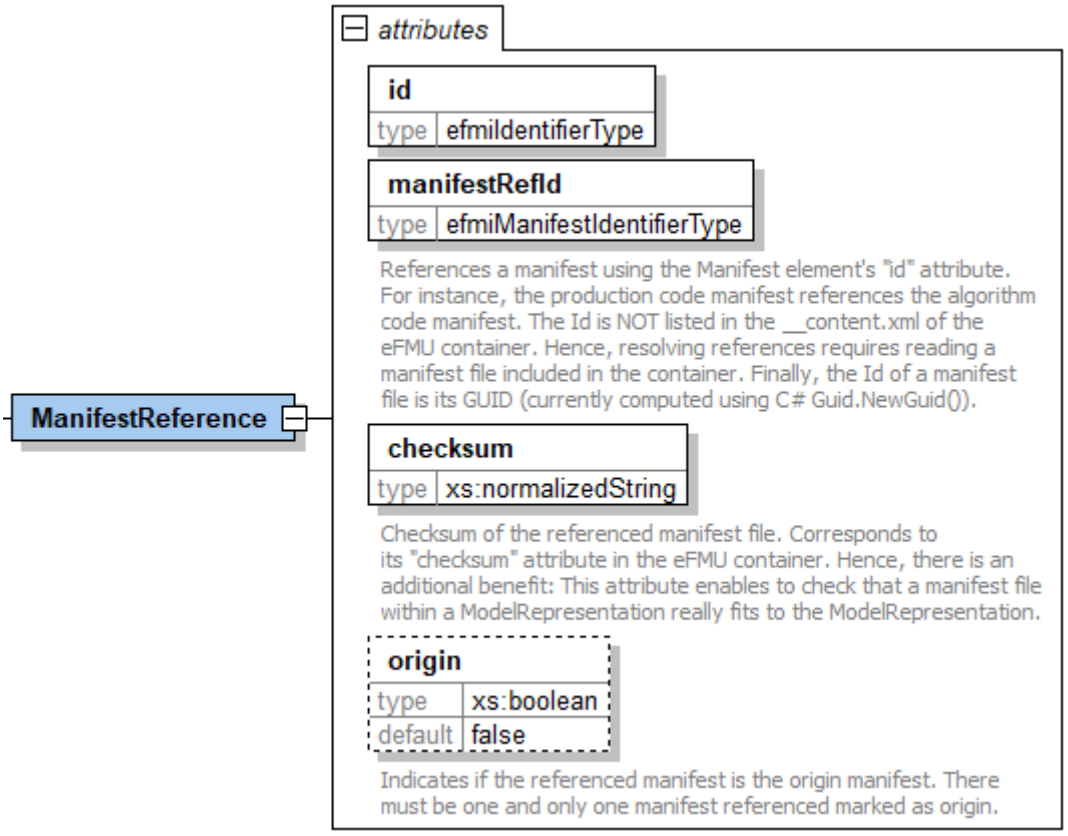
Name	Description
`efmiVersion`	The version of the efmi Standard in *semantic version number* format (https://semver.org) (currently: "0.7.0").
`id`	The UUID for this manifest file.
`name`	The name of the block (controller, diagnosis system etc.) as used in the modeling environment from which the manifest 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 the last modification of the manifest file. The format is a subset of "xs:dateTime" and should be: "YYYY-MM-DDThh:mm:ssZ" (with one "T" between date and time; "Z" characterizes the Zulu time zone, in other words, Greenwich meantime). *[Example: "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



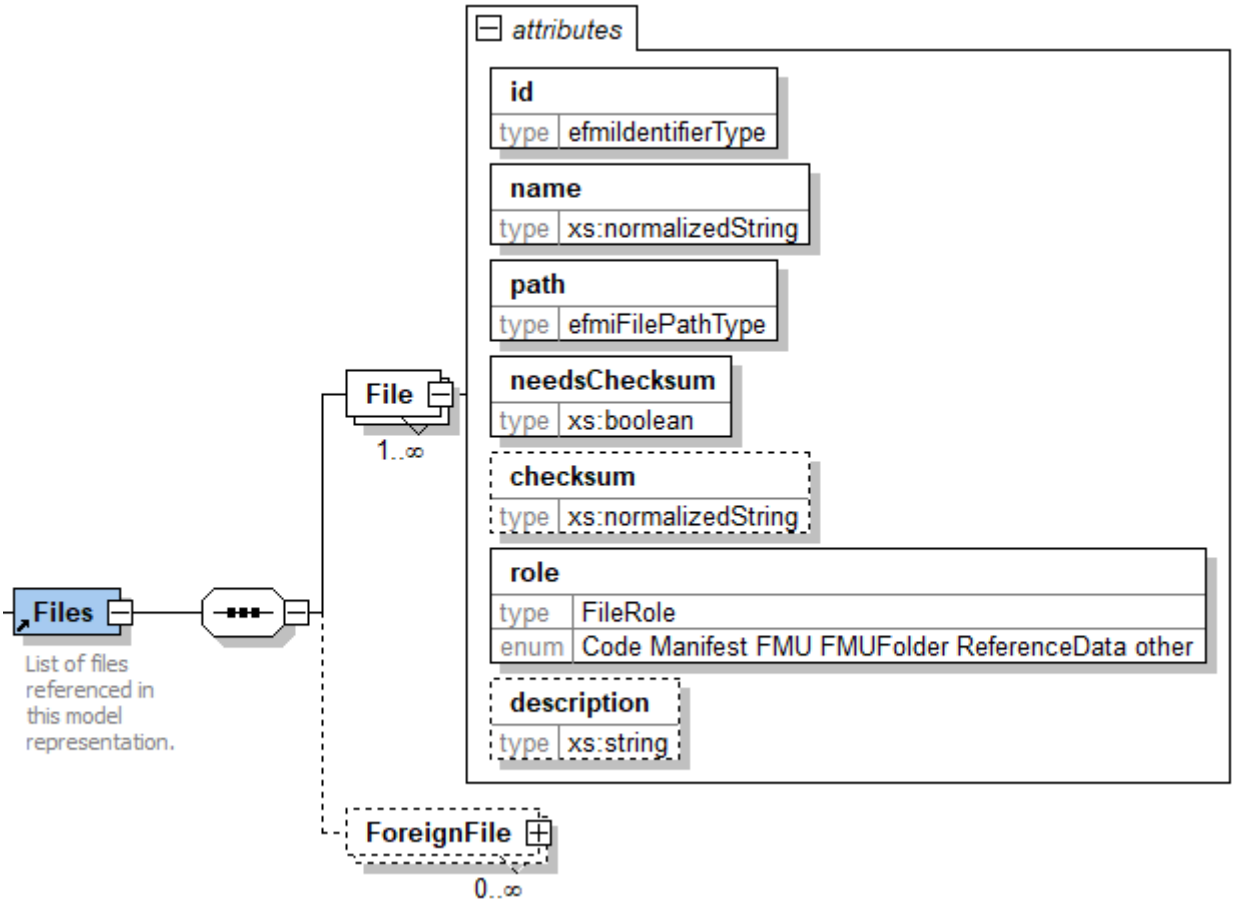
Name	Description
`id`	Unique id of the manifest reference entry. This id is used to establish cross manifest references.
`manifestRefId`	The unique GUID of the manifest. *[Note, the name of the associated model representation in the `__content.xml` file is not used, in order to decouple the manifest files from the container manifest.]*
`checksum`	The checksum of the referenced manifest file.
`origin`	Boolean flag to indicate if that referenced model representation is the one that was used to derive the current model representation.

Example:

```
<ManifestReferences>
  <ManifestReference id          ="ID_1"
                     manifestRefId="{63f8c810-f008-47f0-a4b6-7a243f85e46b}"
                     checksum    ="e29810938a2a535dc8f6f9b8f51c5febe834ee05"
                     origin      = true />
  <ManifestReference id          ="ID_2"
                     manifestRefId="{63f8c810-f008-47f0-a4b1-7a243f85222b}"
                     checksum    ="b4b84af148e587b95300d7a734302d1b911a6e58"
                     origin      =false />
</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 that the file is considered in the checksum calculation (default value "true")
`checksum`	The checksum of the file.

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:

```
<Files>
    <File id="ID_1" name         ="model.c"
                    path         ="./code/"
                    needsChecksum="true"
                    checksum     ="b4b84af148e587b95300d7a734302d1b912a6e58"
                    role         ="Code"/>
    <File id="ID_2" name         ="model.h"
                    path         ="./code/"
                    needsChecksum="true"
                    checksum     ="b4b84af148e587b95300d7a734402d1b911a6e58"
                    role         ="Code"/>
    <File id="ID_3" name         ="misra.doc"
                    path         ="./code/"
                    needsChecksum="true"
                    checksum     ="b4b84af148e587b95300d7a734302d1b914a6e58"
                    role         ="other"/>
    <File id="ID_4" name         ="model.arxml"
                    path         ="./code/"
                    needsChecksum="true"
                    checksum     ="b4b84af148e587b95300d7a734302d1b911a7e58"
                    role         ="other"/>
    <File id="ID_5" name         ="model.doc"
                    path         ="./description/"
                    needsChecksum="false"
                    role         ="other"/>
</Files>
```

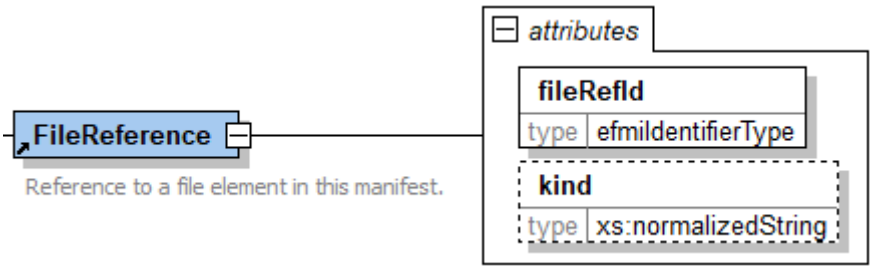
2.3.4. Referencing

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 ????

```
<CodeFile id="ID_13" fileType="ProductionCode">
    <FileReference fileRefId="ID_1" kind="SourceCode"/>
</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 and is the checksum at the point of creation of that container. Through comparison of both consistency can be ensured.

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:

```
<ManifestReferences>
    <ManifestReference id           ="ID_1"
                       manifestRefId="{63f8c810-f008-47f0-a4b6-7a243f85e46b}"
                       checksum     ="e29810938a2a535dc8f6f9b8f51c5febe835ee05"
                       origin       ="true"/>
    ...
</ManifestReference>
 ...
        <Variable name        ="T"
                  id          ="ID_33"
                  typeDefRefId="ID_25"
                  pointer     ="false"
                  value       ="0.1"
                  const       ="false"
                  volatile    ="true"
                  static      ="false" />
        <Variable name        ="_Clocks_interval"
                  id          ="ID_34"
                  typeDefRefId="ID_25"
                  pointer     ="false"
                  value       ="0.005"
                  const       ="false"
                  volatile    ="true"
                  static      ="false" />
        <Variable name        ="gearRatio"
                  id          ="ID_35"
                  typeDefRefId="ID_25"
                  pointer     ="false"
                  value       ="105"
                  const       ="false"
                  volatile    ="true"
                  static      ="false" />
    ...
        <DataReferences>
            <DataReference id="ID_100" variableRefId="ID_33" >
                <ForeignVariableReference manifestReferenceRefId="ID_1"
foreignRefId="ALG_ID_101"/>
            </DataReference>
            <DataReference id="ID_101" variableRefId="ID_34" >
                <ForeignVariableReference manifestReferenceRefId="ID_1"
foreignRefId="ALG_ID_100"/>
            </DataReference>
            <DataReference id="ID_102" variableRefId="ID_35" >
                <ForeignVariableReference manifestReferenceRefId="ID_1"
foreignRefId="ALG_ID_103"/>
            </DataReference>
    ...
```

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

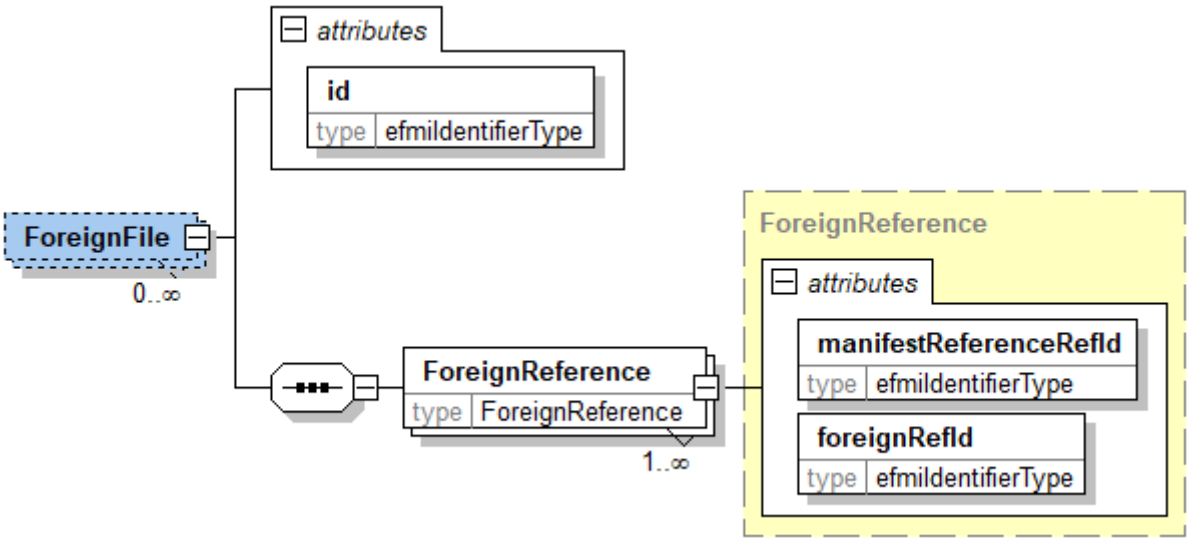
references the Algorithm Code Model Representation with the Manifest id "63f8c810-f008-47f0-a4b6-7a243f85e46b" in the eFMU container and the `foreignVariableRefId` attribute the element in that container with the given id (e.g. "ALG_ID_102").

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.



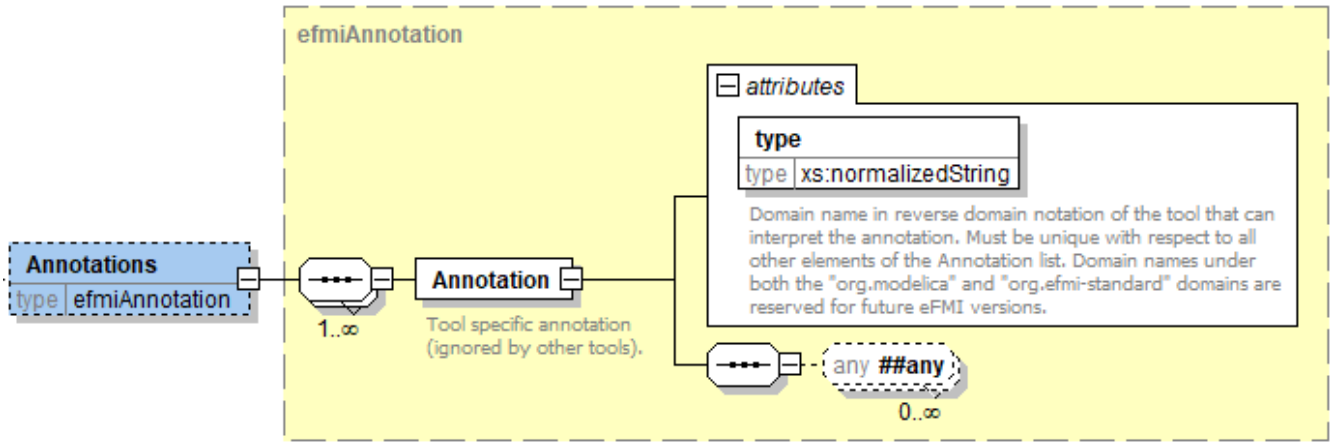
Name	Description
`id`	The (manifest local) id.
`ForeignReference`	Identifying the foreign manifest and the file inside the manifest.

Example:

```
<ManifestReferences>
    <ManifestReference id="ID_0" manifestRefId="{63f8c810-f008-47f0-a4b6-
7a243f85e46b}" checksum="???" origin="true"/>
    ...
</ManifestReference>
 ...
<Files>
    <File id="ID_1" name="model.c" path="./code"
          needsChecksum="true" checksum="e29810938a2a535dc8f6f9b8f51c5febe835ee05"
role="Code"/>
    <File id="ID_2" name="model.h" path="./code"
          needsChecksum="true" checksum="e29810938a2a535dc8f6f9b8f51c6febe835ee05"
role="Code"/>
    <File id="ID_3" name="misra.doc" path="./code"
          needsChecksum="true" checksum="e29810938a2a535dc8f6f9b8f51c5febe835ee06"
role="other"/>
    <File id="ID_4" name="model.arxml" path="./code"
          needsChecksum="true" checksum="e29810938a2a535dc8f6f9b8f51c5febe835ee08"
role="other"/>
    <File id="ID_5" name="model.doc" path="./description"
          needsChecksum="false" role="other"/>
    <ForeignFile id="ID_6">
        <ForeignFileReference manifestReferenceRefId="ID_0"
                              foreignRefId          ="ID_26" />
    </ForeignFile>
</Files>
```

Annotations (efmiAnnotation.xsd)

Additional data that a vendor might want to store and that other vendors might ignore are defined with element `Annotations` (this definition is identical to the corresponding element of FMI 3.0):



Name	Description
`type`	Domain name in reverse domain notation of the tool that can interpret the annotation. Must be unique with respect to all other elements of the Annotation list. Domain names under both the `"org.modelica"` and `"org.efmi-standard"` domains are reserved for future eFMI versions.

2.3.5. Checksum calculation

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 `checksum` 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 calculation for each file is based on a hash algorithm, currently SHA1 [SHA1Wiki] (https://en.wikipedia.org/wiki/Secure_Hash_Algorithms).

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 "unchecksummed" files (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. Behavioral Model Representation

3.1. Introduction

The optional *Behavioral Model* representation provides reference results for different scenarios to allow automatic verification of the Production and Binary Code representations. The reference results are stored in csv format under the Behavioral Model folder (for details see section Section 3.3). 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.

Basically, one reference result set consists of a table, where the columns represent the time and variables of the *original source model* (for example a *AMEsim*, *Modelica*, or *syq* model). Typically, these are the input and output variables of the Algorithm Code representation and the data is produced by simulating the original source model and storing the result in csv file format. Hereby, it is assumed that the simulations use the default values of the tunable parameters and the initial values of the states as defined in the `Startup()` method of the Algorithm Code model representation.

Automatic testing of a Production Code or Binary Code representation requires the following steps:

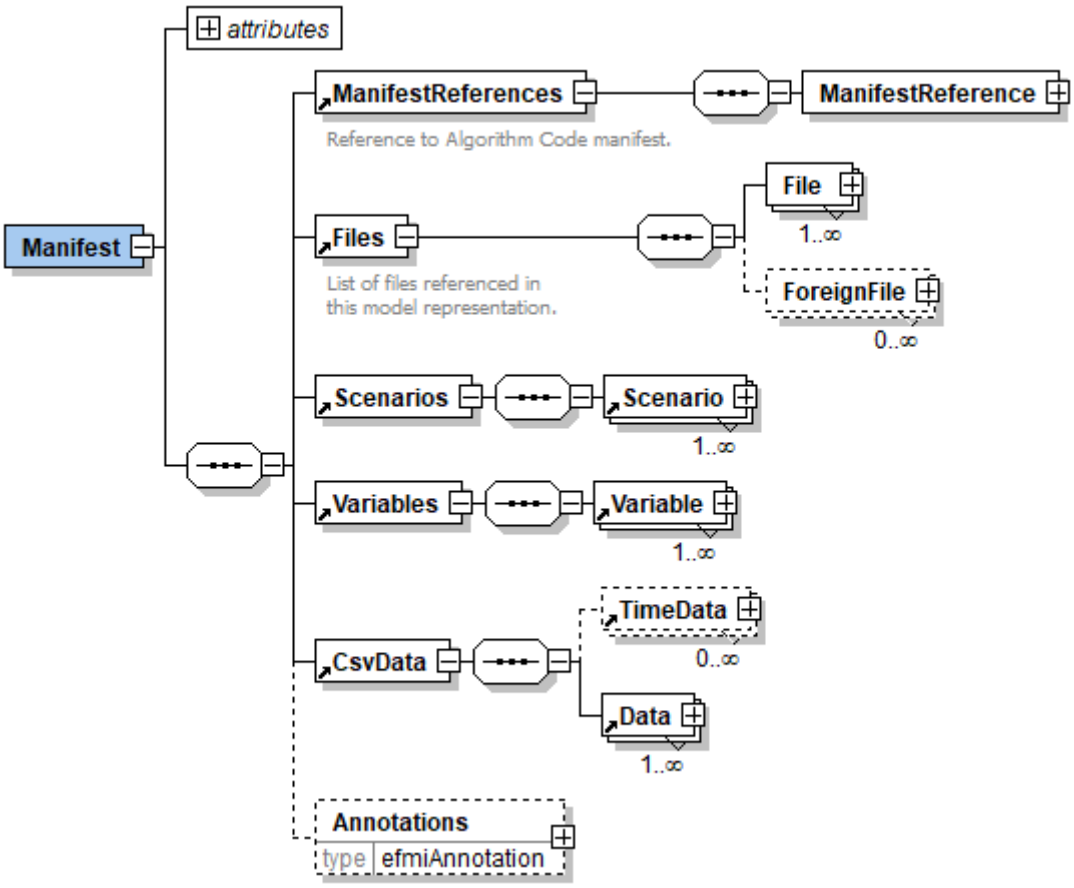
1. The Algorithm Code variable ids of the input/output variables in the Production Code manifest need to be determined (note, the C variable names of the variables are usually different to the variables names in the Algorithm Code). Therefore an indirect link between the variables in the Production Code manifest and the Behavioral Model manifest is established. With additional information in the Behavioral Model manifest the expected reference results for the input/output variables of the Production Code can be deduced from the corresponding csv-files inside the Behavioral Model folder.
2. The units of the variables are defined in the Algorithm Code manifest file.
3. The results produced by executing compiled Production Code resp. Binary Code have to be compared with the results stored in the Behavioral Model representation. In the Behavioral Model manifest optionally relative and absolute error tolerances are defined to assess the match between the data. A second possibilty to specify error tolerances is enabled by having whole data sets of time dependent lower and upper bounds of variables in the reference results. More details are given in the next subsection.

3.2. Behavioral Model Manifest

The *manifest file* of the Behavioral Model representation is an instance of an XML schema definition and defines the available scenarios with reference results and maximum acceptable deviations from them.

3.2.1. Definition of an eFMU Behavioral Model (efmiBehavioralModelManifest.xsd)

This is the root-level schema file of the Behavioral Model representation and contains the following definition:

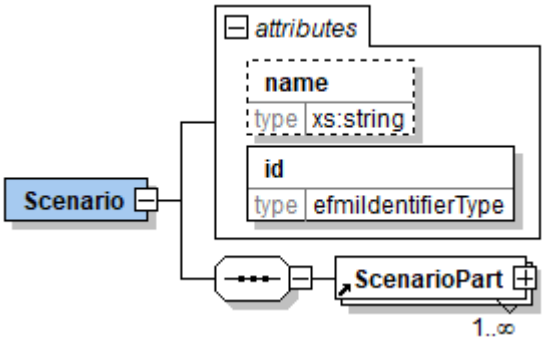


Element-Name	Description
`attributes`	The attributes of the top-level element are the same for all manifest kinds and are defined in section Section 2.3.1. Current kind-specific values: `kind = "BehavioralModel"`, `xsdVersion` (value is the current xsd version of the schema for the Behavioral model manifest), .
`ManifestReferences`	References to manifest files of other model representations for which referencing is needed within this Behavioral Model manifest. Mainly, the *Algorithm Code* manifest on which this Behavioral Model manifest is based on has to be listed. 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. There must be at least one file that contains reference results in csv format. This element is the same for all manifest kinds and is defined in section Section 2.3.3.
`Scenarios`	A scenario groups several simulation results (parts of one scenario) to one unit. At least one scenario definition must be present. For details see Section 3.2.2.
`Variables`	Required list of variables for which a link between columns in reference results and variables in the Algorithm code manifest is established in the Behavioral Model manifest. For details see Section 4.1.6.
`CsvData`	Optional element that defines how the columns of the csv files are mapped to the variables. It also provides information for the variables in each scenario part how acceptable deviations between simulation results of Production/Binary code and reference results are specified. For details see [definition-of-csvdata].

Element-Name	Description
Annotations	Additional data that a vendor might want to store and that other vendors might ignore. For details see Section 2.3.4.5.

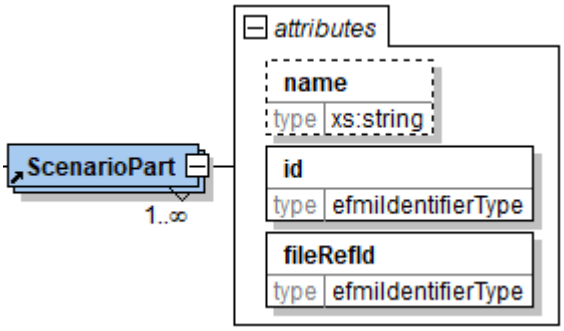
3.2.2. Definition of a Scenario (efmiScenarios.xsd)

A scenario (e.g. open loop test simulations) consists of one or more scenario parts (e.g. simulation runs with different numerical solvers).



Element-Name	Description
name	Optional name of the scenario.
id	The id of the scenario.

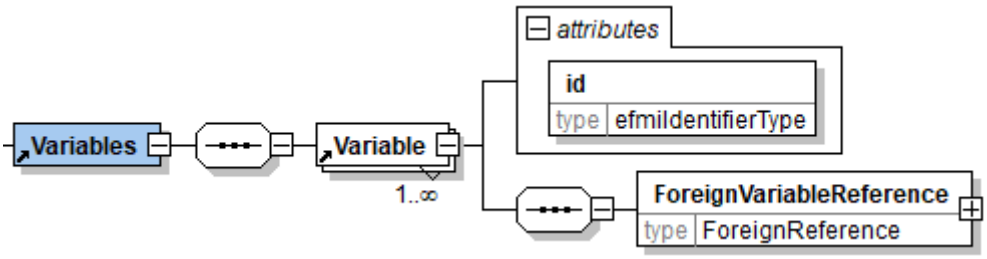
One simulation within a scenario is defined with a ScenarioPart element. The essential content of this element is the reference to a csv file.



Element-Name	Description
name	Optional name of the scenario part.
id	The id of the scenario part.
fileRefId	The reference id of the csv file, in which the reference result data for this scenario part is stored.

3.2.3. Definition of Variables (efmiVariable.xsd)

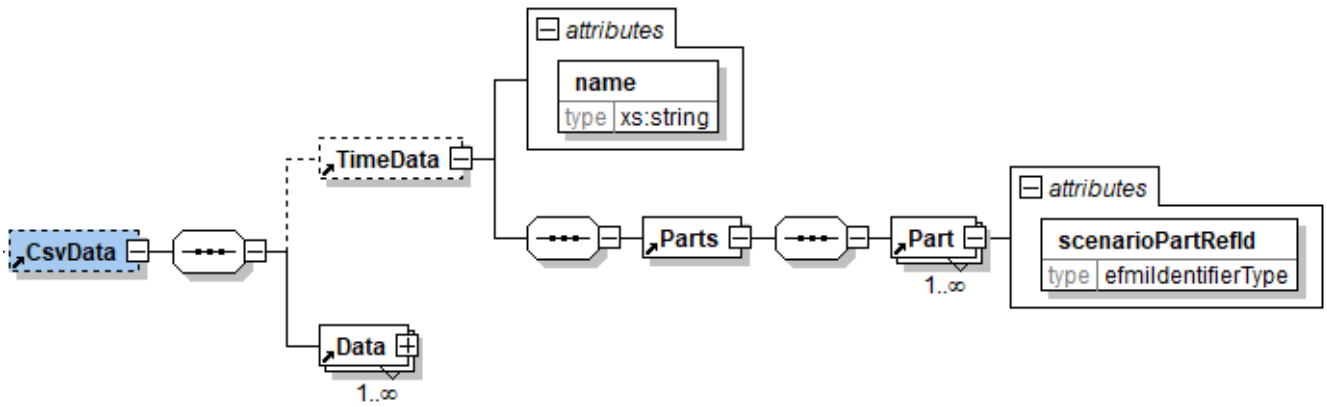
The variables to be compared in one of the scenario parts are listed in the following element:



Element-Name	Description
`id`	The id of the variable within the Behavioral Model manifest.
`ForeignVariableReference`	The reference to the variable defined in the Algorithm Code manifest file. For details see Section 2.3.4.3. A reference to other model representations is not allowed. It is not necessary to define all variables of the Algorithm Code manifest here. Only the variables for that reference data in csv files is provided need to be listed.

3.2.4. Definition of CsvData (efmiCsvData.xsd)

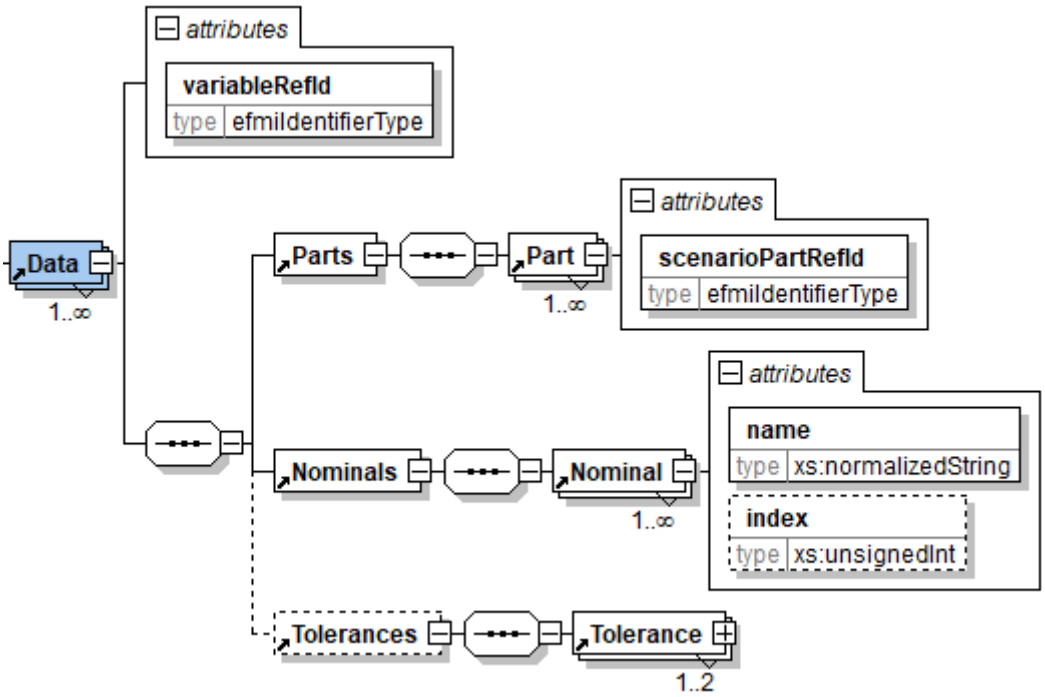
This element is the essential part of the Behavioral Model manifest and provides the information where for the variables the data can be found in the reference data files (= csv files). It also contains the information how the assessment can be realized, that deviations between the eFMI simulation results (by compiled Production Code or Binary Code) and the reference data in the csv files are acceptable.



Element-Name	Description
`TimeData`	Information where the time vectors can be found in the csv files.
`name`	The name of the time variable in the header of the csv files that are referenced by the listed scenario parts in `Part`.
`SenarioPartRefId`	The reference id of the scenario part to which this definition of the time vector is associated with.

Element-Name	Description
Data	Information about reference data and acceptable deviations associated with all variables (without time) of all scenario parts.

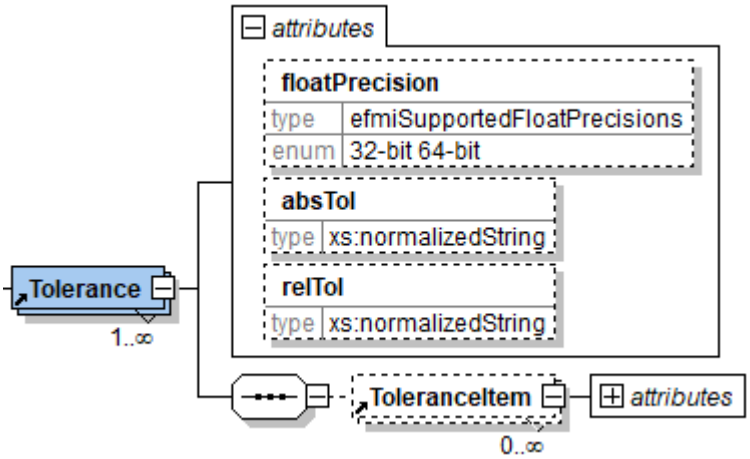
In one element Data the information about reference data and acceptable deviations associated with one variables (but not time) of one or several scenario parts are contained. Several scenario part can only be included, if the information to be provided is identical for these scenario parts. For the scenario parts, for which the information is different for a specific variable, a new element Data has to be listed for this variable. The whole list of all Data elements contains a combination of variables and scenario parts. It is not permitted to have the same combination twice, because otherwise the information is not unique.



Element-Name	Description
variableRefId	Reference id of the variable to be considered in this Data element.
scenarioPartRefId	Reference id of the associated scenario part for which the information is provided in this Data element. Several scenario parts can be listed within the element Parts.
Nominal	The nominal reference value of the variable that is associated with a column of the table in the csv files. If the variable is a vector or a multidimensional array, then each relevant component of the vector/array is listed by a separate element Nominal.
name	The name of the nominal variable in the header of the csv files that are referenced by the listed scenario parts in Part.

Element-Name	Description
index	Index of vectors resp. flattend index of multidimensional arrays. The element has to be absent for scalar variables and is required for vectors and mutlidimensional arrays. The index corresponds to the referenced Algorithm code variable (referenced by the element variableRefId and the corresponding element ForeignVariableReference in the element Variable). For multidimensional arrays the scalar index is according to a *row-major* order of all elements of the array.
Tolerances	Optional error tolerance information to be used for comparison of computed and given values in the csv files of the variable considered in this Data element.

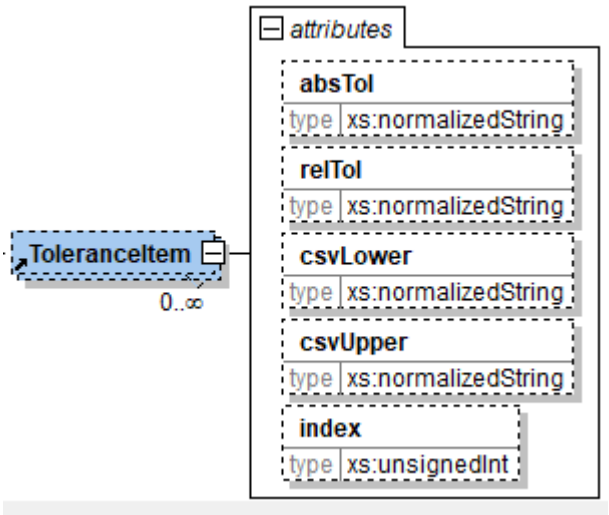
The details of the error tolerance information valid for one variable (scalar or vector or multidimensional array) is given with the following element. For each possible value of floatPrecision maximum one element Tolerance has to be present.



Element-Name	Description
floatPrecision	Floating point precision that has to be used to run the compiled Production Code or Binary code to be compared with the reference results (32-bit or 64-bit).
absTol	Optional default value for the absolute error tolerance that should be used for signal comparisons. For vectors or multidimensional arrays these default values are used for all components.
relTol	Optional default value for the relative error tolerance that should be used for signal comparisons. For vectors or multidimensional arrays these default values are used for all components.
ToleranceItem	Optional list of detailed information about error tolerances and additional columns for time-dependent lower/upper bounds of nominal reference result values.

The element ToleranceItem contains detailed error tolerance information about a scalar variable resp. one component of a vector/multidimensional array. If the considered variable is a scalar, then only one element ToleranceItem has to be present within the list of the element Tolerance. For vectors or multidimensional arrays only entries for relevant indices are needed and the values of

the attribute `index` have to be different for each of the entries.



Element-Name	Description
`absTol`	Optional absolute error tolerance of the scalar value considered in this tolerance item. If there is already a default value of `absTol` specified in the element `Tolerance`, then this more specific value of the element `ToleranceItem` has to be used.
`relTol`	Optional relative error tolerance of the scalar value considered in this tolerance item. If there is already a default value of `relTol` specified in the element `Tolerance`, then this more specific value of the element `ToleranceItem` has to be used.
`csvLower`	Optional name of the lower bound variable in the header of the csv files that are referenced by the listed scenario parts in `Part`.
`csvUpper`	Optional name of the upper bound variable in the header of the csv files that are referenced by the listed scenario parts in `Part`.
`index`	Index of vectors resp. flattend index of multidimensional arrays. The element has to be absent for scalar variables and is required for vectors and mutlidimensional arrays. The index corresponds to the referenced Algorithm code variable (referenced by the element `variableRefId` and the corresponding element `ForeignVariableReference` in the element `Variable`). For multidimensional arrays the scalar index is according to a *row-major* order of all elements of the array.

The lower and upper bound variables are not listed or referenced elsewhere in the manifest. The corresponding columns in the csv files contain data to define time-dependent lower/upper bounds of an acceptable simulation with the given float precision in the element `Tolerance`.

It is permitted to set all elements `absTol`, `relTol`, `csvLower` and `csvUpper`. If `absTol` or `relTol` are set, then `csvLower` and `csvUpper` cannot be set and vice versa. For the case, that `csvLower` and `csvUpper` are set and if there is already a default value of `relTol` or `absTol` specified in the element `Tolerance`, then these default values have to be ignored for the specific scalar variable/component in this `ToleranceItem`.

3.2.5. Comparison of signals

The Behavioral Model manifest contains the neccessary information to use inputs in csv files and to compare the simulation results of compiled Production Code or Binary Code with reference results in csv files. For the assessment, if the result deviations are acceptable, two cases have to be distinguished (depends individually on each variable and scenario part):

1. Check by using absolute and/or relative error tolerances or
2. Check by using lower and/or upper bounds.

For a scalar variable there is a column in a csv file that corresponds to the reference result data of this variable. Each row of the data is associated to a time instant of the time vector. In the following the data vector is called `y_ref`. The corresponding simulation result is called `y_sim` (also a time dependent vector with the same length as `y_ref` and the values of the variable at the time instants given by the time vector). If the check is based on lower/upper bounds, then there are further columns in the csv files that are associated to the time dependent lower and upper bounds of the variable. In the following the vectors are called `lower` and `upper`. The i-th component of all the described vectors are access by `y_ref_i`, `y_sim_i`, `lower_i` and `upper_i`.

If for all time instances `t_i` the following holds, then the test is passed otherwise not:

1. <code>abs(y_ref_i - y_sim)_i ≤ max(absTol, relTol*abs(y_ref_i))</code> resp.
2. <code>lower_i ≤ y_sim_i ≤ upper_i</code>

3.3. Behavioral Model Data

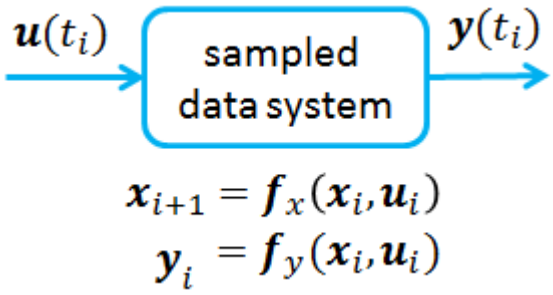
The csv files as they are contained in a Behavioral Model representation are according to (https://en.wikipedia.org/wiki/Comma-separated_values). In the first line there is a list of header names that define the names of the columns separated by a colon. In the following lines for each of the variables defined in the header a numeric value is provided separated by colon.

The unit of the time is seconds. Values of Boolean variables have to be represented by the Integer values 0 (false) and 1 (true).

Example:

```
Time,wLoadRef,wMotor,vMotor,isReset
0,0,10,-200,0
0.001,0.003,10,-193.70000000000002,0
0.002,0.006,10,-187.41e3,1
0.003,0.009,10,-181.12958499999996,1
0.004,0.012,10,-174.85834414999999,0
```

Chapter 4. Algorithm Code Model Representation



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 explicitely 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

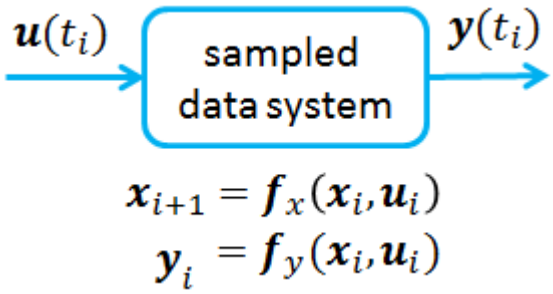
- of a *manifest file* in XML format in which all interface variables are defined (see section [Algorithm Code Manifest]),
- one *code file* with extension *.alg* that represents the executable part of the block consisting of a `block` with declarations, and mandatory definitions of the three methods `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 [GALEC - The Algorithm Code Language]).

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`.

4.1. Manifest

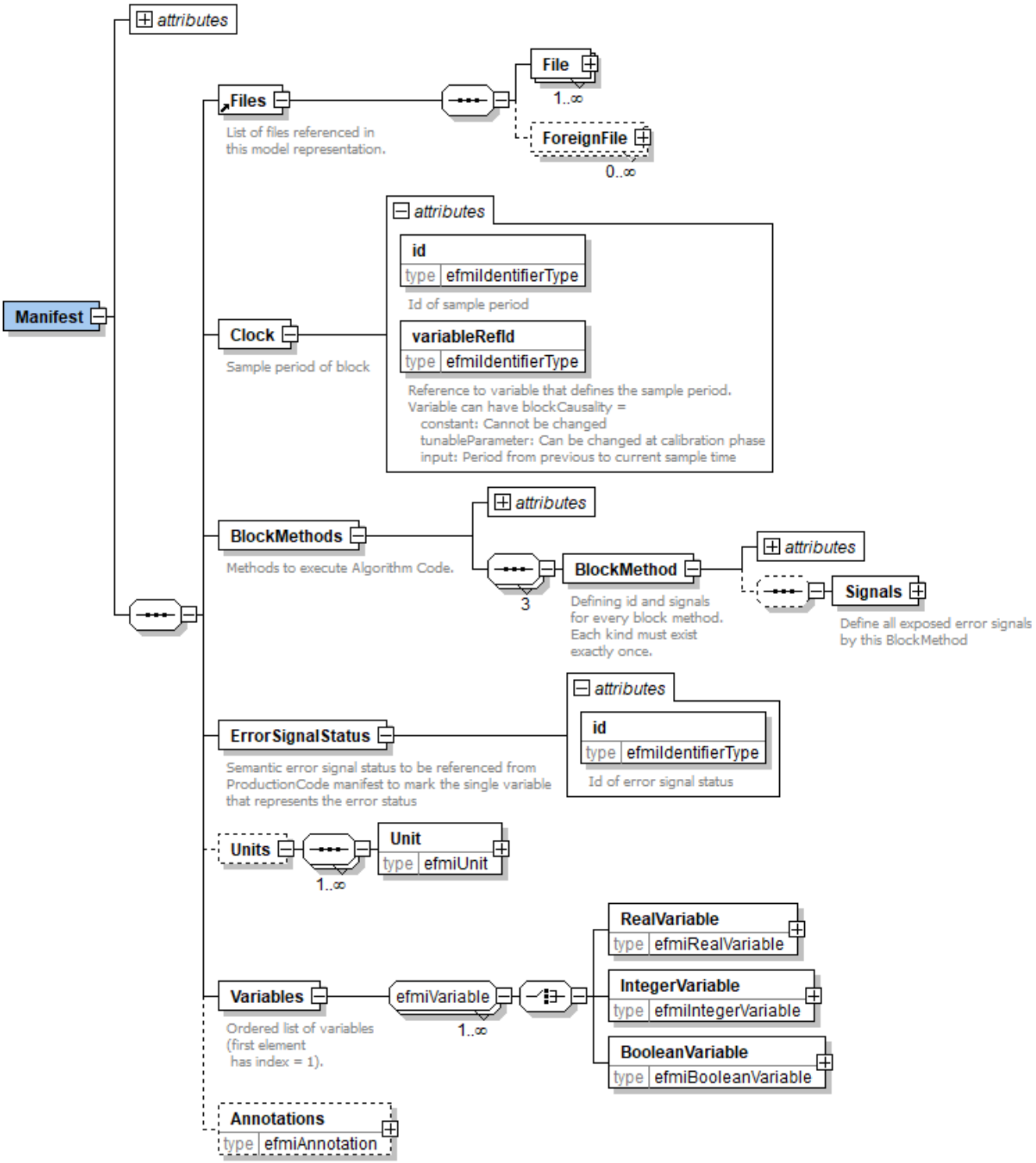


$$x_{i+1} = f_x(x_i, u_i)$$
$$y_i = f_y(x_i, u_i)$$

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.

4.1.1. 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:



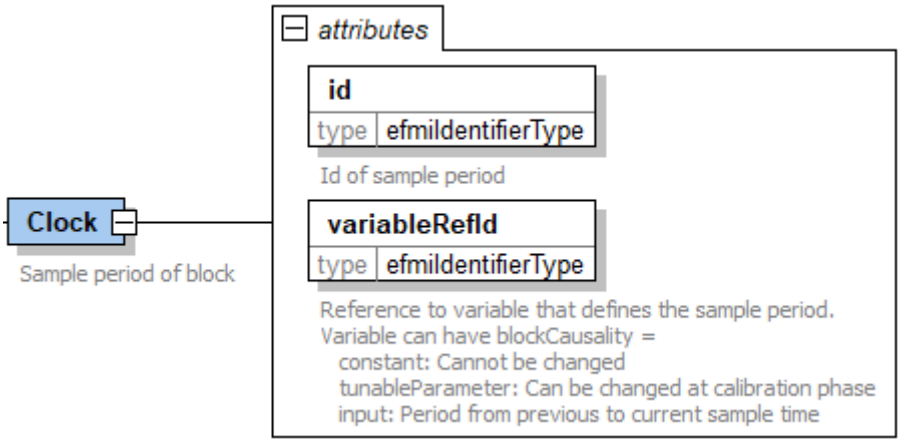
On the top level, the schema consists of the following elements (see figure above):

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 fixed or variable sample period defined by a block variable. For details see Section 4.1.2.
`BlockMethods`	The properties of the block methods DoStep, Recalibrate, and DoStep. For details see Section 4.1.3.
`ErrorSignalStatus`	Semantic error signal status to be referenced from ProductionCode manifest to mark the single variable that represents the error status. For details see Section 4.1.4.
`Units`	An optional global list of unit and display 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 4.1.5.
`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 4.1.6.
`Annotations`	Additional data that a vendor might want to store and that other vendors might ignore. For details see Section 2.3.4.5.

4.1.2. Definition of Clock

Element `Clock` provides a reference to the fixed or variable sample period defined by a block variable. The block should be executed *periodically with the defined fixed or variable sample period.*

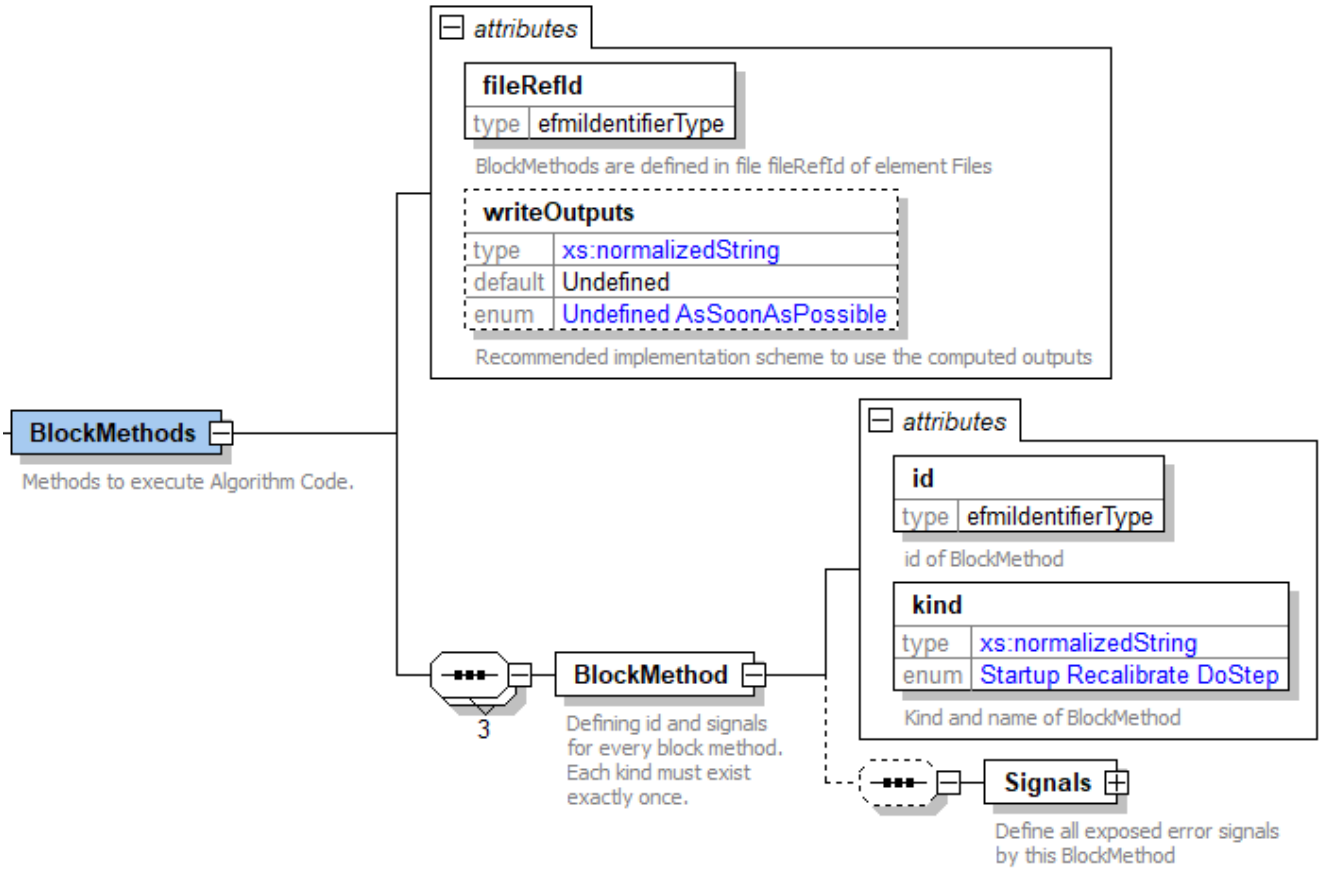


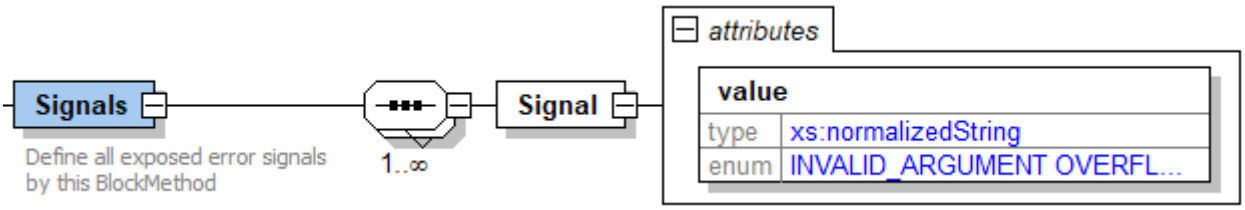
Element-Name	Description
`id`	The id of the sample period of the block.
`variableRefId`	Reference to the variable in `<Variables>` that defines the sample period. This variables is only allowed to have the following values for variable attribute `blockCausality`: `constant`: Sample period cannot be changed. `tunableParameter`: Sample period can be changed in the calibration phase. `input`: Sample period from previous to current clock tick

The referenced variable `variableRefId` defines the sample period for which *the block was designed*. When the production code of this block is integrated in the target system (for example as AUTOSAR runnable), then it is expected that the block is executed as periodic sampled data system with this sample period. It might be that also a slightly changed sample period in the target system may still result in reasonable performance.

4.1.3. 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>, see section Section 2.3.3) in which the code of the block methods is stored.
`writeOutputs`	Defines the recommended implementation scheme to utilize the calculated outputs. Default is `Undefined`. The currently only allowed other value is `AsSoonAsPossible`, meaning to utilize the outputs at once when they are computed, more details are given below.
`id`	The ID of the block method
`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 (for details Section 4.2.5.1) Attribute `value` defines the value of the signal. Currently, the following values are possible: `"INVALID_ARGUMENT"` (= the value of an input variable is not correct) `"OVERFLOW"` (= the value of a variable is Inf) `` `"NAN" `` (= the value of a variable is Not-A-Number) `"SOLVE_LINEAR_EQUATIONS_FAILED"` (= failed to solve a linear equation system) `"NO_SOLUTION_FOUND"` (= no solution found for other equation systems) `"UNSPECIFIED_ERROR"` (= error not further specified)

The scheme `writeOutputs = "AsSoonAsPossible"` is typically used when the controller computes the outputs for the *current clock tick* (e.g. integrates from the previous to the current clock tick). *Pseudo-Code* for this scheme:

```
self = <instance of efmi component>
<initialize self with the manifest start values> or self.Startup()
<write outputs>
<wait until clock starts>

<at every clock tick>
   <read inputs>
   self.DoStep()
   <write outputs>
   if <calibration phase and tunable parameters available>
       <set tunable parameters>
       self.Recalibrate()
   end
   <wait for next clock tick>
<end>
```

The drawback of this scheme is that the computing time of `efmu.DoStep()` introduces a time delay until the outputs are available.

Note, it is also possible to write the outputs inside DoStep *directly after they are computed* (without waiting until all statements are processed and the method returns). This implementation scheme of the Production Code is *recommended* if attribute `writeOutputs` has value `AsSoonAsPossible`.

[*There are also other implementation schemes that might by useful (currently, it is not possible to state this in the Manifest file). Examples*:

Write outputs at next clock tick

This scheme is typically used when the controller computes the outputs for the next clock tick (e.g. integrates from the current to the next clock tick). Pseudo Code:

```
self = <instance of efmi component>
<initialize self with the manifest start values> or self.Startup()
<write outputs>

<at every clock tick>
   <write outputs>
   <read inputs>
   self.DoStep()
   if <calibration phase and tunable parameters available>
       <set tunable parameters>
       self.Recalibrate()
   end
   <wait for next clock tick>
<end>
```

The drawback of this scheme is that the inputs are extrapolated over the sample period because the inputs at the next clock tick are utilized in `DoStep` *but are not known when* `DoStep` *is called.*

Two different clocks for reading inputs and writing outputs

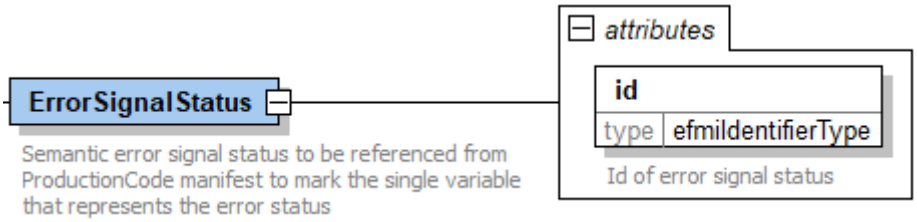
The reading of inputs and the writing of outputs might be performed with different clocks that have the same sample period, but the clock for the outputs is shifted relative to the clock for the inputs.

Event clock (purely event based)

The block might be triggered by an external event (e.g. at a particular angle of the engine shaft). The sample period (from the previous to the current clock tick) is provided as input signal.

4.1.4. Definition of ErrorSignalStatus

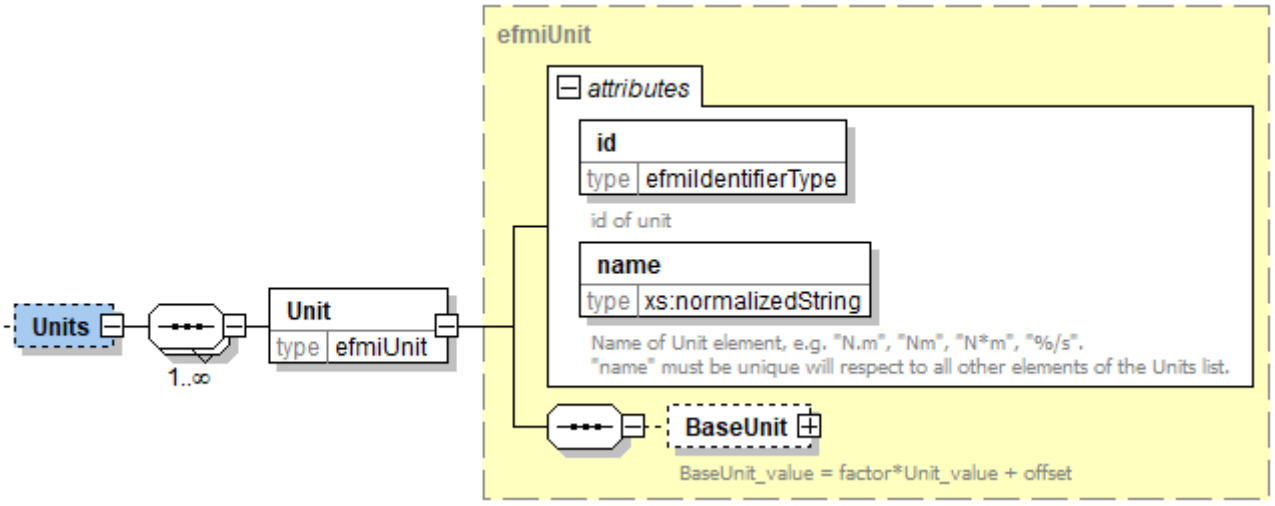
This element defines the single, hidden, error signal variable that holds the error signal status and is referenced from the ProductionCode manifest. It consists only of attribute `id` that defines the ID of this hidden variable:

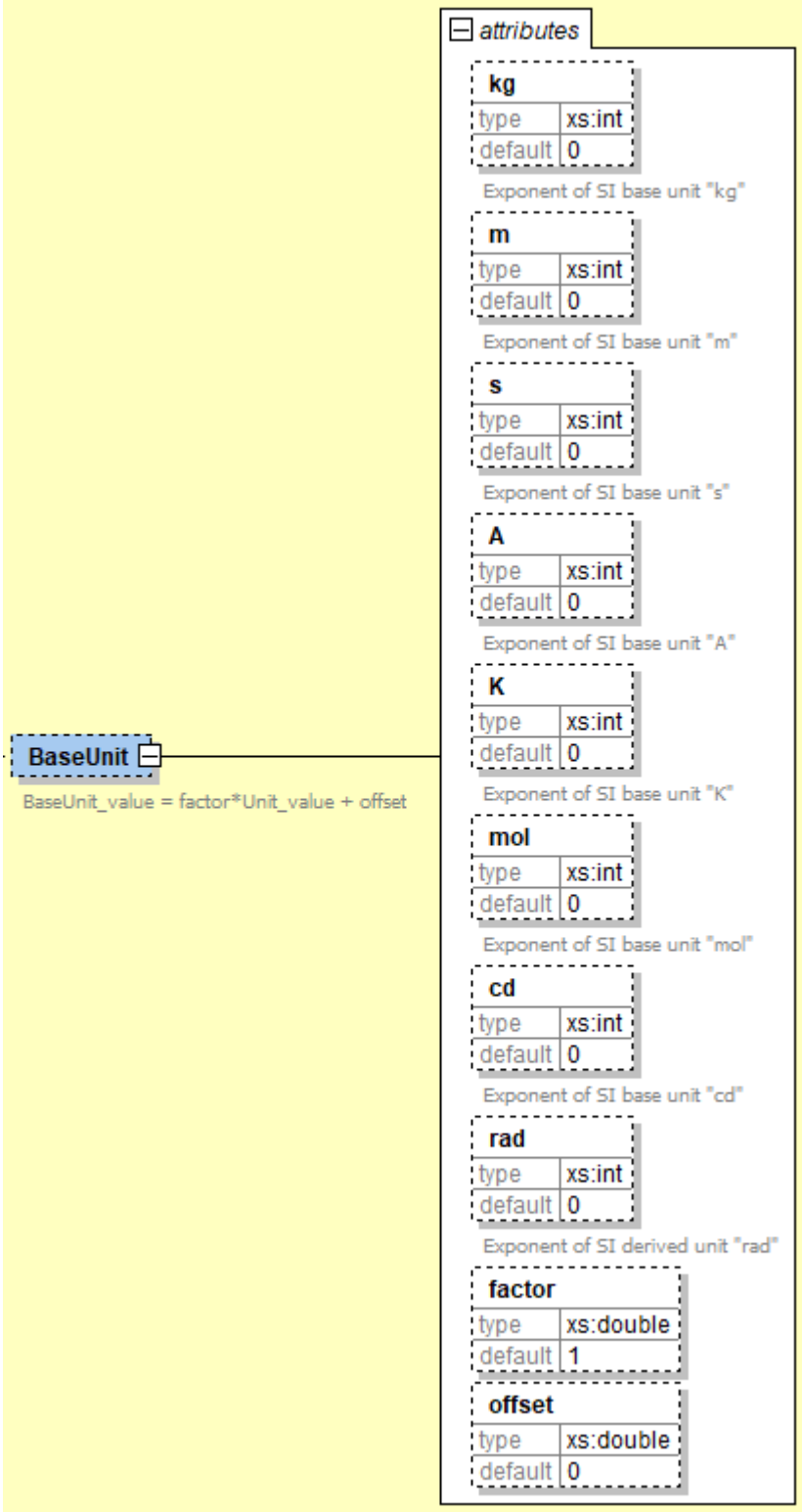


4.1.5. Definition of Units

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 AlgorithmCode manifest file and without element `DisplayUnit`:

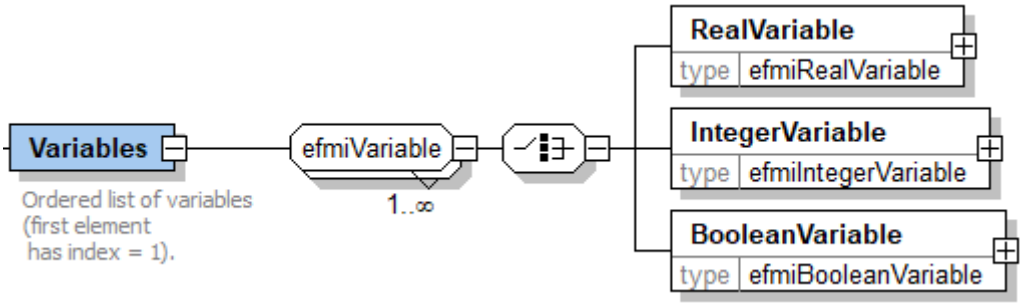




4.1.6. Definition of Variables

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`):

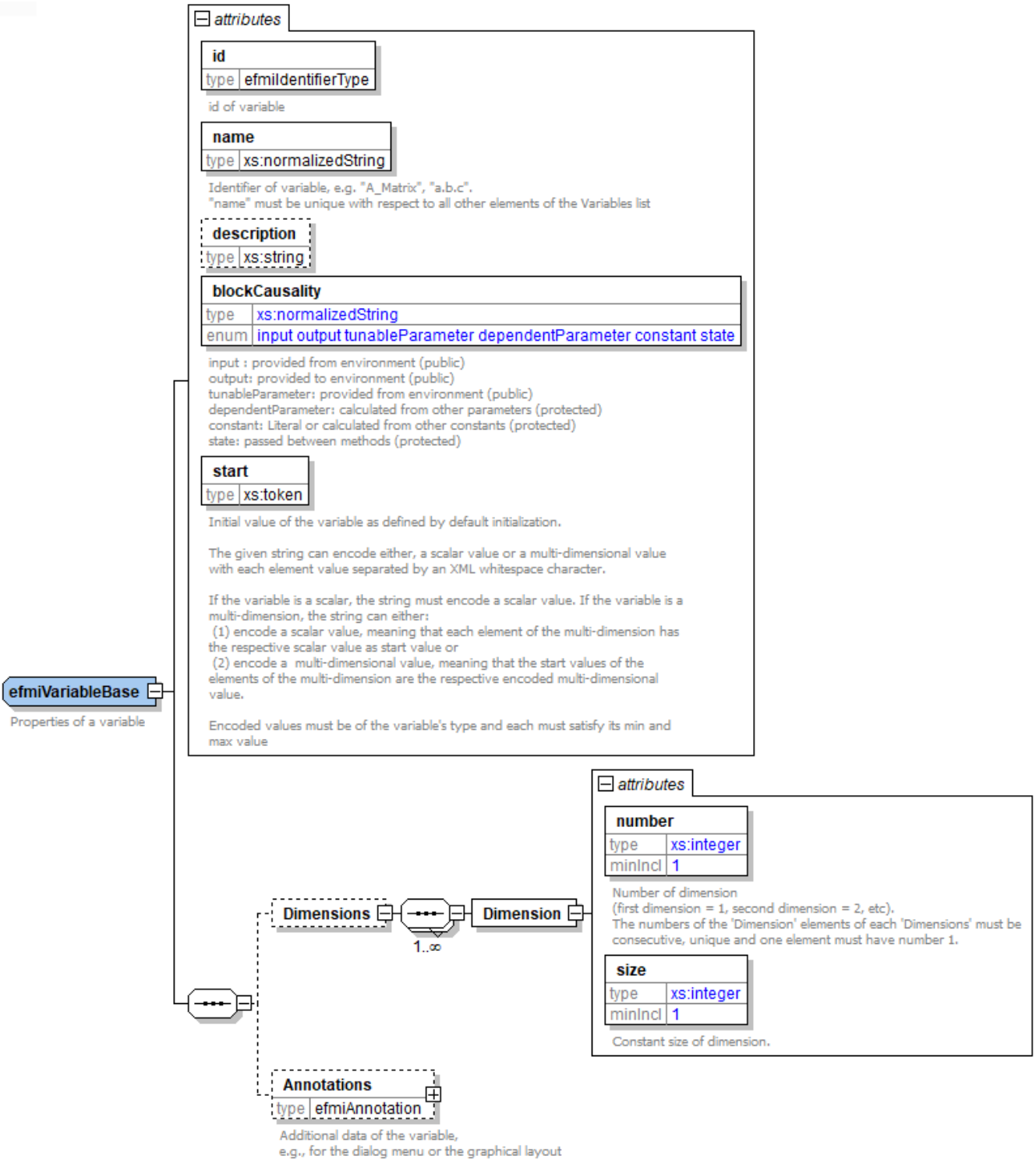


The schema definition contains basically the same information as element `ModelVariables` in FMI 3.0, but using mathematical instead of target types and having the following deviations:

- 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.
`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).

Name	*Description*
`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**
`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:

Attribute-Name	*Description*
unitRefId	Identifier of the unit of the variable defined in list `Units.Unit` (Section 4.1.5). 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.
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:

```
<Units>
  <Unit id="UnitID_1" name="s"/>
</Units

<Variables>
  <RealVariable id="ID_1" name="Ti" unitRefId="UnitID_1"
blockCausality="tunableParameter" start="0.1"/>
  <RealVariable id="ID_A" name="A" blockCausality="constant" start="1.1 1.2 2.1 2.2">
      <Dimensions>
           <Dimension number="1", size="4"/>
      </Dimensions>
  </RealVariable>
  <RealVariable id="ID_2" name="previous(I.x)"  blockCausality="state" start="0.0"
min="0.0" />
</Variables>
```

IntegerVariable-specific attributes

The following `IntegerVariable` specific attributes are defined:

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.

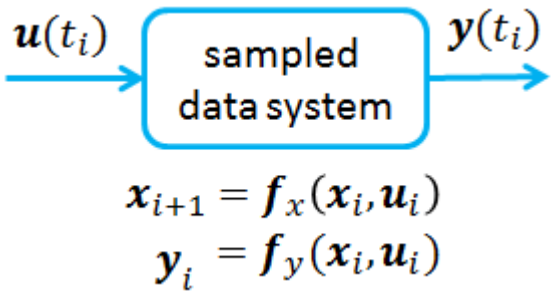
Examples:

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

BooleanVariable-specific attributes

The `BooleanVariable` element has no additional attributes.

4.2. GALEC: The Programming Language for Algorithm Code Containers' Source Code



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 4.2.7.

4.2.1. 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.

4.2.2. Notation Conventions

The concrete syntax of GALEC code is defined using *Extended Backus–Naur Form* (EBNF) according to ISO/IEC 14977. 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 $N_1 =_{lexical} \alpha$, 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 $d = \beta_1, \ldots, \beta_n$ be a single definition according to ISO/IEC 14977; β_i with $1 \leq i \leq n$ 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, $\square i,j \square\square +; i = j - 1; 2 \leq j \leq z: \delta_j$ *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***).

```
/*
    For-loop CONTAINING another for-loop.
    Thus, neither for-loop is BEFORE or AFTER the other.
    Both for-loops are function-call-FREE:
*/
for i in 1:size(A,1) loop
    /*
        If-statement PART OF a for-loop and CONTAINING a
        DIFFERENT for-loop. The if-statement is WITHOUT a
        function-call since it does NOT CONTAIN such:
    */
    if
        /*
            The 2ND CHILD of the if-statement is an expression:
        */
        mod(i,2) == 0
    then
        /*
            NESTED for-loop, i.e., a for-loop CONTAINED in
            another for-loop. The NESTED for-loop FOLLOWS its
            CONTAINING if-statement's 2ND CHILD:
        */
        for j in 1:size(A,2) loop
            /*
                Assignment α PRECEDING another assignment β, with
                which its 1ST CHILD is LEXICALLY-EQUIVALENT.
                The assignment is also BEFORE another assignment γ
                that is DIFFERENT to β; all three assignments are
                SIBLINGS:
            */
            A[i,j] := 1; // α
        end for;
    else
        /*
            Assignment β AFTER a PRECEDING assignment α with
            LEXICALLY-EQUIVALENT 1ST CHILD:
        */
        A[i,j] := 0; // β
    end if;
end for;
/*
    Assignment γ most likely not PART OF a for-loop,
    but for sure with exactly one function-declaration CONTAINER
    that trivially is its CLOSEST function-declaration CONTAINER:
*/
A[size(A,1), size(A,2)] := -1; // γ
```

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

```
function-declaration =
    ( "function" | "method" ),
    name,
    { parameter-declaration },
    [ "protected", { local-variable-declaration } ],
    "algorithm",
    { statement },
    "end",
    name,
    ";" ;
```

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*:

```
function foo
protected
algorithm
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:

```
for i in 1:3 loop // Outer loop.
    for j in 1:3 loop // Inner loop.
        if b1 then
            break i; // First break of outer and inner loop.
        else
            break j; // Illegal: Second break of inner loop.
        end if;
    end for;
    if b3 then
        break i; // Illegal: Second break of outer loop.
    end if;
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***.

4.2.3. 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:

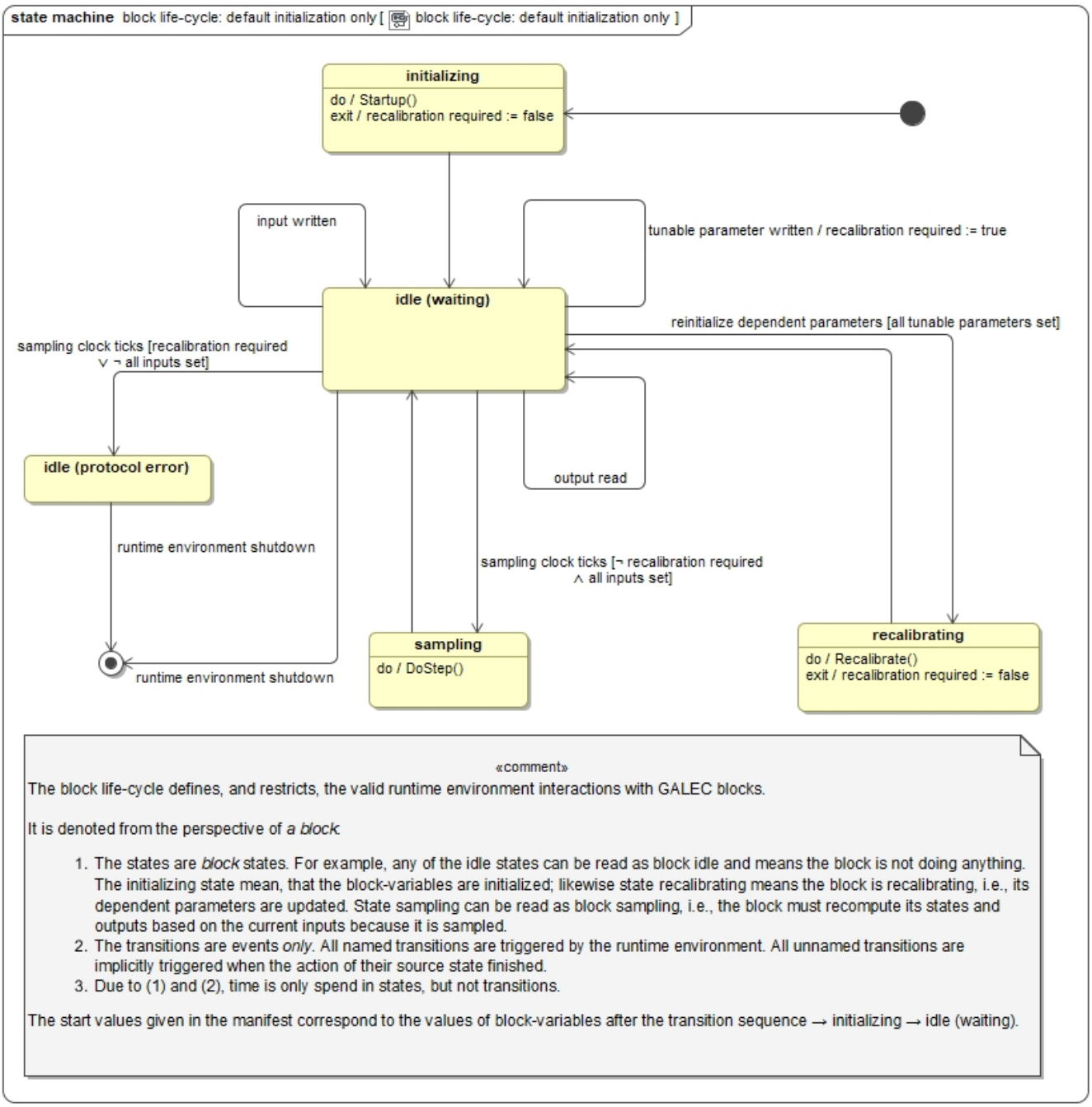
1. Not generate and call `Startup()`, but instead statically evaluate it and store start values in read only memory or only load them once when the runtime system boots.
2. Not generate a dedicated `DoStep()` function, but instead inline the implementation in the runtime environment.
3. Not generate `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, 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:

```
/*
    Initialization:
*/
Startup(c); // Assigns every block-variable a value, particularly outputs.
[[ process initial outputs of block ]]

/*
    Sampling cycle:
*/
while ([[ block not shutdown ]])
{
    if ([[ recalibration desired ]])
    {
        [[ set new tunable parameters of block ]]
        Recalibrate(c); // Recompute dependent parameters of block.
    }
    [[ set new inputs of block ]]
    [[ wait until sampling clock ticks ]]
    DoStep(c); // Recompute internal states and outputs of block.
    [[ process new outputs of block ]]
}
```

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

```
/*
    Initialization:
*/
Startup(c); // Assigns every block-variable a value, particularly outputs.
[[ process initial outputs of block ]]
[[ set new tunable parameters of block ]]
Recalibrate(c); // Recompute dependent parameters of block.

/*
    Sampling cycle:
*/
while ([[ block not shutdown ]])
{
    [[ set new inputs of block ]]
    [[ wait until sampling clock ticks ]]
    DoStep(c); // Recompute internal states and outputs of block.
    [[ process new outputs of block ]]
}
```

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:

```
/*
    Initialization:
*/
Startup(c); // Assigns every block-variable a value, particularly outputs.
[[ process initial outputs of block ]]

/*
    Sampling cycle:
*/
while ([[ block not shutdown ]])
{
    [[ set new inputs of block ]]
    [[ wait until sampling clock ticks ]]
    DoStep(c); // Recompute internal states and outputs of block.
    [[ process new outputs of block ]]
}
```

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:

```
/*
    Initialization:
*/
Startup(c); // Assigns every block-variable a value, particularly outputs.
[[ process initial outputs of block ]]

/*
    Sampling cycle:
*/
while ([[ block not shutdown ]])
{
    // Handle default setup:
    [[ set new inputs of block ]]

    // Handle first special case, modifying the default:
    v1 = [[ some value provided by the runtime environment ]];
    t1 = [[ read tunable parameter t1 ]];
    o1 = [[ read output o1 ]]; // Previous sampling output, or initial if
first sampling.
    if (o1 / t1 > v)
    {
        [[ set input i1 to v1 ]]
        if (t1 > 2*v)
        {
            [[ set tunable parameter t1 ]]
            Recalibrate(c);
        }
    }

    // Handle second special case (may amend the first case):
    v2 = [[ some value provided by the runtime environment ]];
    t2 = [[ read tunable parameter t2 ]];
    if (t2 < v2)
    {
        [[ set tunable parameter t2 ]]
        Recalibrate(c); // Recompute dependent parameters of block.
        [[ set input i2 to input i1 ]]
    }

    // Everything is prepared for next sampling:
    [[ wait until sampling clock ticks ]]
    DoStep(c); // Recompute internal states and outputs of block.
    [[ process new outputs of block ]]
}
```

4.2.4. General Syntactic and Semantic Rules

Lexemes

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

```
character = ? any valid ISO/IEC 10646:2017 code point ? ;

white-space = { space | new-line-character | comment } - ( ) ;

space = " " | ? tabulator (ISO/IEC 10646:2017 code point 9) ? ;

new-line-character =
    ? carriage return, line feed or carriage return followed by line feed
      (ISO/IEC 10646:2017 code point 13 or 10 or 13 followed by 10) ? ;

comment = single-line-comment | multi-line-comment ;

single-line-comment = "//", { character - ( new-line-character ) } ;

multi-line-comment = "/*", { character } - ( { character }, "*/", { character }
), "*/" ;
```

G-1.8 — G-1.17 (constants):

```
boolean = "false" | "true" ;

digit = (* ? any ISO/IEC 10646:2017 code point in range [48, 57]: ? *)
    "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

non-zero-digit = digit - ( "0" ) ;

integer = "0" | positive-integer ;

positive-integer = non-zero-digit, { digit } ;

real = ( integer-places, [ decimal-places ], [ exponent ] ) - ( integer ) ;

integer-places = integer ;

decimal-places = ".", digit, { digit } ;

exponent = ( "e" | "E" ), [ "+" | "-" ], digit, { digit } ;

constant = boolean | integer | real ;
```

G-1.19 — G-1.26 (names):

```
keyword =
      "block" | "protected" | "public" | "end"
    | "record"
    | "function" | "method" | "signals" | "algorithm"
    | "input" | "output"
    | "Boolean" | "Integer" | "Real"
    | "limit"
    | "if" | "signal" | "in" | "then" | "elseif" | "else"
    | "for" | "loop"
    | "and" | "or" | "not" |
    | "size"
    | "self"
    (* reserved for future extensions: *)
    | "while" | "do" | "until"
    | "break" | "return"
    | "enumeration"
    | "__", identifier ;

alphabetic-character =
    (* ? any ISO/IEC 10646:2017 code point in ranges [65, 90] or [97, 122]: ? *)
      "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
    | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
    | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M"
    | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
;

identifier = ( alphabetic-character, { alphabetic-character | "_" | digit } ) - (
keyword ) ;

quoted-identifier =
    "'",
    (
      "previous", "(", scalarized-reference, ")"
    | "derivative", "(", quoted-identifier-higher-order-derivative, ")"
    | scalarized-reference
    ),
    "'" ;

quoted-identifier-higher-order-derivative =
      scalarized-reference
    | "derivative", "(", quoted-identifier-higher-order-derivative, ")" ;

scalarized-reference =
    ( identifier | keyword ),
    [ fixed-dimensions ],
    { ".", ( identifier | keyword ), [ fixed-dimensions ] } ;

fixed-dimensions = "[", positive-integer, { ",", positive-integer }, "]" ;

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:

```
state-reference =
    { white-space },
    "self",
    { white-space },
    ".",
    { white-space },
    name,
    { white-space },
    [ { white-space }, computed-dimensions, { white-space } ],
    { white-space },
    {
        { white-space },
        ".",
        { white-space },
        name,
        { white-space },
        [ { white-space }, computed-dimensions, { white-space } ],
        { white-space }
    },
    { white-space } ;
```

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

```
self.
a . b [2 ] // vector
    . 'c[2].d[3]'
    .
    m [
3 , /* matrix access */ 4
]
```

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

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

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

```
model M
    model MI
        model MII
            Real x;
            Boolean y;
        equation
            ...
        end MII;
        MII b[3];
    equation
        ...
    end MI;
    MI a[2];
equation
    ...
end M;
```

could be scalarized to

```
Real 'a[1].b[1].x';
Boolean 'a[1].b[1].y';
Real 'a[1].b[2].x';
Boolean 'a[1].b[2].y';
Real 'a[1].b[3].x';
Boolean 'a[1].b[3].y';
Real 'a[2].b[1].x';
Boolean 'a[2].b[1].y';
Real 'a[2].b[2].x';
Boolean 'a[2].b[2].y';
Real 'a[2].b[3].x';
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

```
Real 'a[1].b[1].x';
Boolean 'a[1].b[1].y';
Real 'a[1].b[3].x';
Boolean 'a[1].b[3].y';
Real 'a[2].b[1].x';
Boolean 'a[2].b[1].y';
Real 'a[2].b[3].x';
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):

```
block =
    "block",
    name,
    { state-entity-declaration } (* TODO: must be inputs, followed by outputs
followed by parameters *),
    "protected",
    { state-compartment-declaration },
    { state-entity-declaration },
    { error-signal-declaration },
    { function-declaration },
    "public",
    { function-declaration },
    "end",
    name,
    ";" ;

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

state-compartment-declaration =
    "record",
    name,
    { state-entity-declaration },
    "end",
    name,
    ";" ;

function-declaration =
    ( "function" | "method" ),
    name,
    [ signal-interface ],
    { parameter-declaration },
    [ "protected", { local-variable-declaration } ],
    "algorithm",
    { statement },
    "end",
    name,
    ";" ;

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

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

```
state-entity-declaration =
    [ "constant" | "parameter" ], (* TODO: Definition of terms and semantic of
constants and tuneable and dependent parameters *)
    variable-declaration ;

parameter-declaration = data-flow-direction, variable-declaration ;

local-variable-declaration = variable-declaration ;

data-flow-direction = "input" | "output" ;

variable-declaration =
    ( primitive-type | state-compartment-reference ),
    name,
    [ constant-dimensions ],
    ";" ;

primitive-type = "Boolean" | "Integer" | "Real" ;

state-compartment-reference = name ;

constant-dimensions =
    "[",
    ( derived-dimension | constant-scalar-integer-expression ),
    { ",", ( derived-dimension | constant-scalar-integer-expression ) },
    "]" ;

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:

```
record GearBox // Illegal: GearBox and gearBox not lexically-equivalent.
    Real w;
end gearBox;   // Illegal: GearBox and gearBox not lexically-equivalent.

method UpdateGearBox // Illegal: UpdateGearBox and 'UpdateGearBox' not
lexically-equivalent.
    input Real x;
algorithm
    self.gearBox.w := (x / self.gearBox.w) * self.gearBox.w;
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:

```
/*
    State compartment that is the control-state (cf. S-2.8).
    It has two local state entities, one variable and one component.
*/
block Controller
    record C
        Real    r;
        Integer i;
    end C;

    Integer i;        // State entity that is a state variable.
    C c;              // State entity that is a state component.

    function f
        output Real out_1[size(in, 1)]; // Parameter that is an output
parameter.
        input  Real in[:];              // Parameter that is an input
parameter.
        output Real out_2[size(in, 1)]; // Parameter that is an output
parameter.
    protected
        Integer s; // Local variable.
    algorithm
        s := 0;
        for i in 1:size(in, 1) loop
            s := s + in[i];
        end for;
        out_1 := in / s;
        out_2 := s * in;
    end f;
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:

```
block Controller2
    Real 'derivative(shaft[2].x)';   // Scalar named
'derivative(shaft[2].x)'.
    GearBox 'shaft[2].gear'[3];      // State component vector named
'shaft[2].gear'.
    Real w;                          // Scalar named w.

    method 'shaft[2].gear.update'            // Stateful function named
'shaft[2].gear.update'.
        input Real 'previous(shaft[2].y)';   // Scalar input parameter
named 'previous(shaft[2].y)'.
        input Integer index;                 // Scalar input parameter
named index.
    protected
        Real exp_y;                          // Scalar local variable named
exp_y.
    algorithm
        exp_y := exp('previous(shaft[2].y)');
        self.'shaft[2].gear'[index].w :=
        exp_y^2 - self.'derivative(shaft[2].x)' * exp_y;
    end 'shaft[2].gear.update';
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):

```
/*
    The single-line comments in this example are just abbreviations for
        // Illegal: Equally named <C>.
    where <C> is the comment and refers to the relative locations of
    equally named entities.
*/

record efmu // state compartment follows
end efmu;

record efmu // state compartment preceding
    C v; // state entity follows
    Real v; // state entity preceding
end efmu;

record C // function and local variable follow
end C;

method DoStep // function follows
protected
    Real v; // local variable follows
    Real v; // local variable preceding
algorithm
end DoStep;

method DoStep // function preceding
protected
    Integer f; // function follows
algorithm
end DoStep;

function C // state compartment preceding and local variable follows
    output Real r; // local variable follows
protected
    Boolean r[4]; // parameter preceding
algorithm
end C;

function f // local variable preceding
protected
    Integer C; // state compartment and function preceding
algorithm
end f;
```

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

```
block Controller3
    C C; // Type and name are lexically-equivalent.
    /*
        Name lexically-equivalent to self.C.r, parameter of
        function f and local variable of function DoStep:
    */
    Real r;

    record C
        /*
            Name lexically-equivalent to self.r, parameter of
            function f and local variable of function DoStep:
        */
        Real r;
        Boolean DoStep; // Name lexically-equivalent to function DoStep.
    end C;

    method DoStep
    protected
        /*
            Name lexically-equivalent to self.r, self.C.r and
            parameter of function f:
        */
        Real r;
    algorithm
    end DoStep;

    method Startup
    algorithm
    end Startup;

    function f
        /*
            Name lexically-equivalent to self.r, self.C.r and
            local variable of function DoStep:
        */
        output Real r;
    algorithm
    end f;
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.

```
block Controller
    record GearBox
        Real w;        // State variable of type Real.
    end GearBox;

    Boolean s;         // State variable of type Boolean.
    Real w[3];         // State variable of type Real.
    GearBox g[3];      // State component of type GearBox.

    function 'g.w.T_sum'
        input Integer T[3, 3]; // Input parameter of type Integer.
        output Real y;         // Output parameter of type Real;
    protected
        Real 'g.w'[3];         // Local variable of type Real;
    algorithm
        for i in 1:3 loop
            'g.w'[i] := emfu.g[i].w;
        end for;
        y := (if efmu.s then 1 else -1) * sum(real(T) * 'g.w');
    end 'g.w.T_sum';
end Controller;
```

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

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

```
record C1 // Illegal: Part of C1, C2, C3 cycle.
    C2 c;
end C1;

record C2 // Illegal: Part of C1, C2, C3 cycle.
    C3 c;
end C2;

record C3 // Illegal: Part of C1, C2, C3 cycle.
    C1 c;
end C3;

record C // Illegal: Non-efmu root.
    C2 c;
end C;

// Illegal: The control-state (efmu root) is missing.
```

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

```
record C1
end C1;

record C2
end C2;

/*
    Illegal: Control-state is not a root (C1 not reachable from
             efmu in state compartment composition graph):
*/
record efmu
    C2 c;
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***):

```
record C
    input Real i;  // Illegal: Control-input not local to the control-
state.
    output Real o; // Illegal: Control-output not local to the control-
state.
end C;

record efmu
    C c;
    input Real i_1;
    output Real o_1;
    input C i_2;    // Illegal: Control-input is a state component.
    output C o_2;   // Illegal: Control-output is a state component.
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 $N_{user\text{-}defined}$ 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 $n_{control\text{-}cycle}$ be the node labeled with the name of the control-cycle function: $\forall n_{user\text{-}defined} \in N_{user\text{-}defined}$: $n_{user\text{-}defined}$ *is reachable from* $n_{control\text{-}cycle}$).

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):

```
block S
    /*
        A 0-DIMENSIONAL state component, i.e.,
        a state component SCALAR:
    */
    C a;
    /*
        A 1-DIMENSIONAL state component, i.e.,
        a state component VECTOR:
    */
    C b[2];
     /*
        A 2-DIMENSIONAL state component, i.e.,
        a state component MATRIX
        with 2 ROWS and 3 COLUMNS:
    */
    C c[2,3];
    /*
        A 3-DIMENSIONAL state component, i.e.,
        a MULTI-DIMENSIONAL state component, i.e.,
        a state component MULTI-DIMENSION,
        that is neither, a VECTOR nor a MATRIX:
    */
    C d[1,1,1];
    Real r[3,3]; // a state variable MULTI-DIMENSION

    function f
        input  Real i[:,:]; // an input parameter MATRIX
        output Integer o;   // an output parameter SCALAR
    protected
        Integer j[size(i,1)]; // a MULTI-DIMENSIONAL local variable
        Real    k[size(i,2)]; // a VECTOR, i.e., a MULTI-DIMENSION
    algorithm
    end f;
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 $\forall p_i,p_j \in T_{input} ; i,j \in \mathbb{N}^+ ; i < j \leq n: p_i \in S_{input} \wedge p_j \in S_{input} \wedge p_i$ *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 $\forall q_i,q_j \in T_{output} ; i,j \in \mathbb{N}^+ ; i < j \leq m: q_i \in S_{output} \wedge q_j \in S_{output} \wedge q_i$ *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):

```
expression =
      constant
    | reference
    | dimension-query
    | function-call
    | parenthesized-expression
    | if-expression
    | multi-dimension-constructor
    | unary-operation
    | binary-operation ;

parenthesized-expression = "(", expression, ")" ;

dimension-query = "size", "(", reference, ",", constant-scalar-integer-
expression, ")" ;

constant-scalar-integer-expression = expression ;
```

G-3.5 — G-3.11 (operations):

```
unary-operation =
    unary-operator,
    (
          constant
        | reference
        | dimension-query
        | function-call
        | parenthesized-expression
        | if-expression
    ) ;

unary-operator = "-" | "not" ;

binary-operation = expression, binary-operator, expression ;

binary-operator = arithmetic-operator | relational-operator | logical-operator ;

arithmetic-operator = "+" | "-" | "*" | "/" | "^" ;

relational-operator = "<" | ">" | "<=" | ">=" | "==" | "<>" ;

logical-operator = "and" | "or" ;
```

G-3.12 — G-3.15 (multi-dimension constructors, function calls and conditional expressions):

```
multi-dimension-constructor =
    "{",
    multi-dimension-constructor-element,
    { ",", multi-dimension-constructor-element },
    "}" ;

multi-dimension-constructor-element = expression | multi-dimension-constructor ;

function-call = name, "(", [ expression, { ",", expression } ], ")" ;

if-expression =
    "(",
    "if",
    expression,
    "then",
    expression,
    { "elseif", expression, "then", expression },
    "else",
    expression,
    ")" ;
```

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 □ be lexically-equivalent to the operator $O_{□}$ of an operation O; we call O an □-operation and $O_{□}$ the □-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-3.11*** 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 three semantically equivalent expressions, each on a separate line. The first line shows a version of the expression requiring ***S-3.3*** for disambiguation. The second line shows a version not requiring ***S-3.3*** and that is minimal in the usage of parenthesis. The third line shows the expression with completely explicit evaluation order; it discloses the actual evaluation order by parenthesizing even expression parts whose evaluation order is already well-defined by syntactic rules only.

Expression 1:

```
 a + b  + c
(a + b) + c
(a + b) + c
```

Expression 2:

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

Expression 3:

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

Expression 4:

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

Expression 5:

```
3 -   -a ^  -b ^2
3 - ( -a ^( -b ^2))
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
(((((a < b) <> (c < d)) == (e < f)) and ((g == h) <> i)) or (((j + k) < (l
- m)) and n)) or o
(((((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 equivalent to `(a^2)*b`. 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 `a or b and c` is equivalent to `a or (b and c)`. For example, `(a or b) and c` yields *false* if `a` and `b` are *true* and `c` is *false* whereas `a or (b and c)` yields *true*.

The value assigned to `a` in `a := -b^2;` always will be positive whereas for `a := 0 - b^2;`, `a := -1*b^2;` and `a := -(b^2);` it always will be negative.

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:

```
b1 := r1 > r2;
b2 := r1 < r2;
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). 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. 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 $\alpha \in \mathbb{N}^+$.

Expression 1:

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

Expression 1:

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

Expression 1:

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

Expression 1:

```
m_1() // Illegal: Sibling stateful function-call.
 +
m_2() // Illegal: Sibling stateful function-call.
```

Expression 2:

```
(if 0 < m_1() // Illegal: Sibling state variable reference.
    then f_1(self.v_1)
    else 1.0)
```

Expression 3:

```
(if f_1(self.v_1)
    then m_1() // Illegal: Within if branch.
    else self.v_1)
```

Expression 3:

```
(
if
// Parallelizable: Part of separately-evaluable sub-expression-set α:
    f_1(A)
then
// Parallelizable: Part of separately-evaluable sub-expression-set α:
    f_2(A * B) * C
else
// Parallelizable: Part of separately-evaluable sub-expression-set α:
    f_2(A - B) * C
)

    + // NOT parallelizable (cf. S-3.3:L-1).

// Parallelizable: Part of separately-evaluable sub-expression-set α:
f_3(
    D / (E - F),  // Parallelizable: Part of separately-evaluable sub-
expression-set β.
    E * F,        // Parallelizable: Part of separately-evaluable sub-
expression-set β.
    (
        f_2(E) // Parallelizable: Part of separately-evaluable sub-
expression-set γ.
         *        // Parallelizable: Part of separately-evaluable sub-
expression-set β.
        f_2(F) // Parallelizable: Part of separately-evaluable sub-
expression-set γ.
    )
)

    + // NOT parallelizable (cf. S-3.3:L-1).

// Parallelizable: Part of separately-evaluable sub-expression-set α:
(

)

    + // NOT parallelizable (cf. S-3.3:L-1).

// Parallelizable: Part of separately-evaluable sub-expression-set α:
(
    A^3
)
```

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

```
function f_1
    input Real x_1;
    input Real x_2;
    input Real x_3;
    output Real y;
algorithm
    y := x_3 * (x_1 * self.a + x_2 * self.b);
end f_1;

method m_1
    output Real y;
algorithm
    self.a := self.a + 1;
    self.b := self.a;
    y := self.a;
end m_1;

method m_2
    output Real y;
algorithm
    self.b := 2 * self.b;
    self.a := self.b;
    y := self.b;
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)

```
(* references *)
reference = local-reference | state-reference ;

local-reference = name, [ computed-dimensions ] ;

state-reference =
    "self",
    ".",
    name,
    [ computed-dimensions ],
    { ".", name, [ computed-dimensions ] } ;

computed-dimensions =
    "[",
    constant-scalar-integer-expression,
    { ",", constant-scalar-integer-expression },
    "]" ;

(* statements *)
statement =
    (
          limit-statement
        | function-call
        | single-assignment
        | multi-assignment
        | if-statement
        | for-loop
    ),
    ";" ;

limit-statement =
    "limit",
    ( "self" | reference ),
    { ",", ( "self" | reference ) } ;

single-assignment = reference, ":=", expression ;

multi-assignment =
    "(",
    [ reference, { ",", reference } ],
    ")",
    ":=",
    function-call ;

if-statement =
    "if",
    ( expression | error-signal-check ),
    "then",
```

```
    { statement },
    { "elseif", ( expression | error-signal-check ), "then", { statement } },
    [ "else", { statement } ],
    "end",
    "if" ;

error-signal-check =
    "signal",
    [ identifier ],
    [
        [ "not" ],
        "in",
        identifier,
        { ",", identifier  }
    ],
    [ "or", expression ] ;

for-loop = "for", bounded-iteration, "loop", { statement }, "end", "for" ;

bounded-iteration =
    [ loop-iterator-declaration, "in" ],
    start-bound,
    [ ":", iteration-step-size ],
    ":",
    termination-bound ;

loop-iterator-declaration = name ;

start-bound = constant-scalar-integer-expression ;

iteration-step-size = constant-scalar-integer-expression ;

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.

4.2.5. 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.

Error Signals

Error-signal-declaration **semantic**

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

```
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

```
signal INVALID_ARGUMENT;
signal OVERFLOW;
signal NAN;
signal SOLVE_LINEAR_EQUATIONS_FAILED;
signal NO_SOLUTION_FOUND;
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.

Note: Above specification implies that pre- and user-defined error signals *are* error signals and can therefore be explicitly signaled and checked by user-code.

Note: The intended usage of the pre-defined error signals is:

- `INVALID_ARGUMENT`: Unspecified error in one or more input arguments.
- `OVERFLOW`: Computed floating point result is -∞ or +∞.
- `NAN`: Computed floating point result 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.

Error-signal-statement **semantic**

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

```
error-signal-statement =
    "signal",
    identifier,           (* Set of signals set, at least one AND/OR signal-closure
propagation *)
    { ",", 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.
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*.

Functional error interface and exposed error signals

A *function-declaration F* of the form

```
function-declaration =
    ( "function" | "method" ),
    name,
    [ signal-interface ], (* 3rd child defining the signal-set -- i.e, exposed error
signals -- of the function *)
    { parameter-declaration },
    [ "protected", { local-variable-declaration } ],
    "algorithm",
    { statement },
    "end",
    name,
    ";" ;
```

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.

Error-signal-check semantic

An *error-signal-check* of the form

```
error-signal-check =
    "signal",
    [ identifier ],           (* Optional signal-closure *)
    [
        [ "not" ],            (* Optional signal-test-negation *)
        "in",
        identifier,           (* Set of signals tested, at least one *)
        { ",", identifier  }, (* Set of signals tested, at least one *)
    ],
    [ "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 the body of the respective `if`/`elseif` conditional — the error-signal-check-body — similar to loop-iterators (cf. *loop-iterator-declaration*, issue #49). It must never be assigned to.
2. We define the signal-test-set of an *error-signal-check* as follows:
   - **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.

     ```
     In any case, the signal-test-set must be non-empty and a subset of the in-
     reachable-signals-set of the _error-signal-check_.
     ```
3. 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.
4. 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.
5. An *error-signal-check* is satisfied if it is signal-satisfied or conditional-satisfied.
6. The error-signal-check-body *B* of an *error-signal-check* is the executed branch of its *if-statement*, if, and only if, it is satisfied. In this case, all signals of the signal-test-set are unset immediately before the execution of *B* but after initializing the signal-closure if any.

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

The idea is simple: To statically decide which error-signals could be set at any point of execution, we define a data-flow analysis, whereas the propagated data is a set of error signals — the reachable-signals-set; this set in turn can then be used to enforce that *error-checks* only check for error-signals that can be set according to their preceding control-flow and functions only expose signals that can be signaled but are not checked thereafter for any of their possible control-flows.

We define signal-sets for expressions and statements (a signal-set defines which additional signals

can be set by the respective language construct):

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 bodies.

We define reachable-signals-sets for statements and the branches of *if-statements*, particularly *error-signal-check* branches. Thereby we distinguish between the signals that can be set before executing the respective construct (in-reachable-signals-set) and the ones that can be set after its execution finished (out-reachable-signals-set):

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 of an *if-statement S* is the in-reachable-signals-set of *S*; for any further branch of *S* it is the out-reachable-signals-set of its preceding branch.
3. The in-reachable-signals-set of the body of a branch *B* of an *if-statement* is the out-reachable-signals-set of *B*.
4. The in-reachable-signals-set of any other statement *S* of a function-body 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 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 is its in-reachable-signals-set.
6. The out-reachable-signals-set of an *if-statement* is the out-reachable-signals-set of its last branch 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.

Production Code and exposing errors to the runtime environment

Since block-interface methods can only expose the 6 pre-defined error signals (cf. Section *"Semantic: A"*), 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).

Examples

Example 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:

```
/*
    Safe common control-code, potentially selecting or deselecting special
    modes of operation:
*/
...
v := f(A); // f may signal the error f_ERROR.
...
if signal in f_ERROR or not(check(v)) then
    /*
        Error-handling path if f(A) signaled an f_ERROR or
        returned a v not satisfying some check:
    */
    ...
elseif self.operation_mode == 1 then
    /*
        Safe control-code for some special operation mode:
    */
    ...
elseif self.operation_mode == 2 then
    /*
        Safe control-code for some special operation mode:
    */
    ...
else
    /*
        Control-code for normal mode of operations:
    */
    ...
    x := solveLinearEquations(A, b * v);
    ...
    if signal in SOLVE_LINEAR_EQUATIONS_FAILED then
        /*
            Handle the special case that the system of linear equations
```

```
            has no solution:
        */
        ...
    elseif signal then
        /*
            Handle any other unexpected error of the NORMAL operation mode:
        */
        ...
    end if;
end if;

if signal s then
    /*
        The common control-code or the special modes of operation that are
        supposed to be safe missed some error case or introduced
        errors themselves. We now can set the control-outputs and state
        variables to some reasonable default values and propagate the
        unexpected error signals to the runtime environment:
    */
    ...
    signal s;
end if;
```

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

```
method DoStep
/*
    (1) Signal interface of functions (signals exposed to callees):
*/
signals invalid_gear_switch, to_high_velocity;
algorithm
    ...
    /*
        (2) Universal signal checks, catching and un-setting all signals set:
    */
    if signal then
        ...
    end if;
    ...
    /*
        (3) Specialized signal checks, catching and un-setting
            all signals within a specific set:
    */
    if signal in error1, error2 then
        ...
    end if;
    ...
    /*
        (4) Restricted universal signal checks, catching any signal that is
```

```
        not within a certain set:
*/
if signal not in invalid_gear_switch, to_high_velocity then
   ...
end if;
...
/*
    (3) Checks with signal variables enclosing the checked signals
        that have been set at the check point:
*/
if signal s then
    ...
    /*
        (4) Propagation of signal variable, i.e., set all the signals that the
            check s is part of unset:
    */
    signal s;
    ...
else
    ...
end if;
...
if ... then
    ...
    /*
        Explicit setting of signals, i.e., signaling of errors:
    */
    signal invalid_gear_switch, to_high_velocity;
    ...
end if;
...
/*
    (*) And all kind of combinations of the above
        (signals to check with signal variables, signal
        propagation and explicit signaling):
*/
if signal s in f1_error, f2_error or condition1 then
    ...
    signal s, invalid_gear_switch;
    ...
elseif signal s not in invalid_gear_switch, to_high_velocity or condition2 then
    ...
    signal s;
    ...
end if;
...
/*
    Catch all signals not exposed according to the function's interface:
*/
if signal not in invalid_gear_switch, to_high_velocity then
end if;
```

```
end DoStep;
```

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

```
function f
    signals Error1; // Violates B.2: Error1 never exposed and Error2 is missing.
    input Real i;
    output Real o;
protected
algorithm
    if i > 100.0 then
        signal Error1;
    elseif i > 200.0 then
        signal Error2;
    end if;
    o := 2.0 * i;
    if signal in Error1 or o > 350.0 then
        o := 350.0;
    end if;
end f;

method DoStep // Violates B.2: Error2 is exposed but 'signal in Error2;' is missing.
protected
algorithm
    ...
    f(1.0);
    ...
    if signal s then
        ...
        s := Error1; /* Violates C.1: Signal-closures must not be assigned to. */
    elseif signal in Error1 then
        /*
            Above error-signal-check violates C.2: Signal-test-sets must be non-empty.
            Note, that the preceding branch already handles all error signals since it
            is an unrestricted error-signal-check.
        */
    end if;

    signal Error1;

    if signal in Error1, Error2 then
        /*
            Above error-signal-check violates C.2: The signal test-set is not a subset
of the
            in-reachable-signals-set since Error2 can never be set at this point.
        */
        ...
        signal Error2;
    elseif signal in Error2 then
```

```
        /*
            Above error-signal-check violates C.2: The signal test-set is not a subset
of the
            in-reachable-signals-set since Error2 can never be set at this point.
Note, that the
            signal-set of the error-signal-check-body of the preceding branch cannot
be handled
            by this branch; it requires handling in a completely separate if-
statement.
        */
    end if;

    signal Error1;
    if signal in Error1 then
    end if;
    if signal in Error1 then
        /*
            Above error-signal-check violates C.2: Signal-test-sets must be non-empty.
            The preceding if-statement already implicitly unsets Error1 when its
            single error-signal-check is satisfied.
        */
    end if;
end DoStep;

method Startup
protected
algorithm
    // signal in Error1; /* The following if-statement is erroneous, even if this line
is uncommented. */
    if signal not in Error1 then
        /* Above error-signal-check violates C.2: Signal-test-sets must be non-empty.
    end if;
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.

```
function f
    signals f_Error;
    output Boolean b;
protected
algorithm
    b := true;
    signal f_Error;
end f;

method DoStep
    ...
```

```
algorithm
    ...
    if signal then // Unset all error signals.
    end if;
    signal in TestDefinitions1, TestDefinitions2;
    if signal in TestDefinitions1 then
        ...
        signal TestDefinitions3;
        ...
    elseif signal in TestDefinitions2 then
        ...
        if signal TestDefinitions3 then
            ...
        end if;
        ...
    elseif signal in TestDefinitions3 then
        ...
    end if;
    /*
        At this point still TestDefinitions2 and TestDefinitions3 WILL be
        set because only the first branch was tested, its test signal-satisfied,
        the tested signal TestDefinitions1 unset and its body executed.
    */
    ...
    if signal then // Unset all error signals.
    end if;
    signal TestDefinitions1, TestDefinitions2;
    if signal in TestDefinitions1 then
        ...
        if signal in TestDefinitions2 then
            ...
        end if;
        ...
    end if;
    // At this point no error signals WILL be set.
    ...
    /*
        Assume for the following code an execution where NotSetSignal
        is not set:
    */
    if signal not in NotSetSignal then // Unset all error signals except NotSetSignal.
    end if;
    i := 2;
    if signal in NotSetSignal or f() /* Cf. definition of f above! */ then
        i := 2 * i;
    elseif signal in f_Error then
        i := 2 * i;
        signal f_Error;
    /*
    The following branch would be invalid, because f_Error can never be set when it is
tested:
```

```
    elseif signal in f_Error then
        i := 2 * i;
    */
    end if;
    // At this point i WILL be 8 and f_Error set.
end DoStep;
```

-∞, +∞ 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 754-2008, it should still be possible to map GALEC code to such a target, since only a small subset of IEEE 754 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 754-2008. 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 `log(-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:

```
function f_□
   signals NAN;
   input Real a;
   input Real b;
   output Boolean y;
algorithm
   if isNaN(a) or isNaN(b) then
      signal NAN;
      y := false;
   else
      y := a □ b;
   end if;
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 4.2.6) 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 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).

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 □ and upper bound □, then the statement `limit v;` is equivalent to `v := (if v < □ then □ elseif v > □ then □ else v);`. If `v` has only a lower bound □, `limit v` is equivalent to `v := (if v < □ then □ else v);`. If `v` has only an upper bound □, `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):

```
limit self; // Limits all ranged state variables.
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,

```
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:

```
method Startup
protected
    ...
algorithm
    ...
    // initialize stuff
    ...
    limit self; // Implicit by semantic of language.
end Startup;

method DoStep
protected
   ...
algorithm
   limit self; // Implicit by semantic of language.
   ...
   // compute stuff
   ...
   limit self; // Implicit by semantic of language.
end DoStep;

method Recalibrate
protected
   ...
algorithm
   limit self; // Implicit by semantic of language.
   ...
   // compute stuff
   ...
   limit self; // Implicit by semantic of language.
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.]

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`.

4.2.6. Built-in Functions

In this section the built-in functions are defined. If the built-in function is also defined in IEEE 754, 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`).

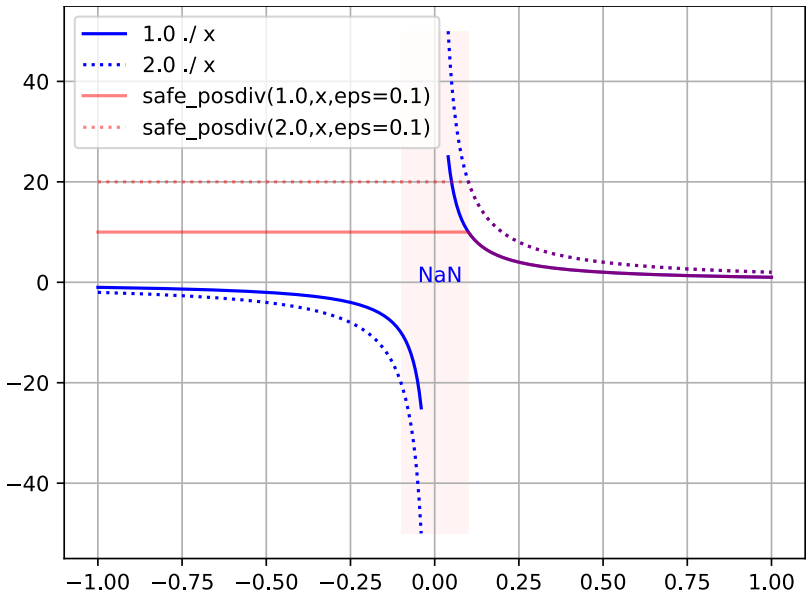
A built-in function only returns an error signal if explicitly mentioned in its definition below; most builtin functions do not signal any errors and instead rely on `qNaN` propagation.

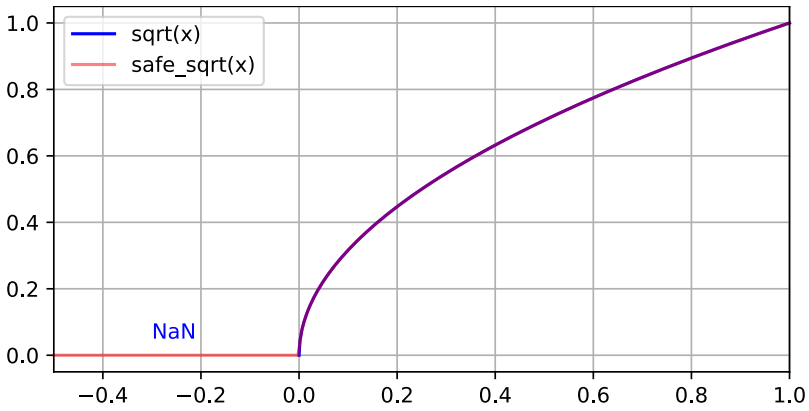
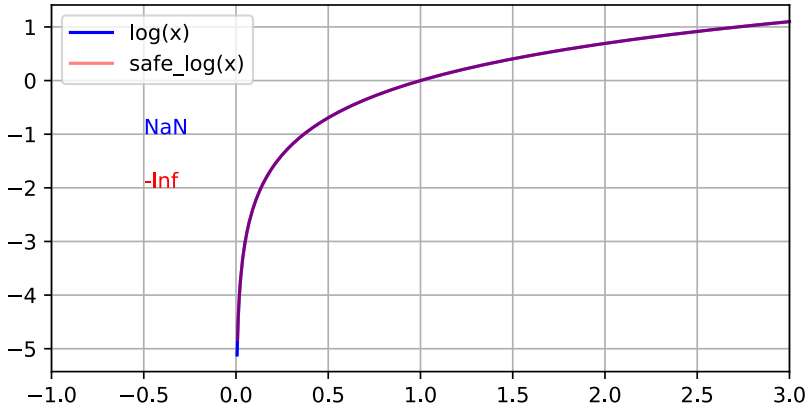
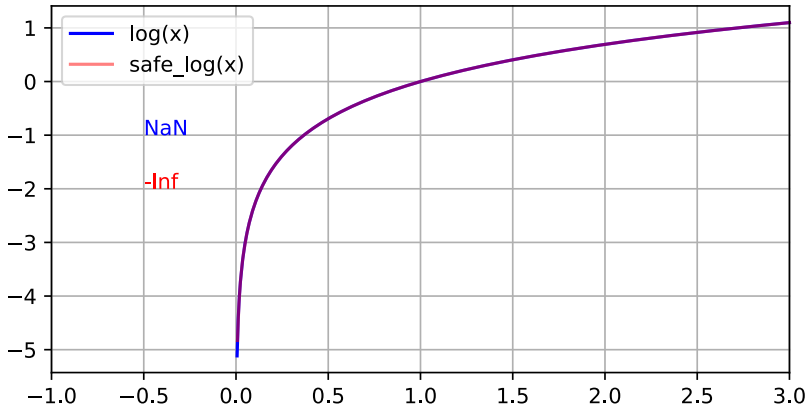
Overview

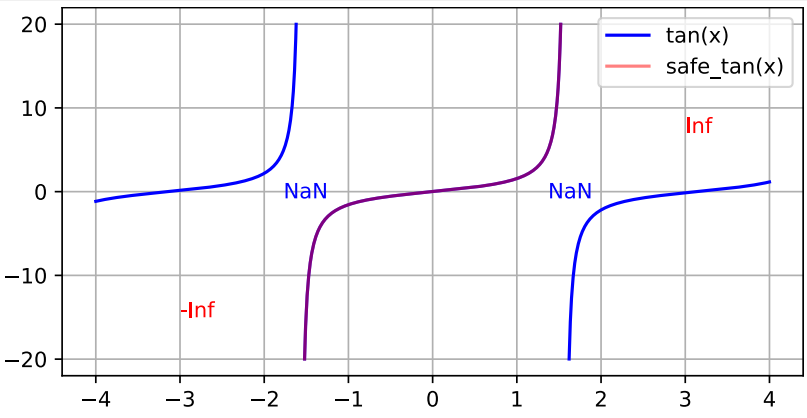
In the following table, an overview of the built-in functions is given (the follow-up sub-section contains the precise definition of the built-in functions):

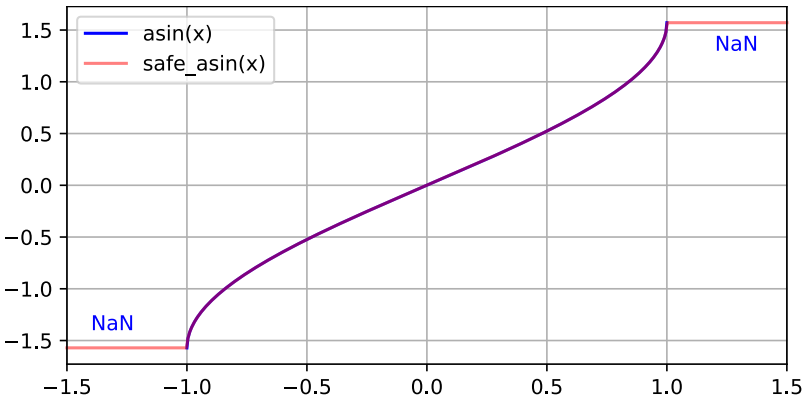
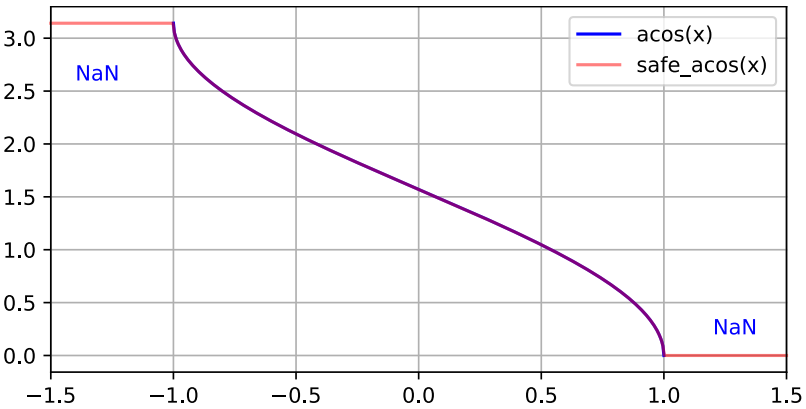
Function-Name	Description
Properties of Integer	
`minInteger()`	Target-specific smallest Integer.
`maxInteger()`	Target-specific largest Integer.
Properties of Real	
`minReal()`	Target-specific smallest Real `r <> minusInfinite()`.
`maxReal()`	Target-specific largest Real `r <> plusInfinite()`.
`r := 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	

Function-Name	Description
`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.
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	
`realRemainderTowardsZero(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	

Function-Name	Description
`euler()`	Target-specific, most-precise representation of Euler's number (= 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`.
`log10(x)`	Logarithm of `x` to base 10. Returns `qNaN` if `x` is `qNaN` or `x < 0.0`.
`safe_posdiv(xn, xd, eps)`	`qNaN`-free division of `xn` by `xd` if `eps > 0.0`: `xn / (if xd >= eps then xd else eps)`. Returns `qNaN` if `xn` or `xd` is `qNaN` or if `eps == 0.0 and xn == 0.0 and xd == 0.0`.
	

Function-Name	Description
`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_log(x)`	`qNaN`-free natural logarithm of x: `log(if x >= 0.0 then x else 0.0)`. Returns `qNaN` if `x` is `qNaN`.
	
`safe_log10(x)`	`qNaN`-free logarithm to base 10 of x: `log10(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 +∞.

Function-Name	Description
`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`.
`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`. 

Function-Name	Description
`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`.
`(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.

Function-Name	Description
`interpolation2D(x1, x2, x1_data, nx1, nx2_data, nx2, y_data, ipo, expo)`	Constant/linear interpolation in 2D with extrapolation.
`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 $C_{builtin} = C_{builtin1} \oplus C_{builtin2} \oplus C_{builtin3} \oplus C_{builtin4}$ where each $C_{builtinn}$ with $n \in \{1,2,...,4\}$ is a sequence of characters defined in the following and $\oplus$ 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:

```
/*
    Note: We distinguish integer and Integer. Integer with uppercase first letter
is the type
          Integer -- a target-specific data-type -- whereas integer with
lowercase first
          letter is the mathematic term for numbers without fractional component.
Likewise,
          we distinguish real and Real.
*/

/*******************************************************************************
*************
    Properties of Integer:
*******************************************************************************
************/

function minInteger
    output Integer i;
algorithm /*
    i := target-specific smallest Integer;
*/ end minInteger;

function maxInteger
    output Integer i;
algorithm /*
    i := target-specific largest Integer;
*/ end maxInteger;

/*******************************************************************************
*************
    Properties of Real:
*******************************************************************************
************/

function minReal
    outputs Real r;
```

```
algorithm /*
    r := target-specific smallest, not -∞ representing, Real;
*/ end minReal;

function maxReal
    outputs Real r;
algorithm /*
    r := target-specific largest, not +∞ representing, Real;
*/ end maxReal;

function posMinReal
    output Real r;
algorithm /*
    r := target-specific smallest Real > 0.0;
*/ end posMinReal;

function epsReal
    output Real r;
algorithm /*
    r := target-specific largest Real r > 0.0 such that 1.0 + r == 1.0;
*/ end epsReal;

function nan
    output Real r;
algorithm /*
    r := target-specific not-a-number representation;
*/ end nan;

function isNaN
    input  Real x;
    output Boolean b;
algorithm /*
    b := true if x is target-specific not-a-number representation, false
otherwise;
*/ end isNaN;

function minusInfinite
    output Real r;
algorithm /*
    r := target-specific -∞ representation;
*/ end minusInfinite;

function plusInfinite
    output Real r;
algorithm /*
    r := target-specific +∞ representation;
*/ end minusInfinite;

function isInfinite
    input  Real x;
    output Boolean b;
```

```
algorithm /*
    b := x == minusInfinite() or x == plusInfinite();
    if signal in NAN then
        b := false;
    end if;
*/ end isInfinite;

function isFinite
    input  Real x;
    output Boolean b;
algorithm /*
    b := not(isNaN(x)) and not(isInfinite(x));
*/ end isFinite;

/*****************************************************************************
*************
    Multi-dimensional properties of Real:
*****************************************************************************
************/

function hasNaN1D
    input  Real x[:];
    output Boolean result;
algorithm /*
    result := false;
    for i in 1:size(x, 1)
        if isNaN(x[i])
            result := true;
        end if;
    end for;
*/ end hasNaN1D;

function hasNaN2D
    input  Real x[:, :];
    output Boolean result;
algorithm /*
    result := false;
    for i in 1:size(x, 1)
        for j in 1:size(x, 2)
            if isNaN(x[i, j])
                result := true;
            end if;
        end for;
    end for;
*/ end hasNaN2D;

/*****************************************************************************
*************
    Numeric type conversions:
*****************************************************************************
************/
```

```
function real
    input  Integer i;
    output Real r;
algorithm /*
    r := target-specific Real representation of i;
*/ end real;

function integer
    signals NAN, OVERFLOW;
    input Real r;
    output Integer i;
protected
    Real tmp;
algorithm /*
    i := 0;
    tmp := roundTowardsZero(r); // Returns qNaN if r is qNaN.
    if tmp < real(minInteger()) or tmp > real(maxInteger()) then
        signal OVERFLOW;
    elseif signal in NAN then
        signal NAN; // tmp was qNaN.
    else
        i := target-specific Integer representation of tmp;
    end if;
*/ end integer;

/*****************************************************************************
*************
    Direct Real rounding:
*****************************************************************************
************/

function roundDown
    input Real r;
    output Real i;
algorithm /*
    // Also known as: flooring, round towards -∞.
    if isNaN(r) then
        i := nan();
    else
        i := target-specific greatest integer ≤ r;
    end if;
*/ end roundDown;

function roundUp
    input Real r;
    output Real i;
algorithm /*
    // Also known as: ceiling, round towards +∞.
    if isNaN(r)
        i := nan();
```

```
    else
        i := target-specific least integer >= r;
    end if;
*/ end roundUp;

/*****************************************************************************
*************
    Nearest Real rounding (using a tie-breaking rule):
*****************************************************************************
************/

function roundHalfToEven
    input Real r;
    output Real i;
algorithm /*
    // Also known as: convergent rounding, statistician's rounding, Dutch
rounding,
    //   Gaussian rounding, odd–even rounding, bankers' rounding.
    i := (if roundHalfDown(r) < roundHalfUp(r)
        then (if (r + 0.5 is even) then r + 0.5 else r - 0.5)
        else roundHalfDown(r));
    if signal in NAN or isNaN(r) then
        i := nan();
    end if;
*/ end roundHalfToEven;

/*****************************************************************************
*************
    Relational Integer functions:
*****************************************************************************
************/

function imin
    input  Integer u1;
    input  Integer u2;
    output Integer y;
algorithm /*
    y := (if u1 < u2 then u1 else u2);
*/ end imin;

function imax
    input  Integer u1;
    input  Integer u2;
    output Integer y;
algorithm /*
    y := (if u1 > u2 then u1 else u2);
*/ end imax;

/*****************************************************************************
*************
    Relational Real functions:
```

```
**********************************************************************************
************/

function min
    input  Real u1;
    input  Real u2;
    output Real y;
algorithm /*
    y := (if u1 < u2 then u1 else u2);
    if signal in NAN then
        y := nan();
    end if;
*/ end min;

function max
    input  Real u1;
    input  Real u2;
    output Real y;
algorithm /*
    y := (if u1 > u2 then u1 else u2);
    if signal in NAN then
        y := nan();
    end if;
*/ end max;

/**********************************************************************************
*************
    Mathematical Real constants and functions:
**********************************************************************************
************/

function euler
    output Real r;
algorithm /*
    r := target-specific, most-precise representation of □;
*/ end euler;

function sign
    input  Real r;
    output Real i;
algorithm /*
    i := (if r > 0.0 then 1.0 elseif r < 0.0 then -1.0 else 0.0);
    if signal in NAN then
        i := nan();
    end if;
*/ end sign;

function fractional
    input  Real x;
    output Real y;
algorithm /*
```

```
    y := x - roundTowardsZero(x);
*/ end fractional;

function absolute
    input  Real x;
    output Real y;
algorithm /*
    y := sign(x) * x;
*/ end absolute;

function sqrt
    input  Real x;
    output Real y;
algorithm /*
    if x < 0.0 then
        y := nan();
    elseif signal in NAN then
        y := nan();
    else
        y := x^0.5;
    end if;
*/ end sqrt;

function exp
    input  Real x;
    output Real y;
algorithm /*
    y := euler()^x;
*/ end exp;

function ln
    input  Real x;
    output Real y;
algorithm /*
    if x < 0.0 then
        y := nan();
    elseif signal in NAN then
        y := nan();
    else
        y := natural logarithm of x;
    end if;
*/ end ln;

function log10
    input  Real x;
    output Real y;
algorithm /*
    if x < 0.0 then
        y := nan();
    elseif signal in NAN then
        y := nan();
```

```
    else
        y := logarithm to base 10 of x;
    end if;
*/ end log10;

function safe_posdiv
    input  Real xn;
    input  Real xd;
    input  Real eps(min = posMinReal());
    output Real y;
algorithm /*
    y := xn / (if xd >= eps then xd else eps);
*/ end isinf;

function safe_sqrt
    input  Real x;
    output Real y;
algorithm /*
    y := sqrt(if x < 0.0 then 0.0 else x);
*/ end safe_sqrt;

function safe_log
    input  Real x;
    output Real y;
algorithm /*
    y = log(if x < 0.0 then 0.0 else x);
*/ end safe_log;

function safe_log10
    signals NAN;
    input  Real x;
    output Real y;
algorithm /*
    y = log10(if x < 0.0 then 0.0 else x)
*/ end safe_log10;

/*******************************************************************************
*************
    Trigonometric Real constants and functions:
*******************************************************************************
************/

function pi
    output Real r;
algorithm /*
    r := target-specific, most-precise representation of π;
*/ end pi;

function sin
    input Real x;
    output Real y;
```

```
algorithm /*
    if not(isFinite(x)) then
        y := nan();
    else
        y := sine of x;
    end if;
*/ end sin;

function cos
    input Real x;
    output Real y;
algorithm /*
    if not(isFinite(x)) then
        y := nan();
    else
        y := cosine of x;
    end if;
*/ end cos;

function tan
    input Real x;
    output Real y;
algorithm /*
    if not(isFinite(x)) then
        y := nan();
    else
        y := sin(x) / cos(x);
    end if;
    if isInfinite(y) then
        y := nan();
    end if;
*/ end tan;

function asin
    input  Real x;
    output Real y;
algorithm /*
    if -1.0 <= x and x <= 1.0 then
        y := inverse of sin(x) in the range -π/2 □ y □ π/2;
    elseif signal in NAN or true then
        y := nan();
    end if;
*/ end asin;

function acos
    input  Real x;
    output Real y;
algorithm /*
    if -1.0 <= x and x <= 1.0 then
        y := inverse of cos(x) in the range 0 □ y □ π;
    elseif signal in NAN or true then
```

```
        y := nan();
    end if;
*/ end asin;

function atan
    input Real x;
    output Real y;
algorithm /*
    if isNaN(x)) then
        y := nan();
    elseif isInfinite(x) then
        y := sign(x) * pi() / 2.0;
    else
        y := inverse of tan(x) in the range -π/2 < y < π/2;
    end if;
*/ end atan;

function atan2
    input  Real y;
    input  Real x;
    output Real z;
algorithm /*
    z := (if     x > 0.0              then atan(y / x)
          elseif x < 0.0 and y >= 0.0 then atan(y / x) + pi()
          elseif x < 0.0 and y <  0.0 then atan(y / x) - pi()
          elseif y > 0.0              then  pi() / 2.0
          elseif y < 0.0              then -pi() / 2.0
          else                             nan());
    if signal in NAN then
        z := nan();
    end if;
*/ end atan2;

function sinh
    input Real x;
    output Real y;
algorithm /*
    y := (euler()^x - euler()^-x) / 2.0;
*/ end sinh;

function cosh
    input Real x;
    output Real y;
algorithm /*
    y := (euler()^x + euler()^-x) / 2.0;
*/ end cosh;

function tanh
    input Real x;
    output Real y;
algorithm /*
```

```
    y := sinh(x) / cosh(x);
*/ end tanh;

function safe_tan
    signals NAN;
    input  Real x;
    output Real y;
algorithm /*
    y := (if x >= pi() / 2.0 then plusInfinite()
        elseif x <= -pi() / 2.0 then minusInfinite()
        else tan(x));
    if signal in NAN then
        signal NAN;
        y := nan();
    end if;
*/ end safe_tan;

function safe_asin
    signals NAN;
    input  Real x;
    output Real y;
algorithm /*
    y := asin(if x > 1.0 then 1.0 elseif x < -1.0 then -1.0 else x)
*/ end safe_asin;

function safe_acos
    input  Real x;
    output Real y;
algorithm /*
    y := acos(if x > 1.0 then 1.0 elseif x < -1.0 then -1.0 else x)
*/ end safe_acos;

/*****************************************************************************
*************
    Systems of linear equations:
*****************************************************************************
************/

function solveLinearEquations
    signals SOLVE_LINEAR_EQUATIONS_FAILED;
    input  Real A[:, size(A,1)];
    input  Real b[size(A,1)];
    output Real x[size(A,1)];
algorithm /*
    Solve system of linear equations A*x = b for x. Hereby it is assumed that
matrix A is
    regular. Typically, the function implements a direct Gaussian elimination
with partial
    pivoting. If A is singular, SOLVE_LINEAR_EQUATIONS_FAILED is signaled
    and at least on element of x is set to qNaN.
*/ end solveLinearEquations;
```

```
function luFactorize
    signals SOLVE_LINEAR_EQUATIONS_FAILED;
    input  Real    A[:, size(A, 1)];
    output Real    LU[:, size(A, 1)];
    output Integer pivots[size(A, 1)];
    /*
    The function returns the LU decomposition with partial pivoting of the
square,
    matrix A: P*L*U = A where P is the permutation matrix (implicitely defined by
vector
    pivots), L is a lower triangular matrix with unit diagonal elements and U is
an upper
    triangular matrix. Matrices L and U are stored in matrix LU on return (the
diagonal of
    L is not stored). With the companion function luSolve, the factorization is
used to
    solve the linear system L*U*x = b[pivots] with different right hand side
vectors b.

    If A is singular, SOLVE_LINEAR_EQUATIONS_FAILED is signaled.

    The algorithm below is "conceptual". A more efficient implementation uses
    BLAS functions, see, e.g., LAPACK function DGETRF.
    */
protected
    Integer n;
    Integer p; // Pivot index.
    Integer pk;
    Real temp;
    Real eta;
    Real d;
    Real d_max;
    Real di;
    Real di_abs;
algorithm
    n := size(A,1);
    LU := A;
    p := 1:n;
    if n < 1 then
       return;
    end if;

    for k in 1:n-1 loop
       // Find pivot
       p :=k;
       d := LU[k,k];
       d_max :=absolute(d);
       for i in k+1:n loop
          di := LU[i,k];
          di_abs := abs(di);
```

```
         if di_abs > d_max then
            p     := i;
            d     := di;
            d_max := di_abs;
         end if;
      end for;

      // Test pivot for singularity
      if d == 0 then
         signals SOLVE_LINEAR_EQUATIONS_FAILED;
      else
         // Swap LU[k,j] and LU[p,j], for j = 1,....,n
         // as well as pivots[k] and pivots[p]
         if k <> p then
            for j in 1:n loop
               temp :=LU[k, j];
               LU[k,j] :=LU[p, j];
               LU[p,j] :=temp;
            end for;
            pk :=pivots[k];
            pivots[k] :=pivots[p];
            pivots[p] :=pk;
         end if;

         // LU factors
         for i in k+1:n loop
            eta :=LU[i,k]/d;
            LU[i,k] :=eta;

            for j in k+1:n loop
               LU[i,j] :=LU[i,j] - eta*LU[k,j];
            end for;
         end for;
      end if;
   end for;
end luFactorize;

function luSolve
   signals SOLVE_LINEAR_EQUATIONS_FAILED;
   input  Real    LU[:, size(LU, 1)];  // Returned from luFactorize.
   input  Integer pivots[size(LU, 1)]; // Returned from luFactorize.
   input  Real    b[size(LU, 1)];
   output Real    x[size(LU, 1)];
   /*
   The function returns the solution x of the linear system of equations:
       L*U*x = b[pivots]
   where L*U and pivots are computed by the companion function luFactorize.
   If a unique solution cannot be computed (i.e., U is singular),
   SOLVE_LINEAR_EQUATIONS_FAILED is signaled and at least one element of x is
qNaN.
```

```
    The algorithm below is "conceptual". A more efficient implementation uses
    BLAS functions, see, e.g., LAPACK function DGETRS.
    */
protected
    Integer n=size(LU,1);
    Real y[size(LU,1)];
algorithm
    if n < 1 then
       return;
    end if;

    // Forward elimination
    for i in 1:n loop
       y[i] := b[pivots[i]];
       for j in 1:i-1 loop
          y[i] :=y[i] - LU[i, j]*y[j];
       end for;
    end for;

    // Backward substitution
    for i in n:-1:1 loop
       x[i] :=y[i];
       for j in i+1:n loop
          x[i] := x[i] - LU[i,j]*x[j];
       end for;
       x[i] := x[i]/LU[i,i];
       if isNaN(x[i])
        signals SOLVE_LINEAR_EQUATIONS_FAILED;
       end
    end for;
end luSolve;


/*************************************************************************
*************
    Interpolation in 1D/2D/3D:

    In all functions the following options are used:
    - interpolation = 1: constant bottom interpolation
                    = 2: linear interpolation
    - extrapolation = 1: hold last value
                    = 2: linear extrapolation through last two boundary points

    A production code generator would typically trigger an error, if the folloing
    conditions are not fulfilled when calling one of the interpolation functions:
    - The values in x1_data[1:nx1], x2_data[1:nx2], x3_data[1:nx3] are
      strict monotonically increasing.
    - The data arguments (x1_data, x2_data, x3_data, nx1, nxs2, nx3) are
parameters.
    - The option arguments (interpolation, extrapolation) are literal constants.
```

```
    The production code generator decides which "search" method to use to find
the
    respective interval, or whether it can be directly found because there is an
    equidistant grid.
********************************************************************************
************/

function interpolation1D
    input  Real    x1;
    input  Real    x1_data[:];               // strict monotonically increasing
values
    input  Integer nx1;                      // 2 □ nx1 □ size(x1_data, 1)
    input  Real    y_data[size(x1_data, 1)];
    input  Integer interpolation;
    input  Integer extrapolation;
    output Real    y;
algorithm /*
    Constant or linear interpolation in [x1_data[1:nx1], y_data[1:nx1]]
    given the abszissa value x1.
*/ end interpolation1D;

function interpolation2D
    input  Real    x1;
    input  Real    x2;
    input  Real    x1_data[:];              // strict monotonically increasing
values
    input  Integer nx1;                     // 2 □ nx1 □ size(x1_data, 1)
    input  Real    x2_data[:];              // strict monotonically increasing
values
    input  Integer nx2;                     // 2 □ nx2 □ size(x2_data, 1)
    input  Real    y_data[size(x1_data, 1), size(x2_data, 1)];
    input  Integer interpolation;
    input  Integer extrapolation;
    output Real    y;
algorithm /*
    Constant or linear interpolation with x1_data[1:nx1], x2_data[1:nx2]
    abszissa values and y_data[1:nx1, 1:nx2] ordinate values, given the abszissa
value x1, x2.
*/ end interpolation2D;

function interpolation3D
    input  Real    x1;
    input  Real    x2;
    input  Real    x3;
    input  Real    x1_data[:];              // strict monotonically increasing
values
    input  Integer nx1;                     // 2 □ nx1 □ size(x1_data, 1)
    input  Real    x2_data[:];              // strict monotonically increasing
values
    input  Integer nx2;                     // 2 □ nx2 □ size(x2_data, 1)
    input  Real    x3_data[:];              // strict monotonically increasing
```

```
values
    input  Integer nx3;                    // 2 ≤ nx3 ≤ size(x3_data, 1)
    input  Real    y_data[size(x1_data, 1), size(x2_data, 1), size(x3_data, 1)];
    input  Integer interpolation;
    input  Integer extrapolation;
    output Real    y;
algorithm /*
    Constant or linear interpolation with x1_data[1:nx1], x2_data[1:nx2],
x3_data[1:nx3]
    abszissa values and y_data[1:nx1, 1:nx2, 1:nx3] ordinate values,
    given the abszissa value x1, x2, x3.
*/ end interpolation3D;
```

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

```
function divisionTowardsZero
    input Integer dividend;
    input Integer divisor;
    output Integer quotient;
algorithm /*
    quotient := integer(roundTowardsZero(real(dividend) / real(divisor)));
*/ end divisionTowardsZero;

function remainderTowardsZero
    input Integer dividend;
    input Integer divisor;
    output Integer remainder;
algorithm /*
    remainder := dividend - divisor * divisionTowardsZero(dividend, divisor);
*/ end remainderTowardsZero;
```

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

```
function realRemainderTowardsZero
    input Real dividend;
    input Real divisor;
    output Real remainder;
algorithm /*
    remainder := dividend - divisor * roundTowardsZero(dividend / divisor);
*/ end realRemainderTowardsZero;
```

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

```
function α1D
    input T1 β[:];
    output T3 δ[size(β, 1)];
algorithm /*
    for i in 1:size(β, 1) loop
        δ[i] := α(β[i]);
    end for;
*/ end α1D;

function α2D
    input T1 β[:, :];
    output T3 δ[size(β, 1), size(β, 2)];
algorithm /*
    for i in 1:size(β, 1) loop
        for j in 1:size(β, 2) loop
            δ[i, j] := α(β[i, j]);
        end for;
    end for;
*/ 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:

```
function α1D
    input T1 β[:];
    input T2 γ[size(β, 1)];
    output T3 δ[size(β, 1)];
algorithm /*
    for i in 1:size(β, 1) loop
        δ[i] := α(β[i], γ[i]);
    end for;
*/ end α1D;

function α2D
    input T1 β[:, :];
    input T2 γ[size(β, 1), size(β, 2)];
    output T3 δ[size(β, 1), size(β, 2)];
algorithm /*
    for i in 1:size(β, 1) loop
        for j in 1:size(β, 2) loop
            δ[i, j] := α(β[i, j], γ(i, j));
        end for;
    end for;
*/ 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 *multi-line-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:

```
block TestSolveLinearEquations
    input  Real u;
    output Real y[2];

protected

public
    method Startup
    protected
    algorithm
        self.y := {0.0, 0.0};
    end Startup;

    method DoStep
    protected
    algorithm
        self.y := solveLinearEquations(
            {
                {1.0       , 2.0*self.u},
                {4.0*self.u, 5.0}
            },
            {-2.0      , 4.0*self.u});
        /* Rudimentary error handling */
        if signal or hasNaN(self.y) then
            self.y = {0.0, 0.0}
        end;
    end DoStep
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*:

```
block TestLuSolve
    input  Real u;
    output Real y[2];

protected

public
    method Startup
    protected
    algorithm
        self.y := {0.0, 0.0};
    end Startup;

    method DoStep
    protected
        Real LU[2,2];
        Real pivots[2];
    algorithm
        (LU, pivots) := luFactorize(
            {
                {1.0,        2.0*self.u},
                {4.0*self.u, 5.0}
            });
        self.y := luSolve(
            LU,
            pivots,
            luSolve(
                LU,
                pivots,
                {-2.0, 4.0*self.u})
            + {-3.0, 6.0*self.u});
        /* Rudimentary error handling */
        if signal or isNaN(self.y) then
            self.y = {0.0, 0.0}
        end;
    end DoStep;
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.

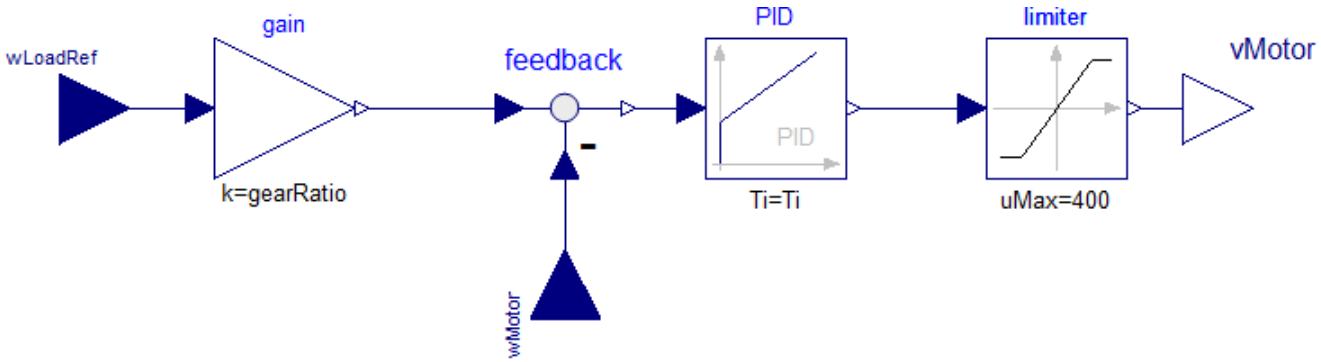
E-3: The following *block* interpolates in a vector of data points:

```
block TestInterpolation
    input  Real x;
    output Real y;

    parameter Real x_data[7];  // Define x-axis data points as tuneable
parameter vector.
    parameter Real y_data[7];  // Define y-axis data as tuneable parameter
vector.
    parameter Integer nx;      // Number of elements to interpolate (1 □ nx
□ 7).

protected

public
    method Startup
    protected
        Real x;
    algorithm
        x := 0.0;
        self.nx := 4;
        self.x_data := {1.0, 2.0, 3.0, 4.0 , 0.0, 0.0, 0.0};
        self.y_data := {1.0, 4.0, 9.0, 16.0, 0.0, 0.0, 0.0};
        self.y := interpolation1D(x, self.x_data, self.nx, self.y_data, 2,
2);
    end Startup;

    method DoStep
    protected
    algorithm
        self.y := interpolation1D(2*self.x, self.x_data, self.nx,
self.y_data, 2, 2);
    end DoStep;
end TestInterpolation;
```

4.2.7. 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:

```
block PID_Controller
    input  Real wLoadRef(min = -1.0e5, max = 1.0e5);
    input  Real wMotor  (min = -1.0e5, max = 1.0e5);
    output Real vMotor  (min = -1.0e7, max = 1.0e7);

    // Tunable parameters (can be changed via recalibration):
    parameter Real 'limiter.uMax'(min = 1.0, max = 1.0e5);
    parameter Real gearRatio(min = 10.0, max = 500.0);
    parameter Real Ti(min = 1.0e-7, max = 100.0);
    parameter Real Td(min = 1.0e-7, max = 100.0);
    parameter Real kd(min = 0.0, max = 1000.0);
    parameter Real k(min = 0.0, max = 1000.0);
    parameter Real stepSize // Can be local constant (if recalibration is not
supported).
        (min = 1.0e-10, max = 0.01 /* in physics-simulation tested sampling-range */
);

protected
    // Dependent parameters:
    parameter Real 'limiter.uMin'(min = -1.0e5, max = -1.0);

    // Discrete states:
    Real 'PID.I.x';
    Real 'PID.D.x';
    Real 'previous(feedback.y)';
    Boolean firstTick;

public
    method Startup
    algorithm
        // Initialize tunable parameters:
        self.'limiter.uMax' := 400.0;
        self.gearRatio := 105.0;
        self.Ti := 0.1;
        self.Td := 0.1;
        self.kd := 0.1;
```

```
        self.k := 10.0;
        self.stepSize := 1e-3;

        // Initialize dependent parameters:
        self.'limiter.uMin' := -self.'limiter.uMax';

        // Initialize discrete states:
        self.'PID.I.x' := 0.0;
        self.'PID.D.x' := 0.0;
        self.'previous(feedback.y)' := 0.0;
        self.firstTick := true;

        // Initialize outputs:
        self.vMotor := 0.0;
    end Startup;


    method Recalibrate
    algorithm
        // Update dependent parameters:
        self.'limiter.uMin' := -self.'limiter.uMax';
    end Recalibrate;


    /*
        Control-cycle function: Called at every clock tick.
    */
    method DoStep
    protected
        Real 'gain.y';
        Real 'feedback.y';
        Real 'derivative(PID.I.x)';
        Real 'derivative(PID.D.x)';
        Real 'PID.D.y';
        Real 'PID.y';
    algorithm

        if self.firstTick then
            self.firstTick := false;
        else
            'derivative(PID.I.x)' := self.'previous(feedback.y)' / self.Ti;
            'derivative(PID.D.x)' := (self.'previous(feedback.y)' - self.'PID.D.x') /
self.Td;

            self.'PID.I.x'        := self.'PID.I.x' + self.stepSize *
'derivative(PID.I.x)';
            self.'PID.D.x'        := self.'PID.D.x' + self.stepSize *
'derivative(PID.D.x)';
        end if;

        'gain.y'     := self.gearRatio * self.wLoadRef;
```

```
        'feedback.y' := 'gain.y' - self.wMotor;

        'PID.D.y' := self.kd * ('feedback.y' - self.'PID.D.x') / self.Td;
        'PID.y'   := self.k * ('PID.D.y' + self.'PID.I.x' + 'feedback.y');

        self.vMotor := (
            if 'PID.y' > self.'limiter.uMax' then
                self.'limiter.uMax'
            elseif 'PID.y' < self.'limiter.uMin' then
                self.'limiter.uMin'
            else
                'PID.y'
        );

        self.'previous(feedback.y)' := 'feedback.y';
    end DoStep;
end PID_Controller;
```

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

```
<?xml version="1.0" encoding="UTF-8"?>
<Manifest
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="../schemas/AlgorithmCode/efmiAlgorithmCodeManifest.xsd"
    xsdVersion="0.13.0"
    kind="AlgorithmCode"
    efmiVersion="1.0.0"
    id="{1e111db5-90e6-4e17-b2e5-4e215dbbdd49}"
    name="PID controller discretized by Explicit Euler method"
    version="0.1"
    generationDateAndTime="2020-11-10T12:33:22Z"
    generationTool="Manual"
    license="MIT">

    <Files>
        <File
            name="Controller.alg"
            id="FileID_1"
            path="./"
            needsChecksum="false"
            role="Code"/>
    </Files>

    <Clock id="ID_Clock" variableRefId="ID_7"/>

    <BlockMethods fileRefId="FileID_1" writeOutputs="AsSoonAsPossible">
        <BlockMethod id="ID_Startup" kind="Startup"/>
        <BlockMethod id="ID_Recalibrate" kind="Recalibrate"/>
        <BlockMethod id="ID_DoStep" kind="DoStep"/>
```

```
    </BlockMethods>

    <ErrorSignalStatus id="ID_ErrorSignalStatus"/>

    <Variables>
        <RealVariable
            name="'limiter.uMin'"
            id="ID_1"
            blockCausality="dependentParameter"
            start="-400.0"
            min="-1.0e5"
            max="-1.0"/>
        <RealVariable
            name="'limiter.uMax'"
            id="ID_2"
            blockCausality="tunableParameter"
            start="400.0"
            min="1.0"
            max="1.0e5"/>
        <RealVariable
            name="Ti"
            id="ID_3"
            blockCausality="tunableParameter"
            start="0.1"
            min="1.0e-7"
            max="100.0"/>
        <RealVariable
            name="Td"
            id="ID_4"
            blockCausality="tunableParameter"
            start="0.1"
            min="1.0e-7"
            max="100.0"/>
        <RealVariable
            name="kd"
            id="ID_5"
            blockCausality="tunableParameter"
            start="0.1"
            min="0.0"
            max="1000.0"/>
        <RealVariable
            name="k"
            id="ID_6"
            blockCausality="tunableParameter"
            start="10.0"
            min="0.0"
            max="1000.0"/>
        <RealVariable
            name="stepSize"
            id="ID_7"
            blockCausality="tunableParameter"
```

```
            start="1e-3"
            min="1.0e-10"
            max="0.01"/>
        <RealVariable
            name="gearRatio"
            id="ID_8"
            blockCausality="tunableParameter"
            start="105.0"
            min="10.0"
            max="500.0"/>
        <RealVariable
            name="wLoadRef"
            id="ID_9"
            blockCausality="input"
            start="0.0"
            min="-1.0e5"
            max="1.0e5">
        </RealVariable>
        <RealVariable
            name="wMotor"
            id="ID_10"
            blockCausality="input"
            start="0.0"
            min="-1.0e5"
            max="1.0e5">
        </RealVariable>
        <RealVariable
            name="vMotor"
            id="ID_11"
            blockCausality="output"
            start="0.0"
            min="-1.0e7"
            max="1.0e7">
        </RealVariable>
        <RealVariable
            name="'PID.I.x'"
            id="ID_12"
            blockCausality="state"
            start="0.0"/>
        <RealVariable
            name="'PID.D.x'"
            id="ID_13"
            blockCausality="state"
            start="0.0"/>
        <RealVariable
            name="'previous(feedback.y)'"
            id="ID_14"
            blockCausality="state"
            start="0.0"/>
        <BooleanVariable
            name="firstTick"
```

```
            id="ID_15"
            blockCausality="state"
            start="true"/>
    </Variables>

</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:

```
block LinearEquationSystem
    input Real u[4] (min=-1.0e7, max=1.0e7);
    output Real y[4];

protected
    // Constants:
    constant Real pi;
    constant Real stepSize;

    // Discrete states:
    Real x[4];
    Real v[4];
    Real 'derivative(x)'[4];
    Real 'derivative(v)'[4];


public
    /*
        Startup function: Called once at startup to initialize the
          internal memory of the block and return initial outputs.
    */
    method Startup
    algorithm
        // Initialize constants
        self.pi := 3.141592653589793;
        self.stepSize := 1.0e-2;

        // Initialize discrete states:
        self.x := {-3.0, 7.0, 19.0, 1.0};
        self.v := {0.0, 0.0, 0.0, 0.0};
```

```
        // Initial values for derivatives:
        self.'derivative(x)' := {0.0, 0.0, 0.0, 0.0};
        self.'derivative(v)' := {0.0, 0.0, 0.0, 0.0};

        // Return initial control-outputs:
        self.y := {0.0, 0.0, 0.0, 0.0};
    end Startup;


    method Recalibrate
    algorithm
    end Recalibrate;


    /*
        Control-cycle function: Called at every clock tick.
    */
    method DoStep
    protected
        Real M[4,4];
        Real F[4];

    algorithm
        self.x := self.x + self.stepSize * self.'derivative(x)';
        self.v := self.v + self.stepSize * self.'derivative(v)';

        self.y := {
            sin(self.x[1]) + self.x[3],
            -self.x[2],
            self.pi * 2.0 * cos(self.x[4] - self.x[2]),
            self.x[3] + self.x[1] / self.x[4]
        };

        // Check for NaN, e.g. if there was no solution of the linear system in the
previous call
        if isNaN(self.y[1]) or isNaN(self.y[2]) or isNaN(self.y[3]) or
isNaN(self.y[4]) then
            // Re-initialize the whole system to its start state
            self.x := {-3.0, 7.0, 19.0, 1.0};
            self.v := {0.0, 0.0, 0.0, 0.0};
            self.y := {0.0, 0.0, 0.0, 0.0};
        end if;

        M := {
            {
                -sin(self.x[3] + self.x[4]),
                self.x[4]^2 - self.x[2]^3,
                -4.0 * exp(self.x[3] * self.x[1]),
                cos(-self.x[2]) * self.x[3]
            },
            {
```

```
                (self.x[2] + 2.0 * self.x[4]) / self.x[1],
                -self.x[1],
                self.x[1] * self.x[2],
                sin(self.x[1] * self.x[2] * self.x[3])
            },
            {
                -self.x[4] + self.x[2] * self.x[1],
                6.0 * self.pi * cos(self.x[2]),
                -self.x[2],
                2.0 * (self.x[1] + sin(self.x[3] * self.pi))
            },
            {
                self.x[1]+cos(self.x[3]),
                -2.0*self.x[3]*self.x[4],
                -4.0 * self.x[3] * cos(self.x[2]),
                self.x[4] - self.x[1] * self.x[2]
            }
        };
        F := {
            self.u[1] - self.x[3]^2,
            -self.u[4] + self.x[2] * cos(self.x[1]),
            -self.u[4] + self.u[2] * self.x[4],
            self.u[2] + self.u[3]
        };

        self.'derivative(v)' := solveLinearEquations(M, F);
        self.'derivative(x)' := self.v;

    end DoStep;
end LinearEquationSystem;
```

The manifest summarising the controller's interface is:

```
<?xml version="1.0" encoding="UTF-8"?>
<Manifest
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="../schemas/AlgorithmCode/efmiAlgorithmCodeManifest.xsd"
    efmiVersion="1.0.0"
    xsdVersion="0.13.0"
    id="{351131cd-1e50-46d0-913a-240451d247c7}"
    kind="AlgorithmCode"
    name="Dynamic system discretized by Explicit Euler method"
    generationDateAndTime="2020-10-15T16:49:20Z"
    version="0.4.0"
    generationTool="Manual"
    license="MIT">

    <Files>
        <File
```

```
            name="Controller.alg"
            id="FileID_1"
            path="./"
            needsChecksum="false"
            role="Code"/>
    </Files>

    <Clock id="ID_Clock" variableRefId="ID_2"/>

    <BlockMethods fileRefId="FileID_1" writeOutputs="AsSoonAsPossible">
        <BlockMethod id="ID_Startup" kind="Startup"/>
        <BlockMethod id="ID_Recalibrate" kind="Recalibrate"/>
        <BlockMethod id="ID_DoStep" kind="DoStep"/>
    </BlockMethods>

    <ErrorSignalStatus id="ID_ErrorSignal"/>

    <Variables>
        <RealVariable
            name="pi"
            id="ID_1"
            blockCausality="constant"
            start="3.141592653589793"/>
        <RealVariable
            name="stepSize"
            id="ID_2"
            blockCausality="constant"
            start="1e-2"/>
        <RealVariable
            name="u"
            id="ID_3"
            blockCausality="input"
            start="0.0 0.0 0.0 0.0"
            min="-1.0e7"
            max="1.0e7">
            <Dimensions>
                <Dimension number="1" size="4"/>
            </Dimensions>
        </RealVariable>
        <RealVariable
            name="y"
            id="ID_4"
            blockCausality="output"
            start="0.0 0.0 0.0 0.0">
            <Dimensions>
                <Dimension number="1" size="4"/>
            </Dimensions>
        </RealVariable>
        <RealVariable
            name="v"
            id="ID_5"
```

```
            blockCausality="state"
            start="0.0 0.0 0.0 0.0">
            <Dimensions>
                <Dimension number="1" size="4"/>
            </Dimensions>
        </RealVariable>
        <RealVariable
            name="x"
            id="ID_6"
            blockCausality="state"
            start="-3.0 7.0 19.0 1.0">
            <Dimensions>
                <Dimension number="1" size="4"/>
            </Dimensions>
        </RealVariable>
        <RealVariable
            name="'derivative(x)'"
            id="ID_7"
            blockCausality="state"
            start="0.0 0.0 0.0 0.0">
            <Dimensions>
                <Dimension number="1" size="4"/>
            </Dimensions>
        </RealVariable>
        <RealVariable
            name="'derivative(v)'"
            id="ID_8"
            blockCausality="state"
            start="0.0 0.0 0.0 0.0">
            <Dimensions>
                <Dimension number="1" size="4"/>
            </Dimensions>
        </RealVariable>
    </Variables>

</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:

```
```

```
block VehicleModel
    input Real u[2](min=-1.0e7, max=1.0e7);
    input Real 'derivative(u)'[2](min=-1.0e7, max=1.0e7);
    output Real x[8];

    // Tunable parameters (can be changed via recalibration):
    parameter Real FdF;
    parameter Real m;
    parameter Real m2;
    parameter Real h;
    parameter Real lF;
    parameter Real lR;
    parameter Real g;
    parameter Real Jx2;
    parameter Real mu;
    parameter Real cF;
    parameter Real cR;
    parameter Real Jz1;
    parameter Real Jz2;
    parameter Real Jy2;
    parameter Real cphi;
    parameter Real dphidot;
    parameter Real b1;
    parameter Real b2;
    parameter Real stepSize;

protected
    // Dependent parameters:
    parameter Real FlV;
    parameter Real FzR;
    parameter Real FzF;

    // Discrete states:
    Real q[4];
    Real dx[8];

public
    /*
        Startup function: Called once at startup to initialize the
          internal memory of the block and return initial outputs.
    */
    method Startup
    algorithm
        // Initialize tunable parameters
        self.FdF := 15.0;
        self.m := 14300.0;
        self.m2 := 12487.0;
        self.h := 1.15;
        self.lF := 1.95;
        self.lR := 1.54;
        self.g := 9.81;
```

```
        self.Jx2 := 24201.0;
        self.mu := 1.0;
        self.cF := 582.0e+3;
        self.cR := 783.0e3;
        self.Jz1 := 3654.0;
        self.Jz2 := 34917.0;
        self.Jy2 := 3491.7;
        self.cphi := 457.0e+3;
        self.dphidot := 100.0e3;
        self.b1 := 0.2;
        self.b2 := 0.1;
        self.stepSize := 1.0e-2;

        // Initialize dependent parameters
        self.FlV := self.FdF;
        self.FzR := self.m*self.g*self.lF/(self.lR + self.lF);
        self.FzF := self.m*self.g - self.FzR;

        // Initialize inputs
        // u = {0.0, 0.0};
        // 'derivative(u)' = {0.0, 0.0};

        // Initialize states and outputs
        self.q := {0.0, 0.0, 0.0, 0.0};
        self.dx := {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
        self.x := {0.0, 0.0, 0.0, 0.0, 10.0, 0.0, 0.0, 0.0};
    end Startup;

    /*
        Recalibration function: Called to change tunable parameters
          during operation.
    */
    method Recalibrate
    algorithm
        // Update dependent parameters:
        self.FlV := self.FdF;
        self.FzR := self.m*self.g*self.lF/(self.lR + self.lF);
        self.FzF := self.m*self.g - self.FzR;
    end Recalibrate;

    /*
        Control-cycle function: Called at every clock tick.
    */
    method DoStep
    protected
        Real sx;
        Real sy;
        Real psi;
        Real phi;
        Real vx;
        Real vy;
```

```
        Real r;
        Real phidot;
        Real delta;
        Real FyD;
        Real q1;
        Real q2;
        Real q3;
        Real q4;
        Real deltadot;
        Real FyDdot;

        Real FdF;
        Real FlV;
        Real m;
        Real m2;
        Real h;
        Real lF;
        Real lR;
        Real g;
        Real Jx2;
        Real mu;
        Real cF;
        Real cR;
        Real Jz1;
        Real Jz2;
        Real Jy2;
        Real FzR;
        Real FzF;
        Real cphi;
        Real dphidot;
        Real b1;
        Real b2;

        Real G[4,4];

        Real rs2[4];
        Real dx1[4];

        Real help1;
        Real help2;
        Real help3;

    algorithm

        for i in 1:8 loop
            self.x[i] := self.x[i] + self.dx[i];
        end for;

        for i in 1:4 loop
            self.q[i] := self.dx[4+i]/self.stepSize;
        end for;
```

```
sx     := self.x[1];
sy     := self.x[2];
psi    := self.x[3];
phi    := self.x[4];
vx     := self.x[5];
vy     := self.x[6];
r      := self.x[7];
phidot := self.x[8];

delta  := self.u[1];
FyD    := self.u[2];

q1     := self.q[1];
q2     := self.q[2];
q3     := self.q[3];
q4     := self.q[4];

deltadot := self.'derivative(u)'[1];
FyDdot   := self.'derivative(u)'[2];

FdF := self.FdF;
FlV := self.FlV;

m   := self.m;
m2  := self.m2;
h   := self.h;
lF  := self.lF;
lR  := self.lR;
g   := self.g;
Jx2 := self.Jx2;

mu := self.mu;
cF := self.cF;
cR := self.cR;

Jz1     := self.Jz1;
Jz2     := self.Jz2;
Jy2     := self.Jy2;
FzR     := self.FzR;
FzF     := self.FzF;
cphi    := self.cphi;
dphidot := self.dphidot;

b1 := self.b1;
b2 := self.b2;

help1 := sqrt(vx^2 + vy^2);
help2 := (vx^2 + vy^2)^1.5;
help3 := h^2*m2 + Jy2 - Jz2;
```

```
        G[1,1] :=
            (
                  mu*(lF*r*vx + help1*vy)*self.stepSize*cF*sin(delta)
                + help2*m
            )
            / (help2*self.stepSize);
        G[1,2] :=
            -(
                   mu*(-lF*r*vy + help1*vx)*cF*sin(delta)
                + help2*r*m
            )
            / help2;
        G[1,3] :=
            (
                  2.0*h*m2*phidot*cos(phi)*self.stepSize*help1
                - mu*cF*lF*sin(delta)*self.stepSize
                + h*m2*sin(phi)*help1
                - m*vy*self.stepSize*help1
            )
            / (self.stepSize*help1);
        G[1,4] :=
            h*m2*(
                  -2.0*sin(phi)*phidot*r*self.stepSize
                + cos(phi)*q3*self.stepSize
                + 2.0*r*cos(phi)
            );
        G[2,1] :=
            (
                  (-cos(delta)*cF*mu*vy - cR*mu*vy + m*r*(vx^2+vy^2))*help1
                - r*mu*vx*(cos(delta)*cF*lF - cR*lR)
            )
            / help2;
        G[2,2] :=
            (
                  (cos(delta)*cF*mu*vx*self.stepSize + cR*mu*vx*self.stepSize +
m*(vx^2 + vy^2))*help1
                - self.stepSize*r*mu*vy*(cos(delta)*cF*lF - cR*lR)
            )
            / (help2*self.stepSize);
        G[2,3] :=
            (
                  2.0*h*m2*r*sin(phi)*help1
                + mu*cF*lF*cos(delta)
                + m*vx*help1
                - mu*cR*lR
            )
            / help1;
        G[2,4] :=
            m2*(
                  (-1.0 + (phidot^2 + r^2)*self.stepSize^2)*cos(phi)
                + self.stepSize*sin(phi)*(q4*self.stepSize + 2.0*phidot)
```

```
            )
            * (h/self.stepSize);
        G[3,1] :=
            (
                (
                      -cos(delta)*cF*lF*mu*vy*self.stepSize
                    + h*m2*(vx^2 + vy^2)*sin(phi)
                    + cR*lR*mu*vy*self.stepSize
                ) * help1
                - self.stepSize*r*mu*vx*(lF^2*cF*cos(delta) + lR^2*cR)
            )
            / (help2*self.stepSize);
        G[3,2] :=
            -(
                  (-cos(delta)*cF*lF*mu*vx + h*r*m2*(vx^2 + vy^2)*sin(phi) +
cR*lR*mu*vx)*help1
                + vy*r*mu*(lF^2*cF*cos(delta) + lR^2*cR)
            )
            / help2;
        G[3,3] :=
            2.0*(
                (
                      (-0.5*h^2*m2 - 0.5*Jy2 + 0.5*Jz2)*cos(phi)^2
                    + phidot*self.stepSize*sin(phi)*help3*cos(phi)
                    - 0.5*sin(phi)*h*m2*vy*self.stepSize
                    + 0.5*h^2*m2
                    + 0.5*Jy2
                    + 0.5*Jz1
                ) * help1
                + 0.5*mu*self.stepSize*(lF^2*cF*cos(delta) + lR^2*cR)
            )
            / (help1*self.stepSize);
        G[3,4] :=
              4.0*phidot*self.stepSize*r*help3*cos(phi)^2
            + (2.0*help3*(q3*self.stepSize + r)*sin(phi) - h*self.stepSize*m2*(r*vy -
q1))*cos(phi)
            - 2.0*phidot*self.stepSize*r*help3;
        G[4,1] := -h*m2*r*cos(phi);
        G[4,2] := -h*m2*cos(phi) / self.stepSize;
        G[4,3] := -2.0*(help3*r*sin(phi) + 0.5*h*m2*vx)*cos(phi);
        G[4,4] :=
            (
                  -2.0*self.stepSize^2*r^2*help3*cos(phi)^2
                - cos(phi)*g*h*m2*self.stepSize^2
                + self.stepSize^2*h*m2*(r*vx + q2)*sin(phi)
                + (help3*r^2 + cphi)*self.stepSize^2
                + dphidot*self.stepSize
                + h^2*m2
                + Jx2
            )
            / self.stepSize;
```

```
        rs2[1] :=
            2.0*(
                (
                      -0.5*self.stepSize*(cF*mu*(delta - atan2(vy, vx))*cos(delta) +
sin(delta)*(cF*mu + FlV))*deltadot
                    + 0.5*atan2(vy, vx)*sin(delta)*cF*mu
                    - 0.5*sin(delta)*cF*delta*mu
                    - 0.5*h*m2*phidot*(q3*self.stepSize + 2.0*r)*cos(phi)
                    + 0.5*FlV*cos(delta)
                    + r*(sin(phi)*h*m2*phidot^2*self.stepSize + 0.5*m*vy)
                ) * help1
                + 0.5*cF*lF*mu*r*(deltadot*cos(delta)*self.stepSize + sin(delta))
            )
            / help1;
        rs2[2] :=
            -(
                (
                    (
                          mu*cF*(delta - atan2(vy, vx))*sin(delta)
                        - cos(delta)*(cF*mu + FlV)
                    ) * (self.stepSize*deltadot)
                    - FyDdot*self.stepSize
                    + mu*(cos(delta)*cF + cR)*atan2(vy, vx)
                    - cos(delta)*cF*delta*mu
                    + h*m2*(phidot*q4*self.stepSize + phidot^2+r^2)*sin(phi)
                    + h*phidot*self.stepSize*m2*(phidot^2 + r^2)*cos(phi)
                    + m*r*vx
                    - FlV*sin(delta)
                    - FyD
                ) * help1
                - mu*r*(self.stepSize*sin(delta)*deltadot*cF*lF - cos(delta)*cF*lF +
cR*lR)
            )
            / help1;
        rs2[3] :=
            -2.0*(
                (
                      0.5*lF*(mu*cF*(delta - atan2(vy, vx))*sin(delta) -
cos(delta)*(cF*mu + FlV))*self.stepSize*deltadot
                    - 0.5*FyDdot*b1*self.stepSize
                    + 0.5*mu*(cos(delta)*cF*lF - cR*lR)*atan2(vy, vx)
                    + 2.0*r*phidot^2*self.stepSize*help3*cos(phi)^2
                    + phidot*(help3*(q3*self.stepSize + r)*sin(phi) +
0.5*h*self.stepSize*m2*(-r*vy + q1))*cos(phi)
                    - 0.5*sin(phi)*h*m2*r*vy
                    - 0.5*cos(delta)*cF*delta*lF*mu
                    - 0.5*FlV*sin(delta)*lF
                    - r*phidot^2*self.stepSize*help3
                    - 0.5*b1*FyD
                ) * help1
```

```
                - 0.5*r*mu*(deltadot*sin(delta)*cF*lF^2*self.stepSize -
lF^2*cF*cos(delta) - lR^2*cR)
            )
            / help1;
        rs2[4] :=
              self.stepSize*b2*FyDdot
            + 2.0*phidot*self.stepSize*r^2*help3*cos(phi)^2
            + (r^2*help3*sin(phi) + h*m2*(g*phidot*self.stepSize + r*vx))*cos(phi)
            + (-phidot*(r*vx+q2)*self.stepSize + g)*h*m2*sin(phi)
            - phidot*(help3*r^2 + cphi)*self.stepSize
            - cphi*phi
            - dphidot*phidot
            + b2*FyD;

        dx1 := solveLinearEquations(G, rs2);
        for i in 1:4 loop
            self.dx[4+i] := dx1[i];
            self.dx[i] := self.stepSize*(self.x[4+i]+dx1[i]);
        end for;

        // Check for NaN, caused by e.g. a failed solution of the linear system
        if isNaN(self.x[1]) or isNaN(self.x[2]) or isNaN(self.x[3]) or
isNaN(self.x[4]) or
           isNaN(self.x[5]) or isNaN(self.x[6]) or isNaN(self.x[7]) or
isNaN(self.x[8]) then
           self.q := {0.0, 0.0, 0.0, 0.0};
           self.dx := {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
           self.x := {0.0, 0.0, 0.0, 0.0, 10.0, 0.0, 0.0, 0.0};
        end if;

    end DoStep;
end VehicleModel;
```

The resulting manifest is:

```
<?xml version="1.0" encoding="utf-8"?>
<Manifest efmiVersion="1.0.0"
          generationDateAndTime="2020-10-15T16:52:13Z"
          generationTool="Manual"
          id="{e3eae104-6417-4783-8c05-7c14e6fab8a6}"
          kind="AlgorithmCode"
          license="MIT"
          name="Vehicle model discretized by Linearly implicit Euler method"
          version="0.2"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsdVersion="0.13.0"

xsi:noNamespaceSchemaLocation="../schemas/AlgorithmCode/efmiAlgorithmCodeManifest.xsd"
>
  <Files>
```

```
    <File
          id="FileID_1"
          name="Controller.alg"
          needsChecksum="false"
          path="./"
          role="Code" />
  </Files>
  <Clock id="ID_Clock" variableRefId="ID_1" />
  <BlockMethods fileRefId="FileID_1" writeOutputs="AsSoonAsPossible">
    <BlockMethod id="ID_Startup" kind="Startup" />
    <BlockMethod id="ID_DoStep" kind="DoStep" />
    <BlockMethod id="ID_Recalibrate" kind="Recalibrate" />
  </BlockMethods>
  <ErrorSignalStatus id="ID_ErrorSignal"/>
  <Variables>
    <RealVariable blockCausality="tunableParameter"
          id="ID_1"
          name="stepSize"
          start="1e-2" />
    <RealVariable blockCausality="tunableParameter"
          id="ID_2"
          name="FdF"
          start="15.0" />
    <RealVariable blockCausality="dependentParameter"
          id="ID_3"
          name="FlV"
          start="15.0" />
    <RealVariable blockCausality="tunableParameter"
          id="ID_4"
          name="m"
          start="14300.0" />
    <RealVariable blockCausality="tunableParameter"
          id="ID_5"
          name="m2"
          start="12487.0" />
    <RealVariable blockCausality="tunableParameter"
          id="ID_6"
          name="h"
          start="1.15" />
    <RealVariable blockCausality="tunableParameter"
          id="ID_7"
          name="lF"
          start="1.95" />
    <RealVariable blockCausality="tunableParameter"
          id="ID_8"
          name="lR"
          start="1.54" />
    <RealVariable blockCausality="tunableParameter"
          id="ID_9"
          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"
        min="-1.0e7"
        max="1.0e7">
      <Dimensions>
        <Dimension number="1"
                   size="2" />
      </Dimensions>
    </RealVariable>
    <RealVariable blockCausality="input"
        id="ID_24"
        name="&amp;#39;derivative(u)&amp;#39;"
        start="0.0 0.0"
        min="-1.0e7"
        max="1.0e7">
      <Dimensions>
        <Dimension number="1"
                   size="2" />
      </Dimensions>
    </RealVariable>
    <RealVariable blockCausality="output"
        id="ID_25"
        name="x"
        start="0.0 0.0 0.0 0.0 10.0 0.0 0.0 0.0">
      <Dimensions>
        <Dimension number="1"
                   size="8" />
      </Dimensions>
    </RealVariable>
    <RealVariable blockCausality="state"
        id="ID_26"
        name="q"
        start="0.0 0.0 0.0 0.0">
      <Dimensions>
        <Dimension number="1"
                   size="4" />
      </Dimensions>
    </RealVariable>
    <RealVariable blockCausality="state"
        id="ID_27"
        name="dx"
        start="0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0">
      <Dimensions>
        <Dimension number="1"
                   size="8" />
      </Dimensions>
    </RealVariable>
  </Variables>
```

```
</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

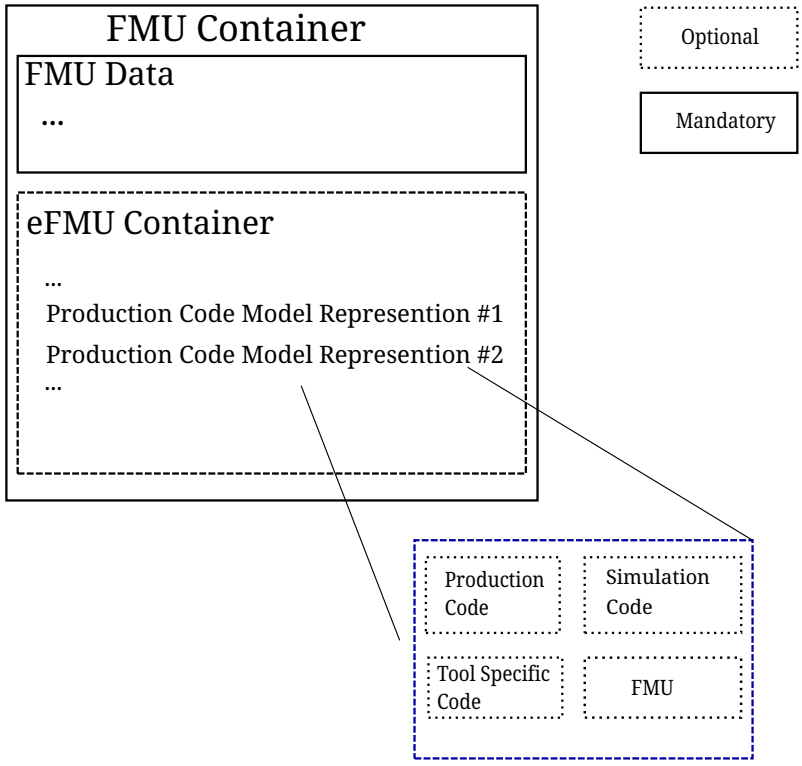
[2] Only the *bounded-iteration* rule has *loop-iterator-declaration* within its definition-list.

Chapter 5. Production Code Model Representation

5.1. Introduction

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.

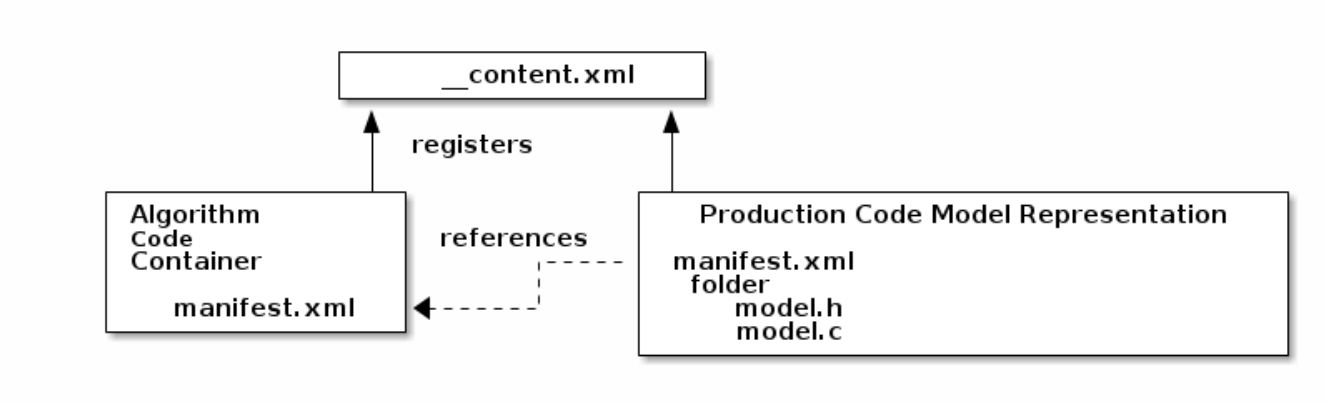
As mentioned before an eFMU container can contain any number of Production Code Model Representations.



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 the 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.



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.

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.

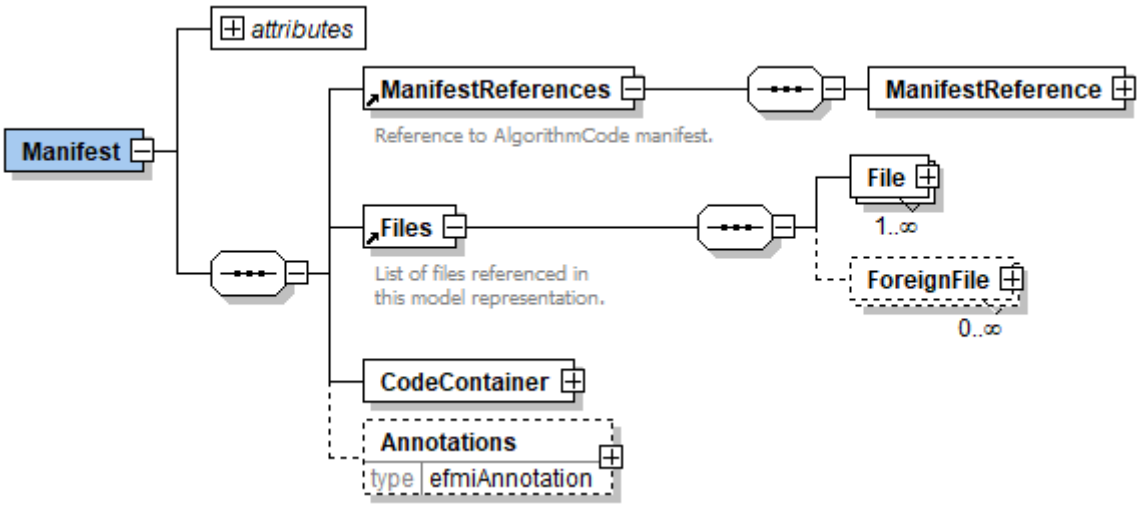
A generic target allows for testing and simulating the Production Code in an environment other than the target embedded device, which may require additional software parts to interface with the environment. These software parts can simulate parts of an operating system of the microcontroller, create stubs to represent other software functions that interact with the software-under-test or handle inputs, outputs and the execution.

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.

5.2. 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 5.2.2.
`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.

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.

```
...
typedef struct {
   float *deflection;
   float rigidity;
} spring;

....

float f4(float x1, float x2, spring[] springs, float *y2) {
    ...
    springs[0].deflection = ....  // update of state s1
    *y2 = ...  // y2 expression
    return ...;  // y1 expression
}
```

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.

5.2.1. 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:

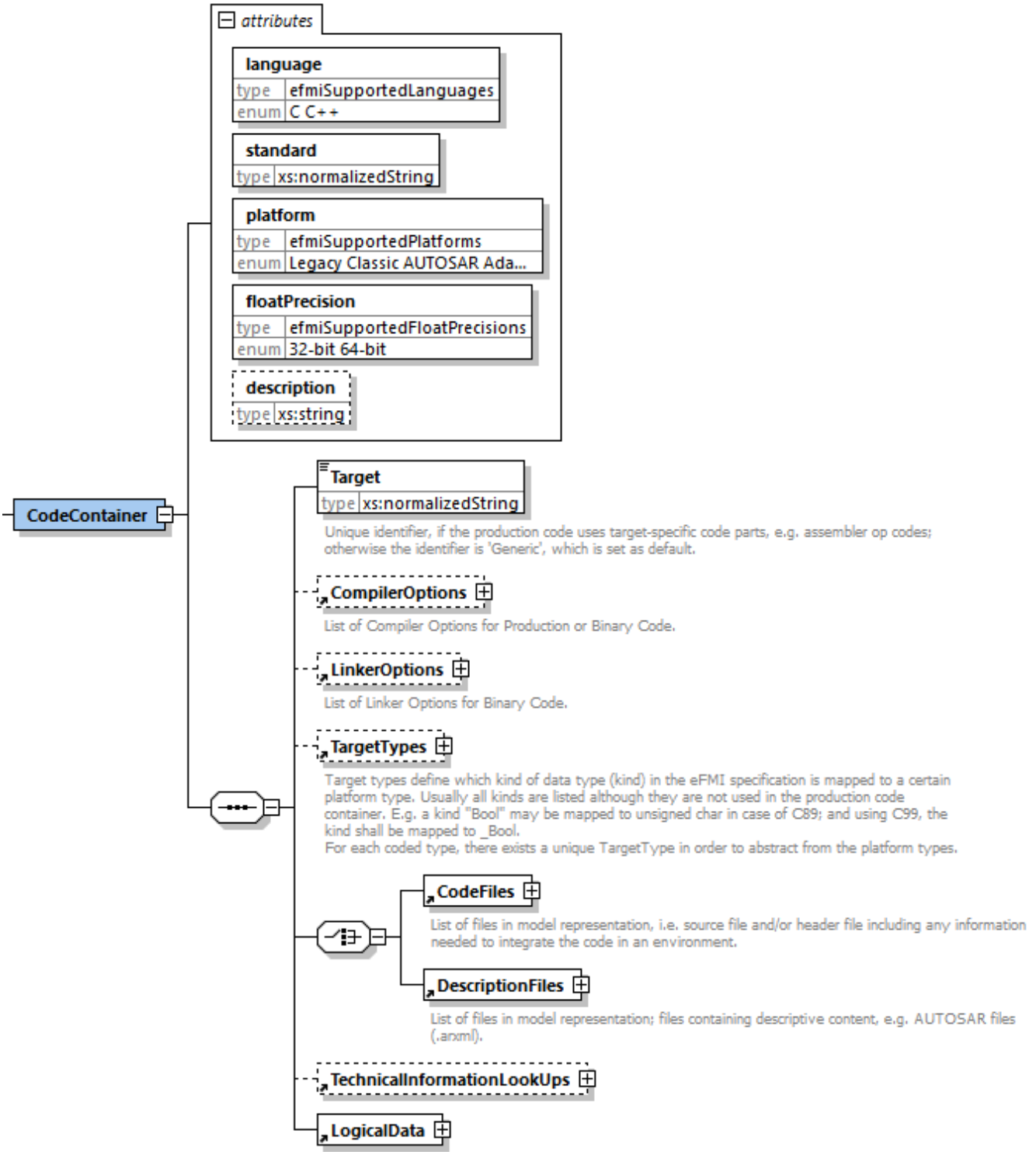
- references ("includes") to other files (defined in the Production Code manifest).
- defined types in that file (refering to the defined and standardized target types). Usually these are specifically defined names for the type like e.g. "uint8" that are used in the actual Production Code. These defined 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.

5.2.2. Code Container

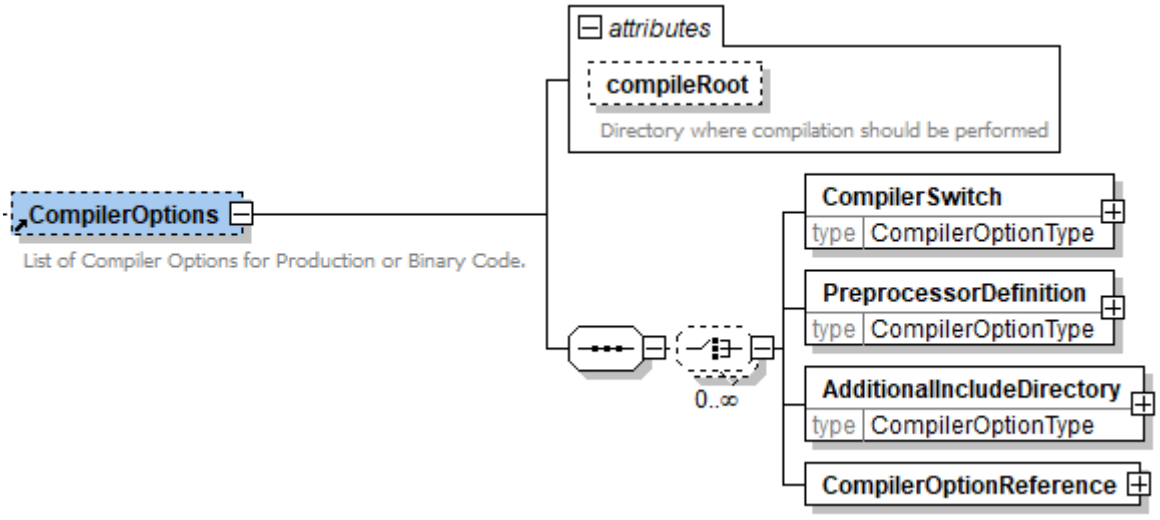
The code container groups the actual Production Code Model Representaion content, and gives specification for the following details:



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)

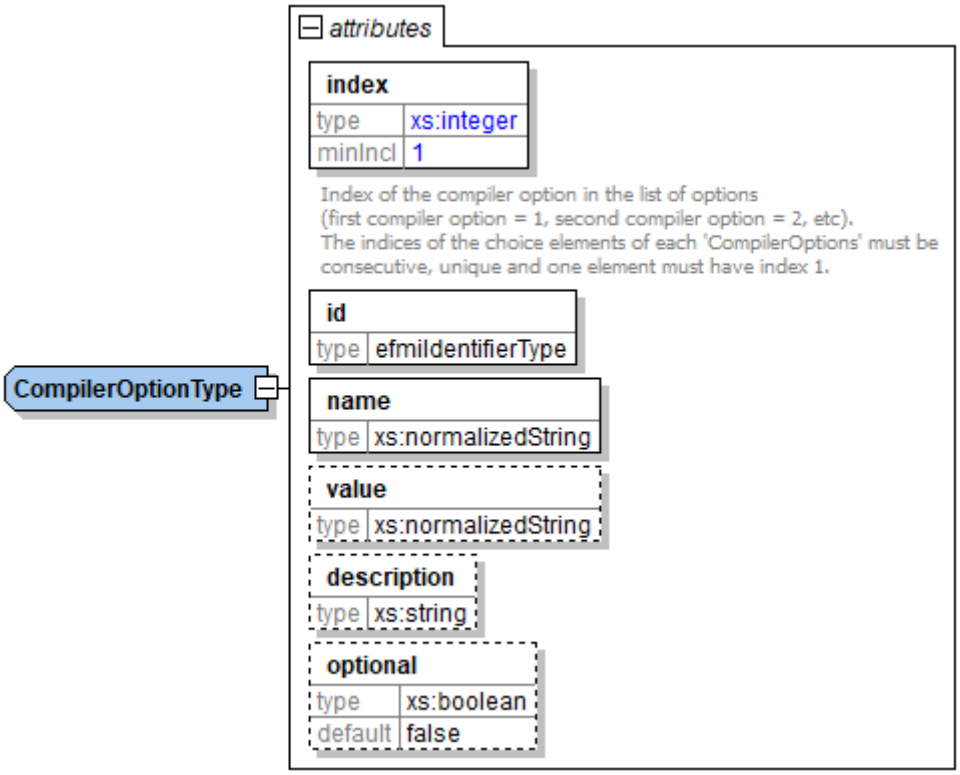
Name	Description
floatPrecision	Floating point precision of the target platform. Currently, the following values are possible: "32-bit" or "64-bit".
description	Optional description
Target	Unique identifier, if the production code uses target-speciic code parts, for example assembler op codes; otherwise the identifier is the default Generic.
CompilerOptions	List of Compiler Options for Production or Binary Code. For more details, see section Section 5.2.2.1.
LinkerOptions	List of Linker Options for Production or Binary Code. For more details, see section Section 5.2.2.4.
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 5.2.2.7.
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 5.2.3.
DescriptionFiles	List of files in model representation; files containing descriptive content, e.g. AUTOSAR files (.arxml). For more details, see section Section 5.2.4.
TechnicalInformationLookUps	Facilitates a quick access to information in the manifest and the associated C files. For more details, see section Section 5.2.5.
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 5.2.6.

Compiler Options



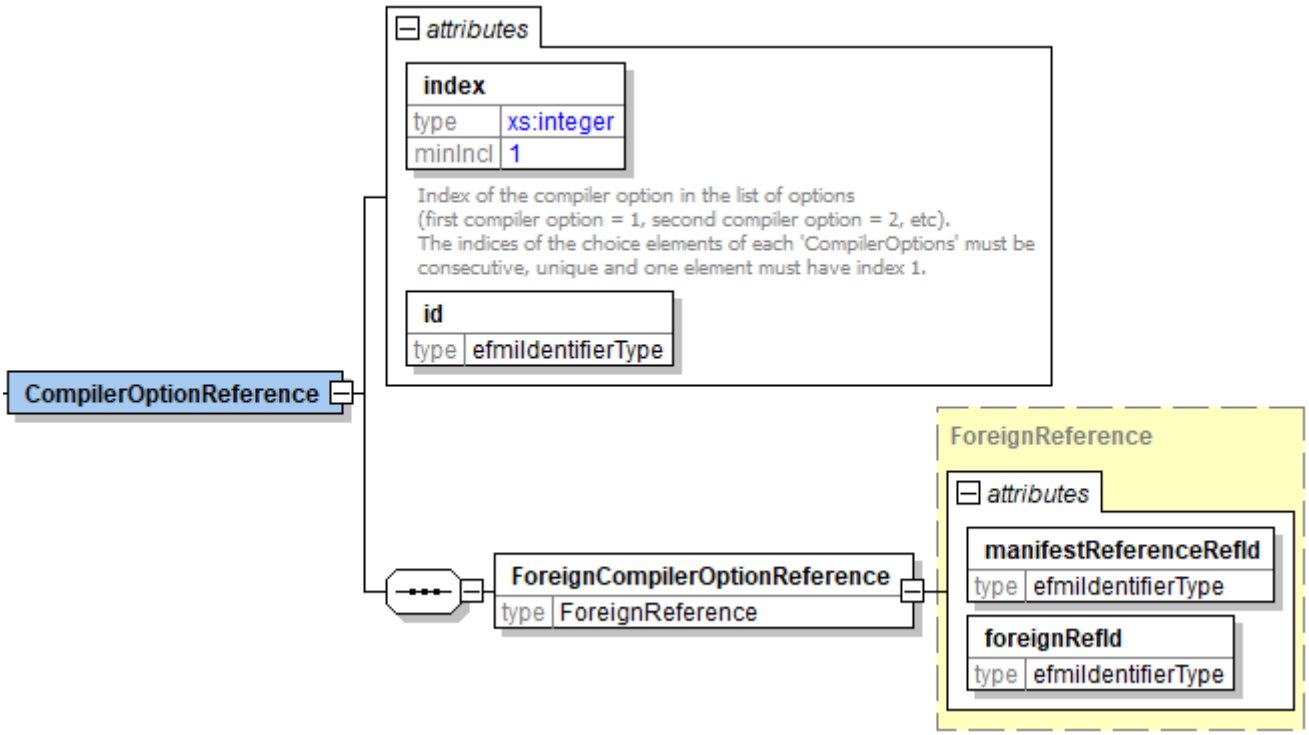
Name	Description
compileRoot	Directory where compilation should be performed.
CompilerSwitch	Compiler switch, see Section 5.2.2.2.
PreprocessorDefinition	Preprocessor definition, see Section 5.2.2.2.
AdditionalIncludeDirectory	Additional include directory, see Section 5.2.2.2.
CompilerOptionReference	Reference to option in another manifest file, see Section 5.2.2.3.

Compiler Option Type



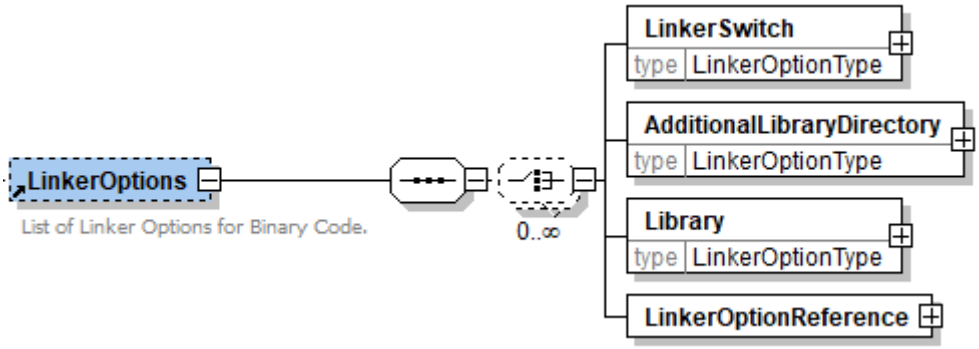
Name	Description
id	Id of option.
name	Name of option.
value	Value of option.
description	Optional description of option.
optional	Definition of option is optional. Possible values: "false" (default) or "true".

Compiler Option Reference



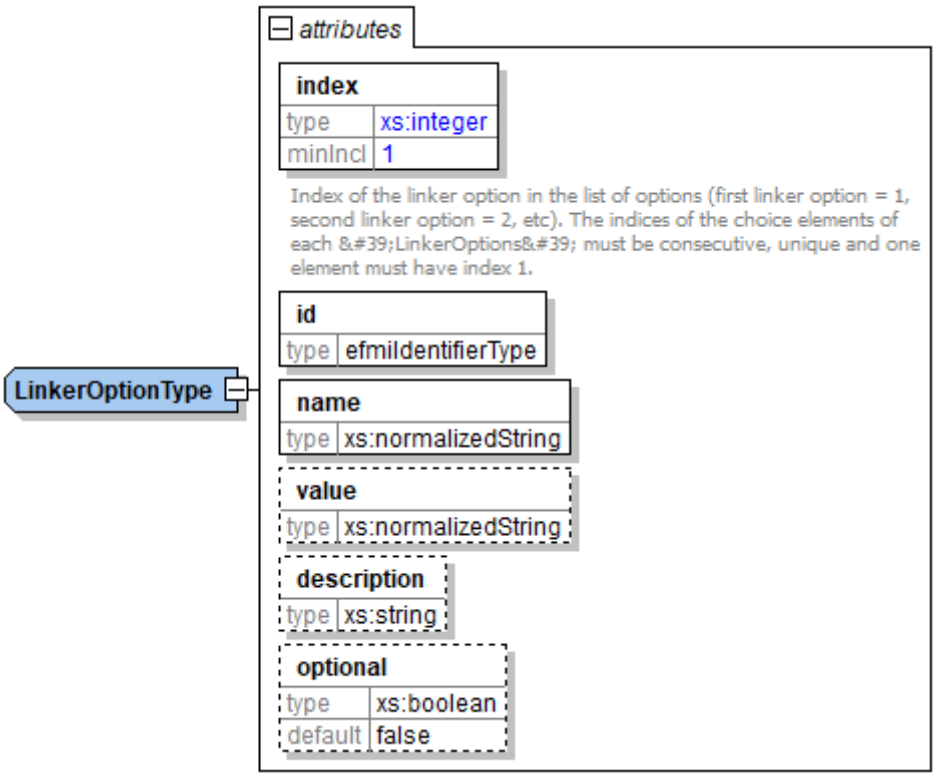
Name	Description
index	Index of the option reference in the list of option references.
id	Id of option reference.
manifestReferenceRefId	If of foreign manifest file.
foreignRefId	Id of option in foreign manifest file.

Linker Options



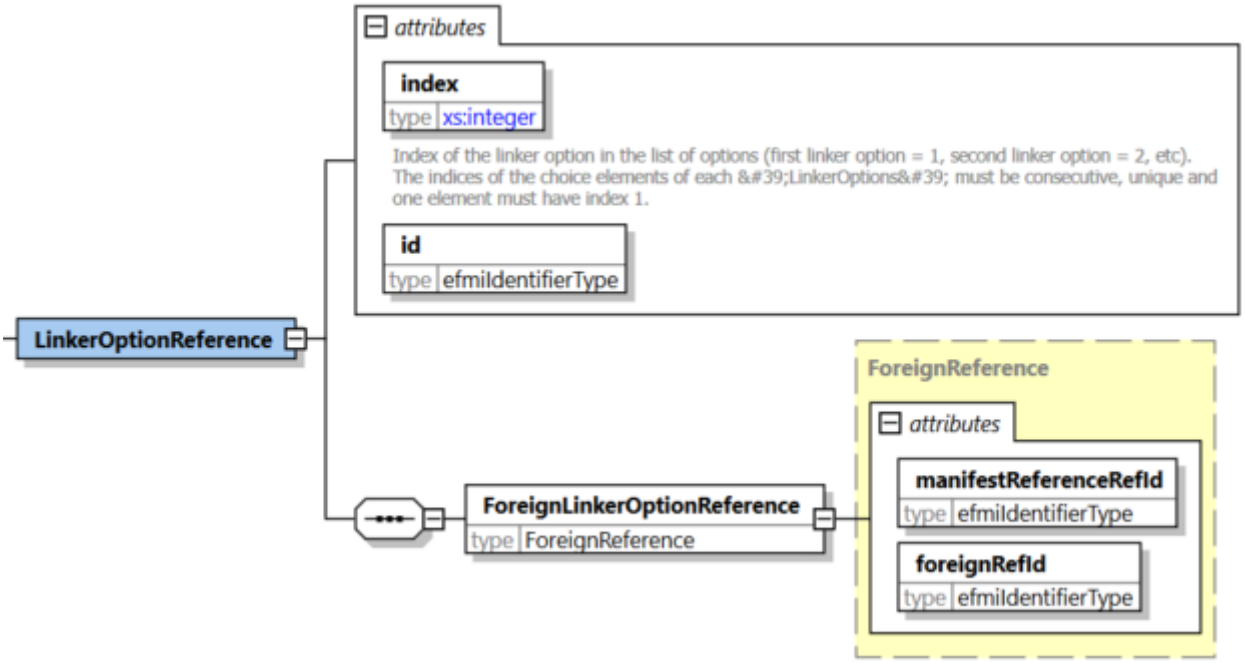
Name	Description
LinkerSwitch	The linker switches of type [LinkerOptionType].
Library	Library of type [LinkerOptionType].
AdditionalLibraryDirectory	Additional library directory of type [LinkerOptionType].
LinkerOptionReference	A list of option references, see [OptionReference].

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"`.

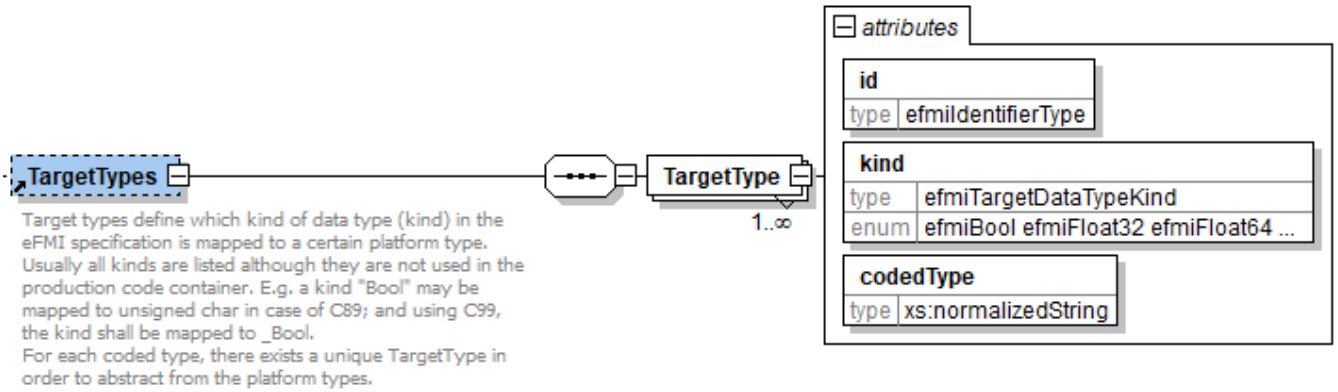
Linker Option Reference



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.

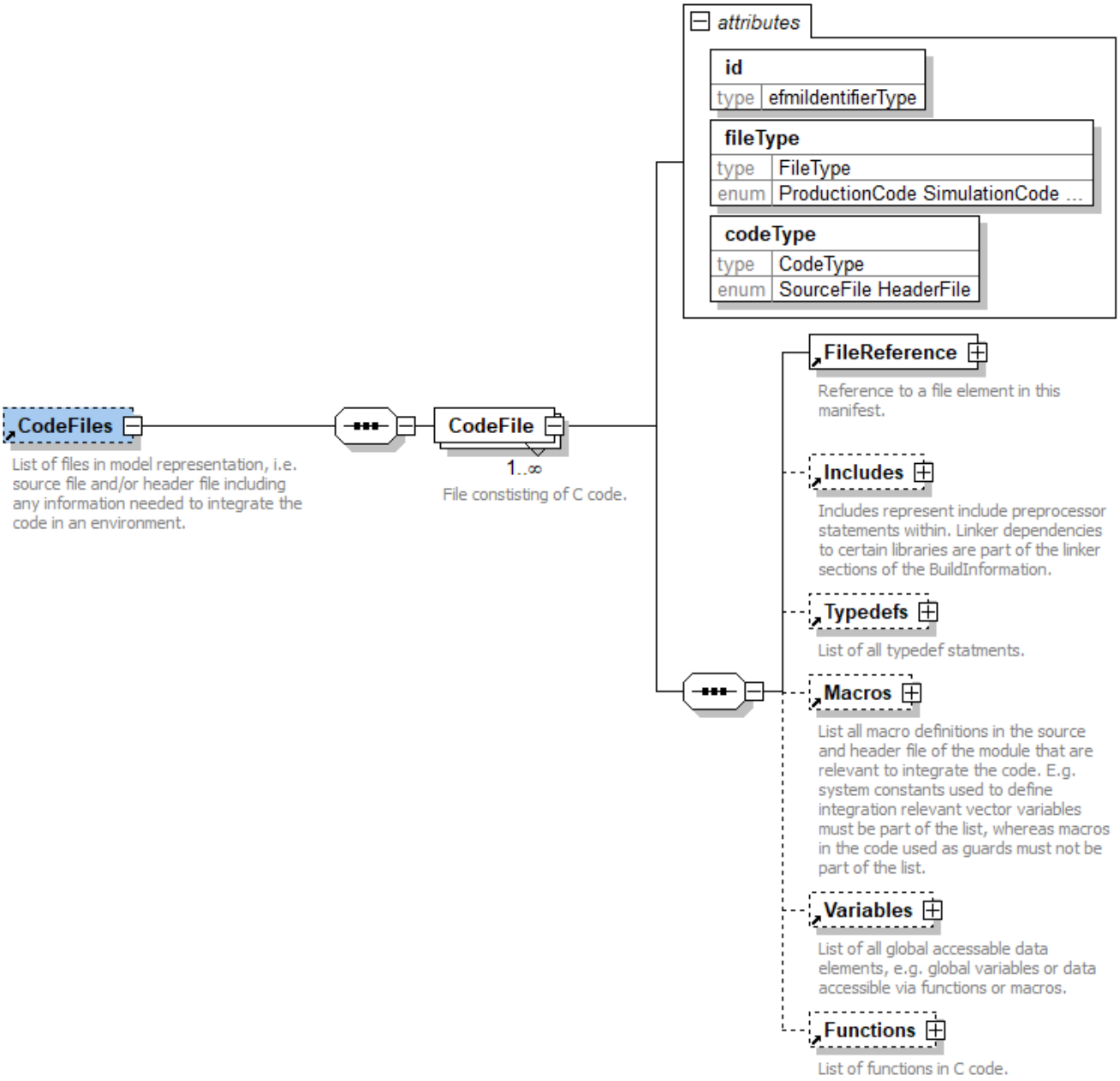
Name	Description
`kind`	The kind of the target type. The value must be one of the predefined kinds from the following list: `"efmiInteger8"`, `"efmiUnsignedInteger8"`, ... , `"efmiUnsignedInteger64"`, `"efmiFloat32"`, `"efmiFloat64"`, `"efmiFloat128`, `"efmiBoolean"`, `"efmiVoid"`.
`codedType`	The actual Production Code type to be used, e.g. `"unsigned char"`.

Example:

```
<TargetType id="TT_float64" kind="efmiFloat64" codedType="double"/>
```

5.2.3. 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



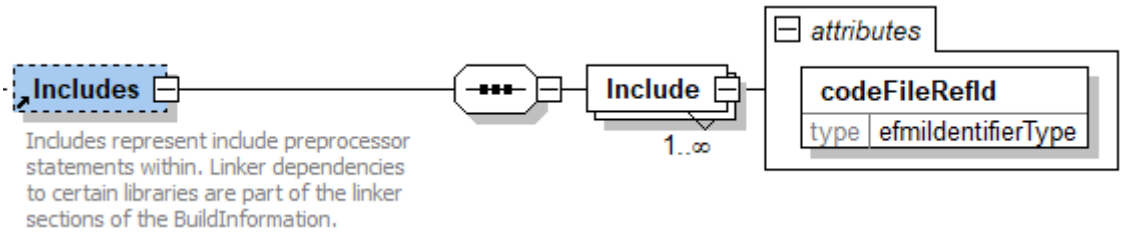
Name	Description
id	Unique id.
fileType	Type of the file. Allowed values: "ProductionCode", "SimulationCode", "ToolSpecificCode".
codeType	Type of the code. Allowed values: "SourceFile", "HeaderFile".
FileReferece	Reference to a file element in this manifest file, see Section 2.3.4.2.
Includes	Definition of include files, see Section 5.2.3.1.
Typedefs	Definition of typedefs, see Section 5.2.3.2.
Macros	Definition of macros, see Section 5.2.3.2.3.
Variables	Definition of variables, see Section 4.1.6.
Functions	Definition of functions, see Section 5.2.3.2.5.

Example:

```
<CodeFile id="C_1" fileType="ProductionCode" codeType="SourceFile">
   <FileReference fileRefId="F_22" kind="code"/>
   .....
</CodeFile>
```

Includes

Includes represent include preprocessor statements. Linker dependencies to certain libraries are part of the linker sections of the BuildInformation.



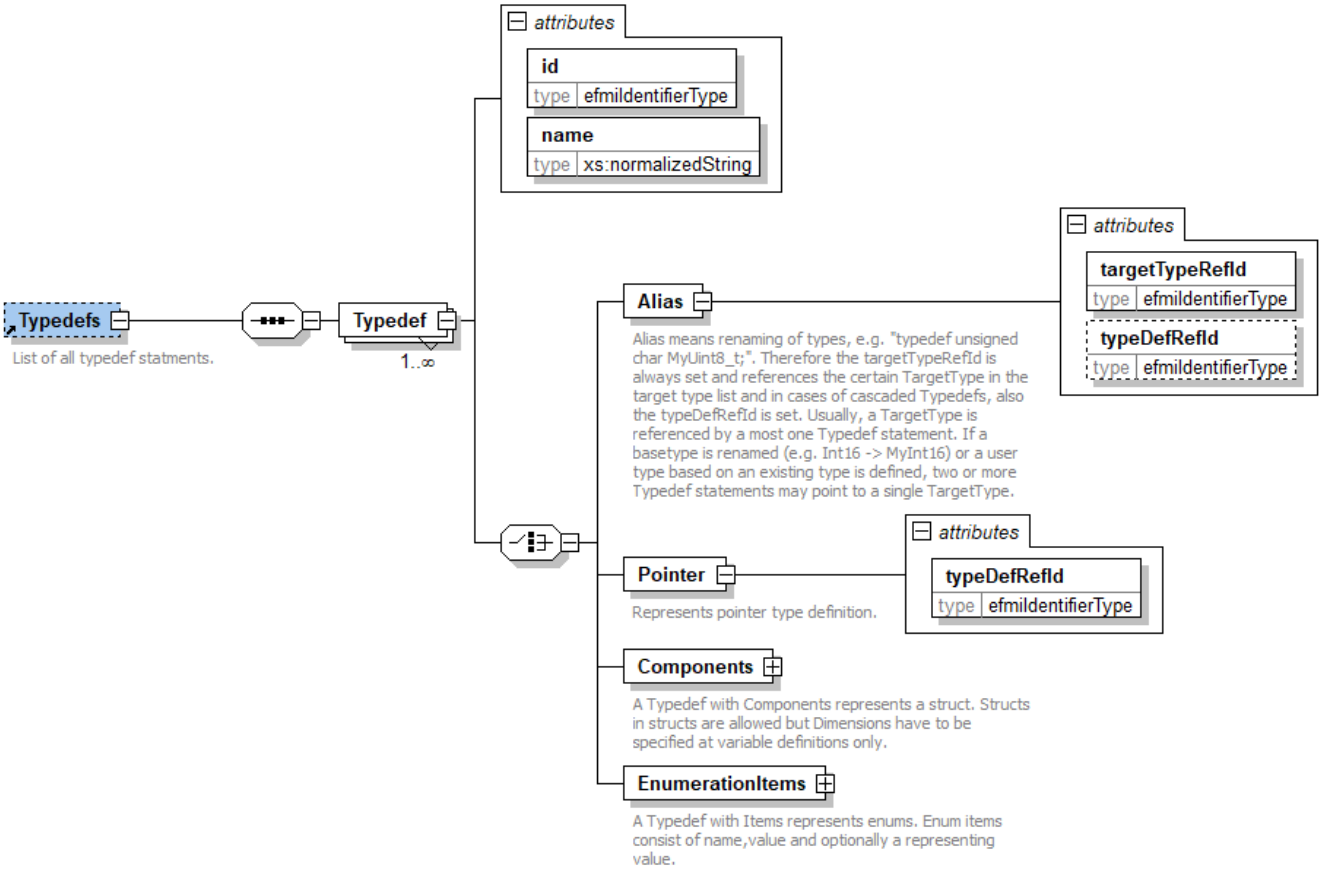
Name	Description
codeFileRefId	id of the included file. This attribute might be empty if the include is of a library.

Example:

```
<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 5.2.3.2.1.
`EnumerationItems`	Definition of an enum. For details see Section 5.2.3.2.2.

The following is an example of a simple alias declaration

Example:

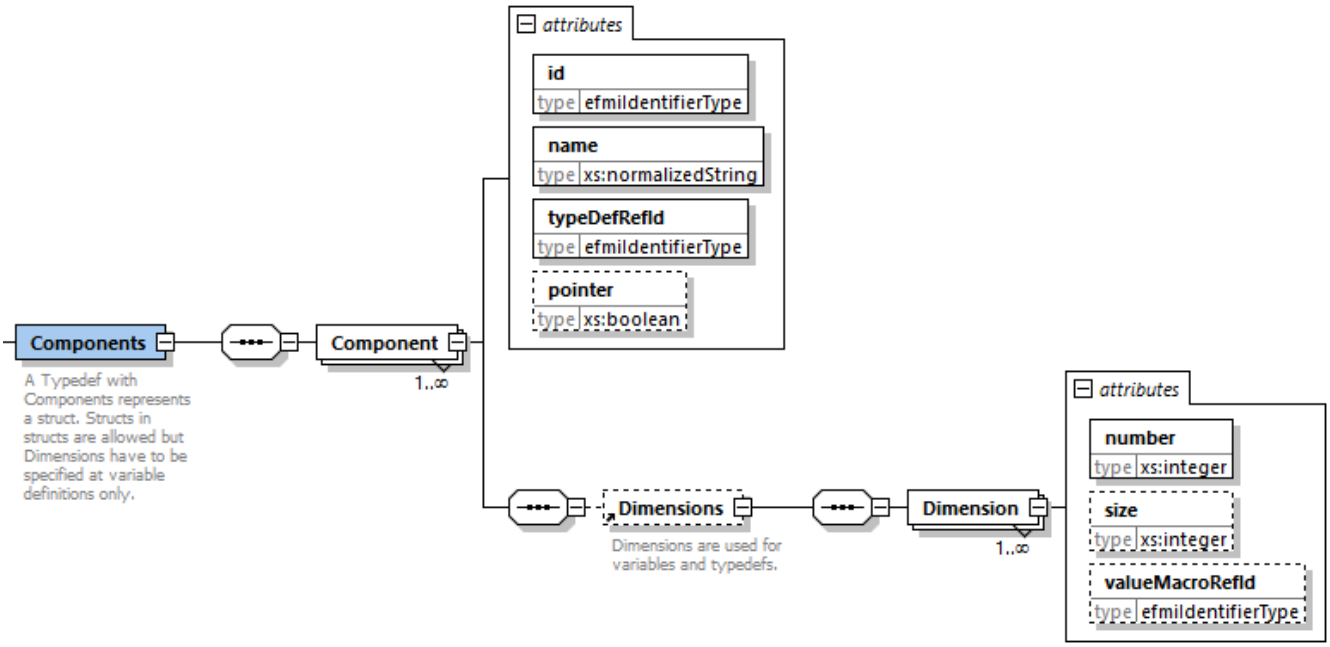
```
<Typedef name="Float32" id="TD_F32">
    <Alias targetTypeRefId="TT_float32" />
</Typedef>
```

The more complex data structure of function **spring** of the fourth example would be described by the following snipppet:

```
<Typedef name="spring" id="TD_spring">
    <Components>
      <Component id="C_1" name="deflection" typeRefId="TD_F32" pointer="true">
      <Component id="C_2" name="rigidity" typeRefId="TD_F32">
    <Components>
    <Alias targetTypeRefId="TT_float64" />
</Typedef>
```

Components (struct)

`Components` declare a structure and are a list of `Component`:

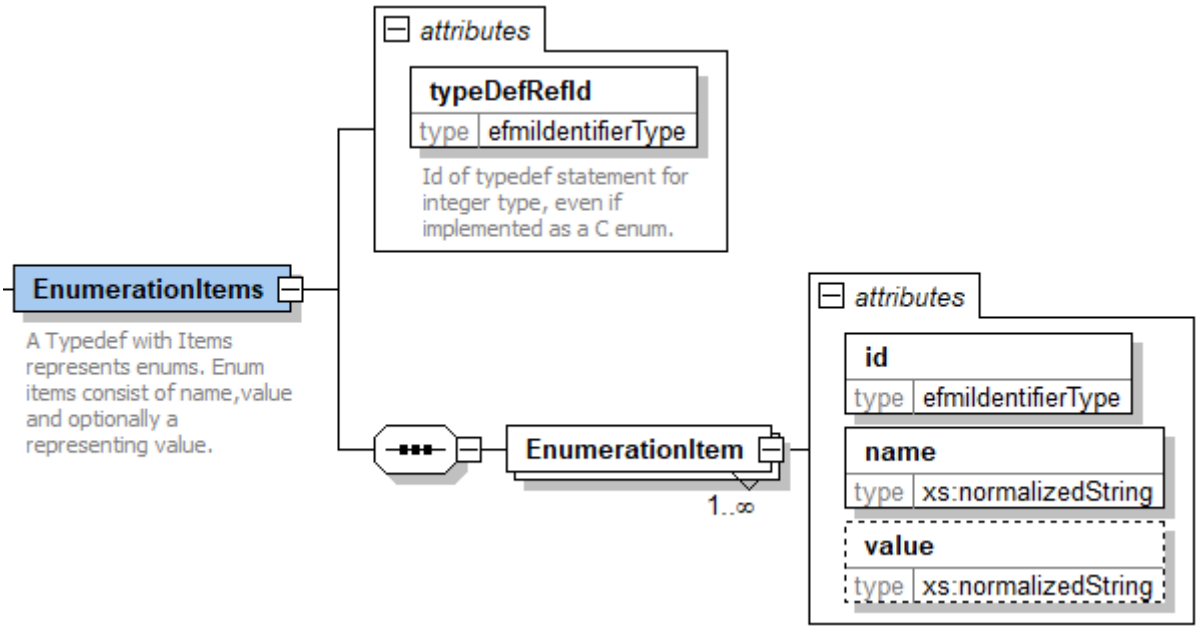


Name	Description
`id`	Unique id.
`Name`	Name of the field. Must be unique within one `<Components>` tag.
`typeDefRefId`	Reference of the type of the field.
`pointer`	Boolean flag on whether the field is a reference or not (optional field).

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.
`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)



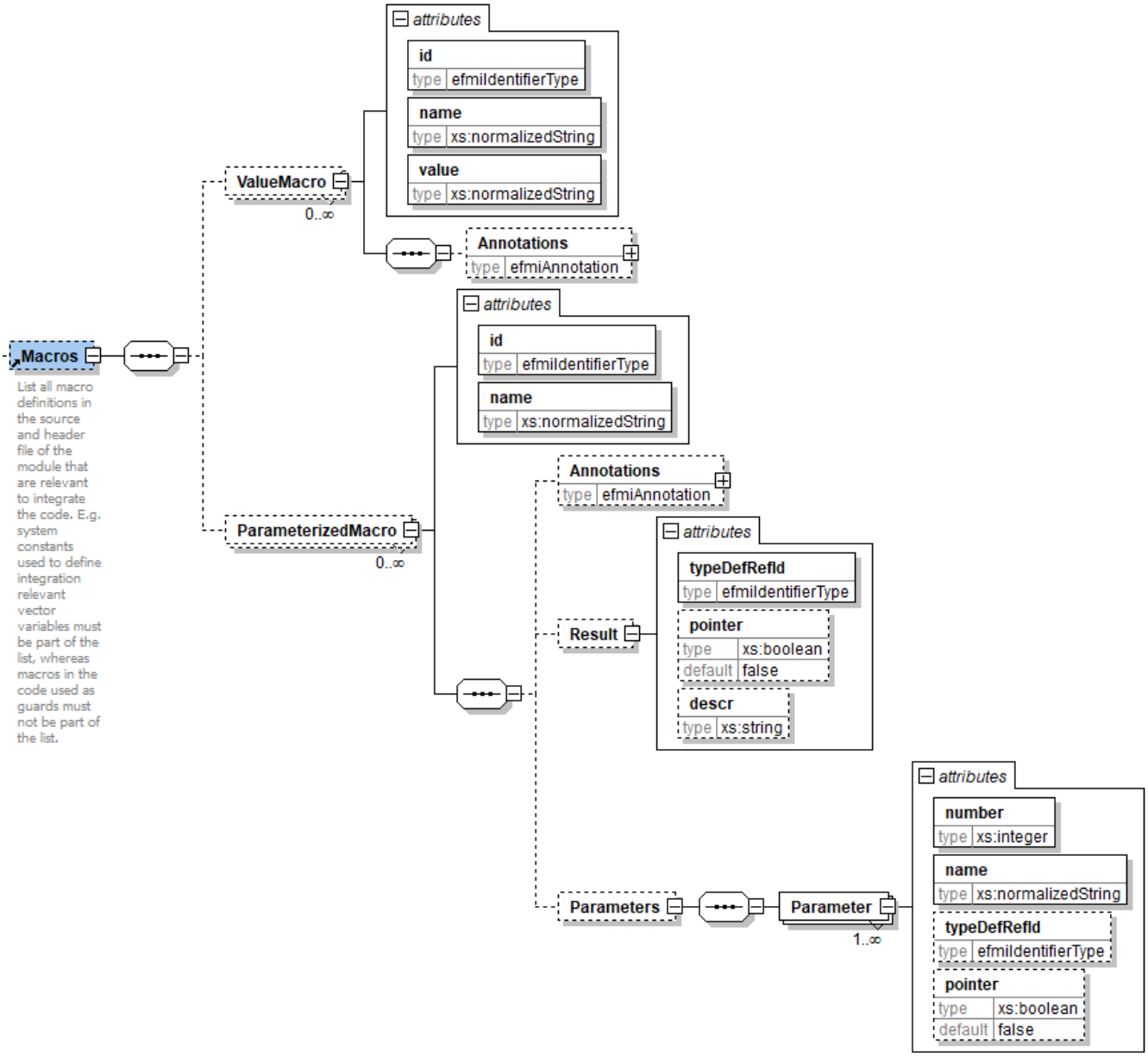
`<EnumerationItems>` declares an enumeration type with the list of enumeration items. Each `<EnumerationItem>` has the following fields

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.

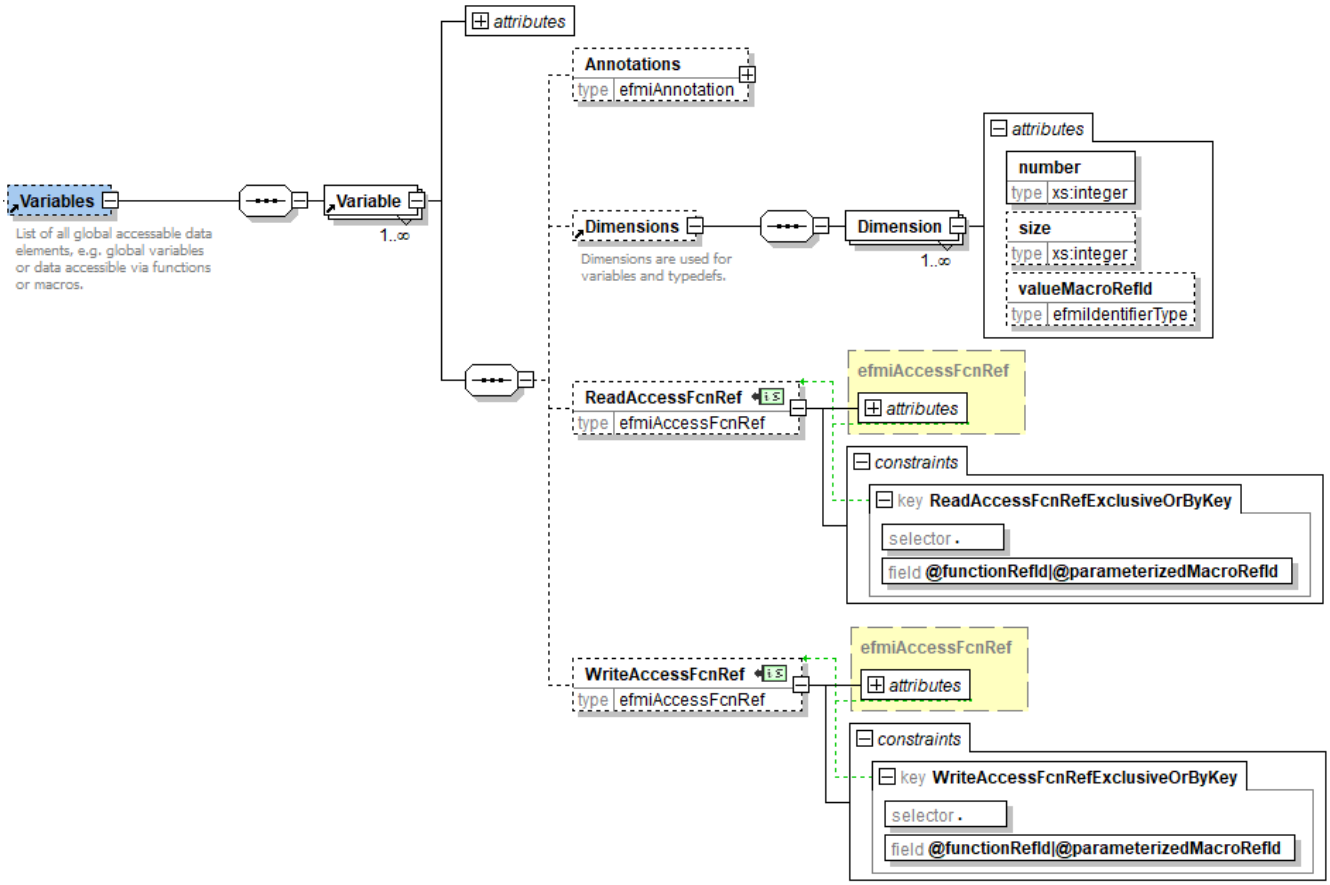
Name	Description
Number	Position of the macro argument.

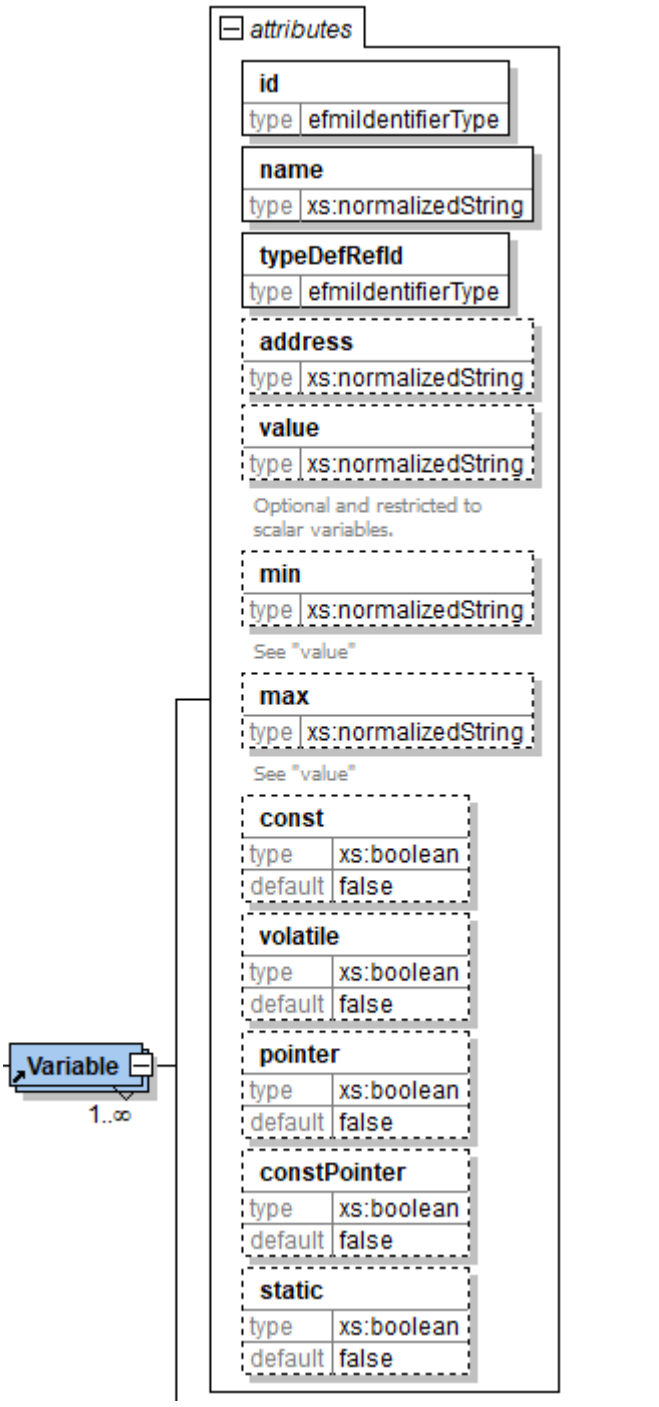
The following example shows the declaration of a value and a parametrized macro

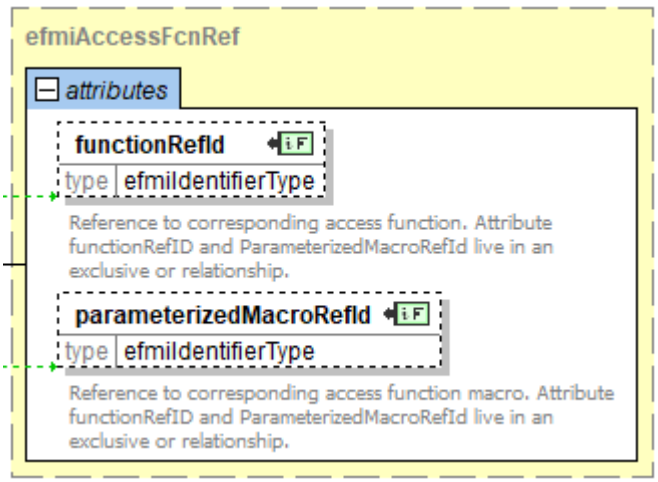
```
<Macros>
    <ValueMacro id="VM_1" name="num_Cyl" value="4"/>
    <ParameterizedMacro id="PM_1" name="myMax">
          <Parameter name="a" number="0">
          <Parameter name="b" number="1">
    </ParameterizedMacro>
</Macros>
```

Variables

<Variable> elements are grouped in the <Variables> element.







Each variable has the following attributes:

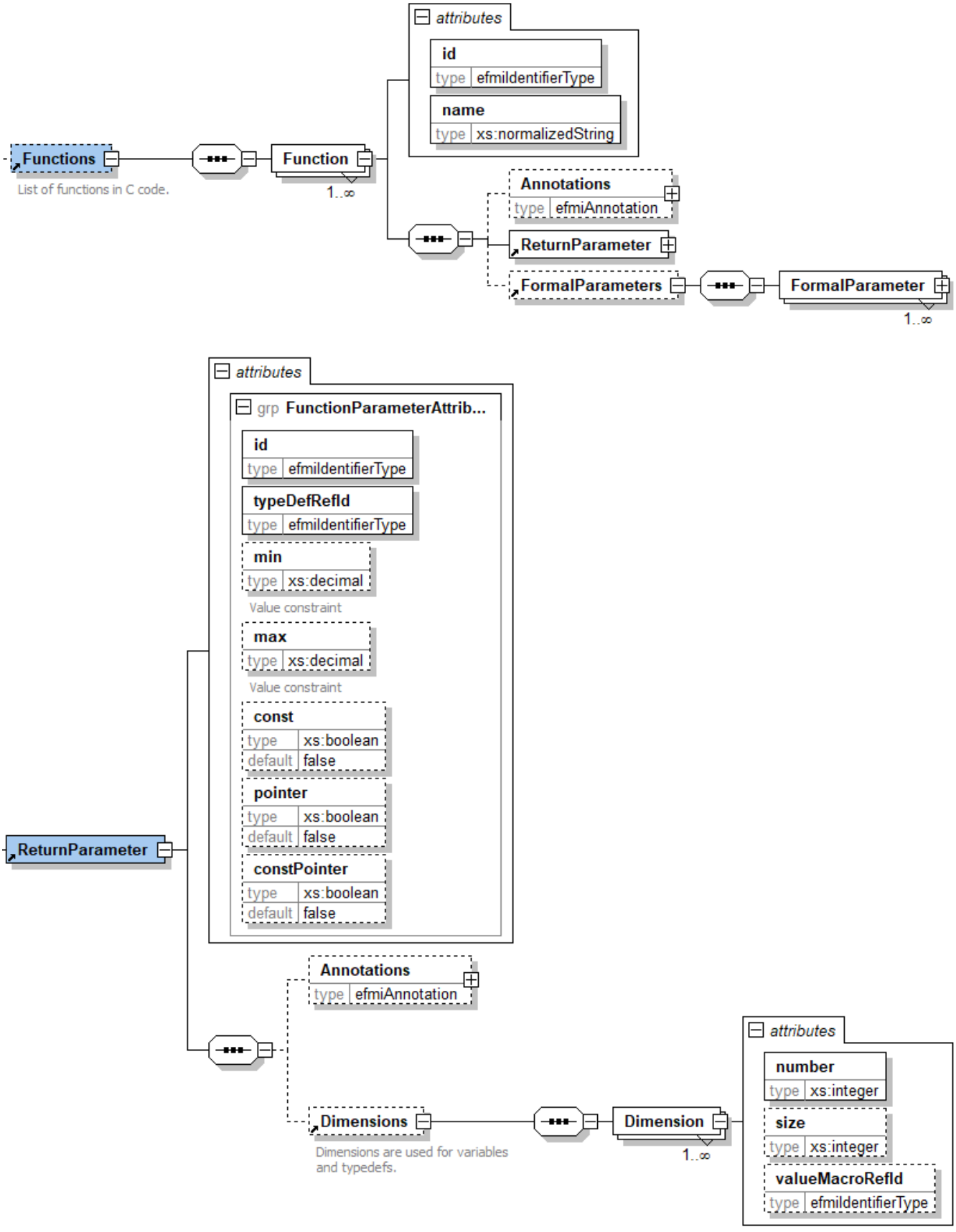
Name	Description
id	Unique id of the variable.
name	Name of the variable.
typedefRefId	id of the defined type of the variable.
address	Optional address.
value	Optional initial value of that variable that must be consistent which the initial value in Algorithm Code. Value might be different because of 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.

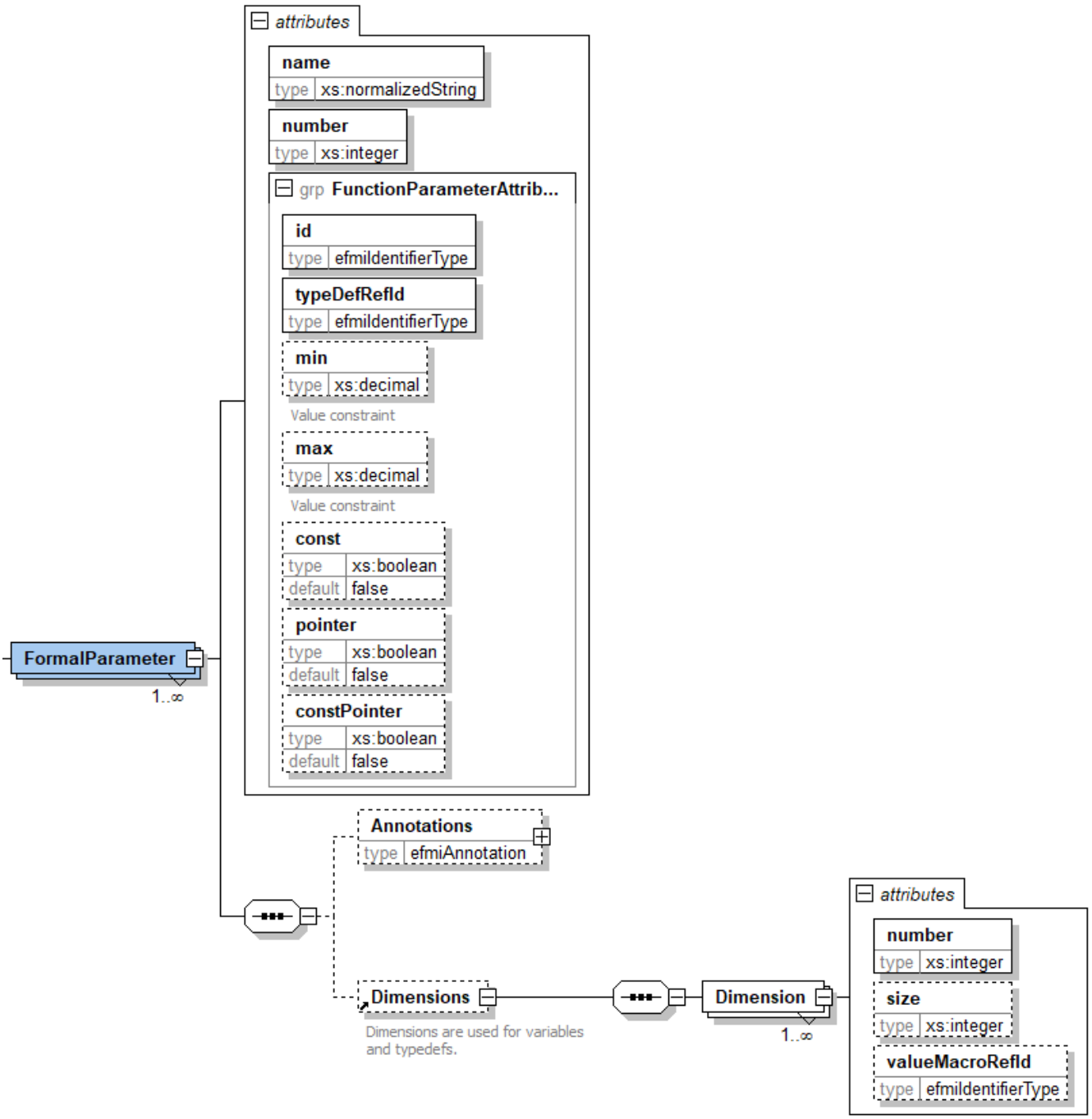
Similar like a field in a <Component> a <Variable> can also be multidimensional by adding the <Dimensions> element. The following example defines a 2x2 array of variables with name "T".

```
  <Variable id="V_33" name="T" typeDefRefId="TD_F64" pointer="false" value="0.1"
const="false" volatile="true" static="false">
     <Dimensions>
        <Dimension number="0" size="2">
        <Dimension number="1" size="2">
     </Dimensions>
   </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.


attributes
id
type efmildentifierType
name
type xs:normalizedString
Functions
List of functions in C code.
Function
1..∞
Annotations
type efmiAnnotation
ReturnParameter
FormalParameters
FormalParameter
1..∞
attributes
grp FunctionParameterAttrib...
id
type efmildentifierType
typeDefRefId
type efmildentifierType
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
ReturnParameter
Annotations
type efmiAnnotation
attributes
number
type xs:integer
size
type xs:integer
valueMacroRefId
type efmildentifierType
Dimensions
Dimensions are used for variables and typedefs.
Dimension
1..∞

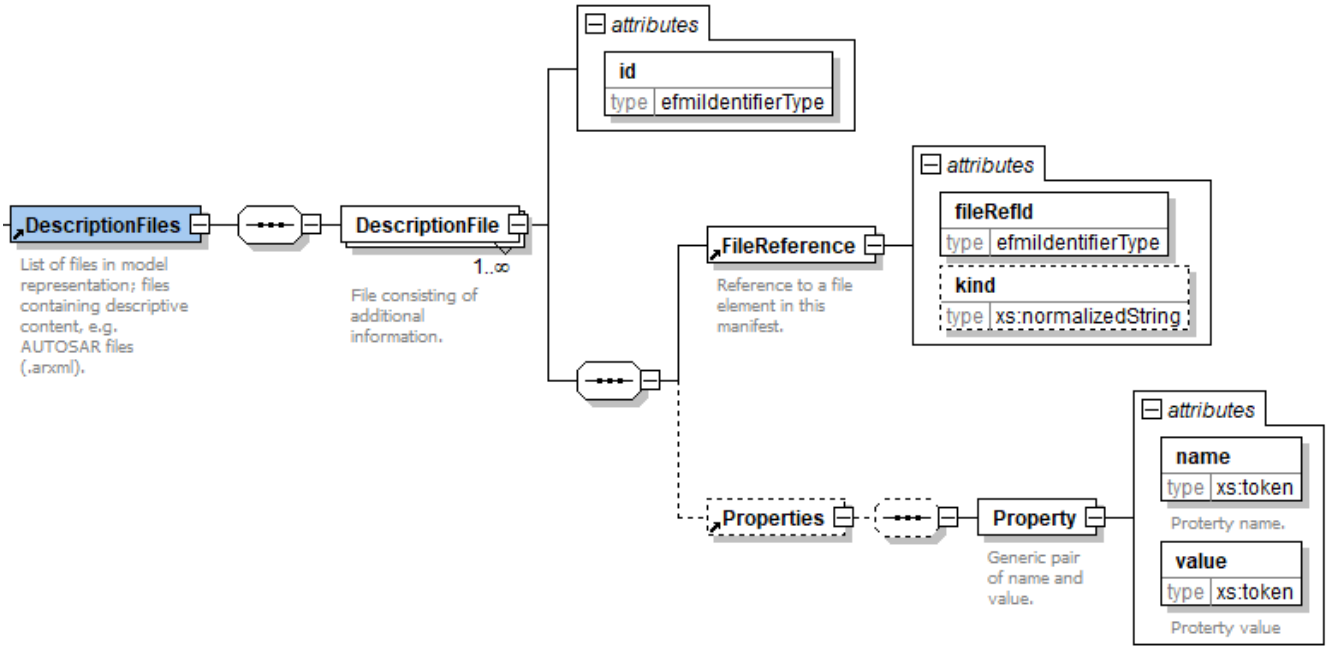


Example:

```
<Functions>
    <Function id="Func_1" name="doStep">
        <FormalParameters>
            <FormalParameter id="V_33" name="T" number="0" typeDefRefId="TD_F64">
        </FormalParameters>
    <Function/>
    <Function id="Func_2" name="doStep2">
        <ReturnParameter id="Func_2_ret" typeDefRefId="TD_F64" pointer="false">
    <Function/>
<Functions/>
```

5.2.4. Description Files

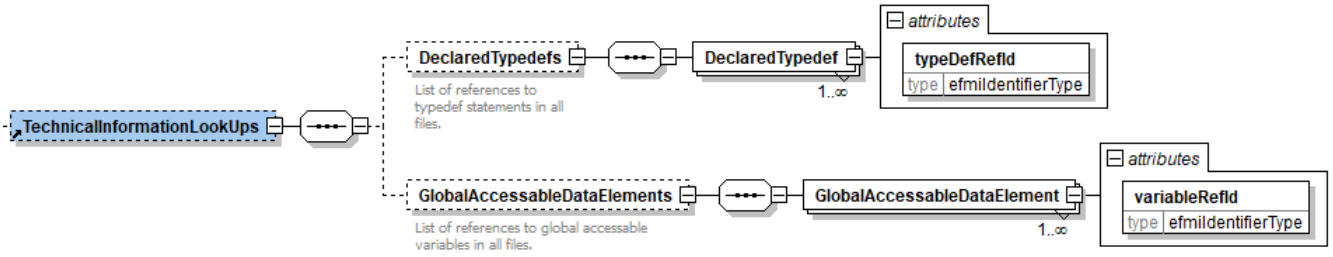
List of files containing descriptive content, for example AUTOSAR files (.arxml). Those files are the alternative to the detailed code description by e.g. typedefs, variables, etc. Usually all kinds of description files are allowed, but as they are used as alternative to the detailed description, elements that should be mapped to elements in the algorithm code manifest must be uniquely identifyable, e.g. they must have identifiers that are unique within a file, similar to identifiers used in manifests, or reachable by a given path expression.



Technically, a DescriptionFile has a `FileReference` pointing to a file in the manifest's file list and additional optional `Properties` as property value list.

5.2.5. 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:

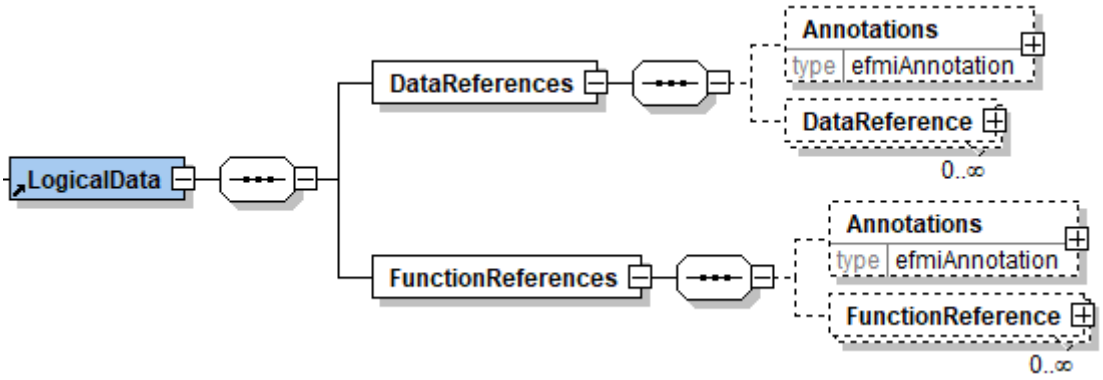
Name	Description
variableRefId	Reference to a Variable element in the manifest.

5.2.6. Logical Data

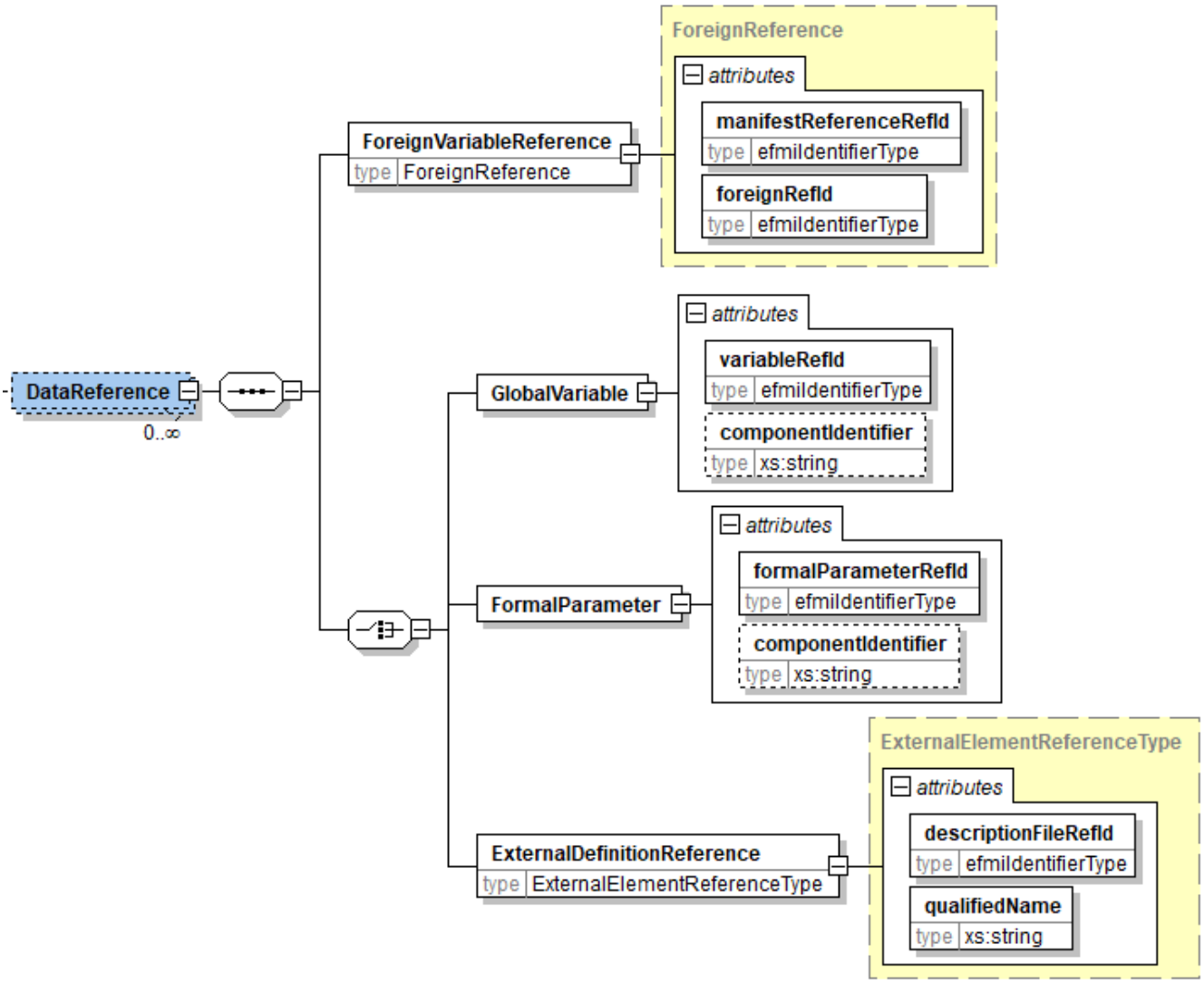
Defines how the logical elements (variables, functions etc.) are mapped to the actual data structures and elements of functions and defined variables.

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.

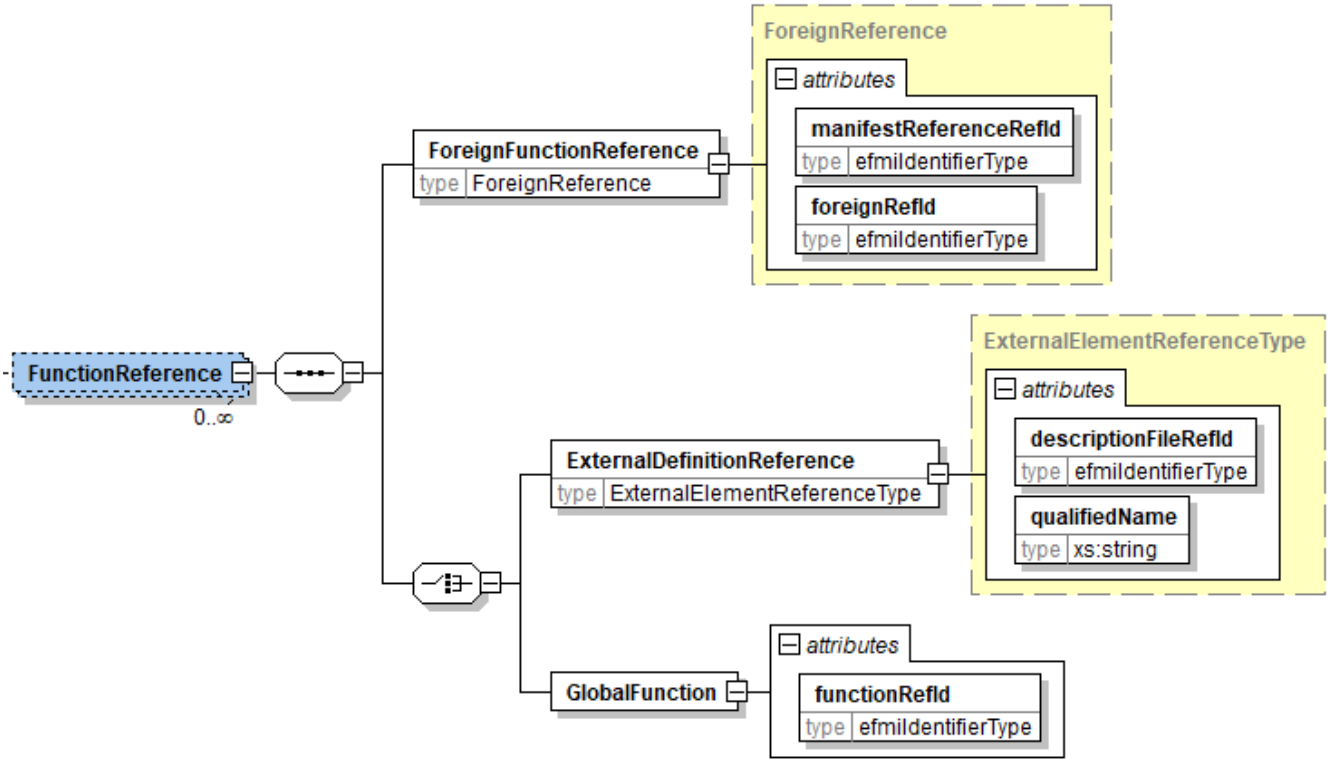


A DataReference itself contains the following attributes and elements to identify the variable in the Production Code and the mapped variable in the Algorithm Code



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".
`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".
`ExternalDefinitionReference`	Reference to an item by the `qualifiedName` attribute inside a referenced description file by a `descriptionFileRefId` attribute.

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.
`ExternalDefinitionReference`	Reference to an item by the `qualifiedName` attribute inside a referenced description file by a `descriptionFileRefId` attribute.

5.3. 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 [KR79] 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

6.1. Introduction

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.

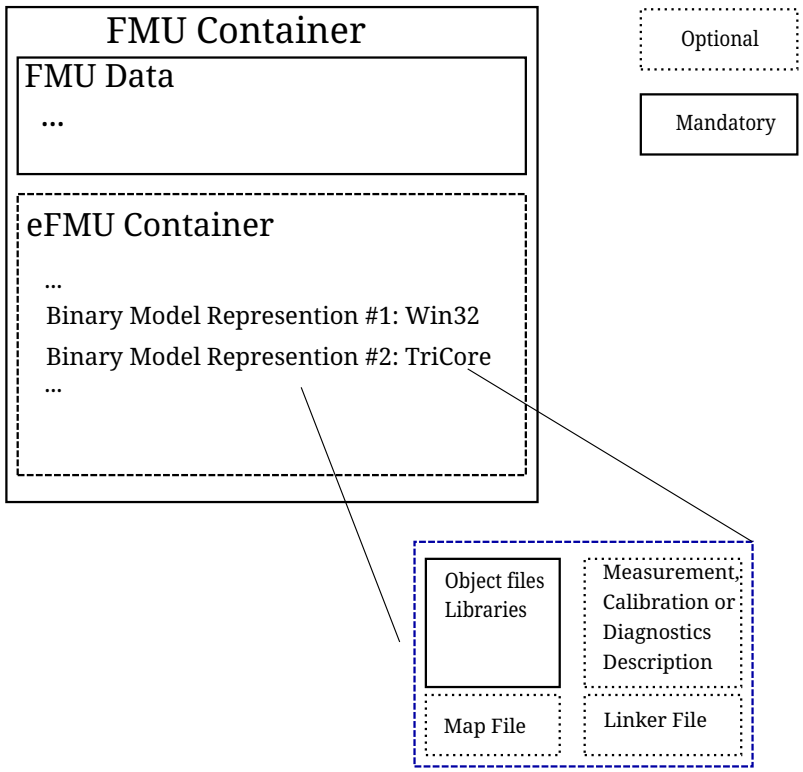


Figure 1. Structure of Binary Model Represention

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.

6.2. 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

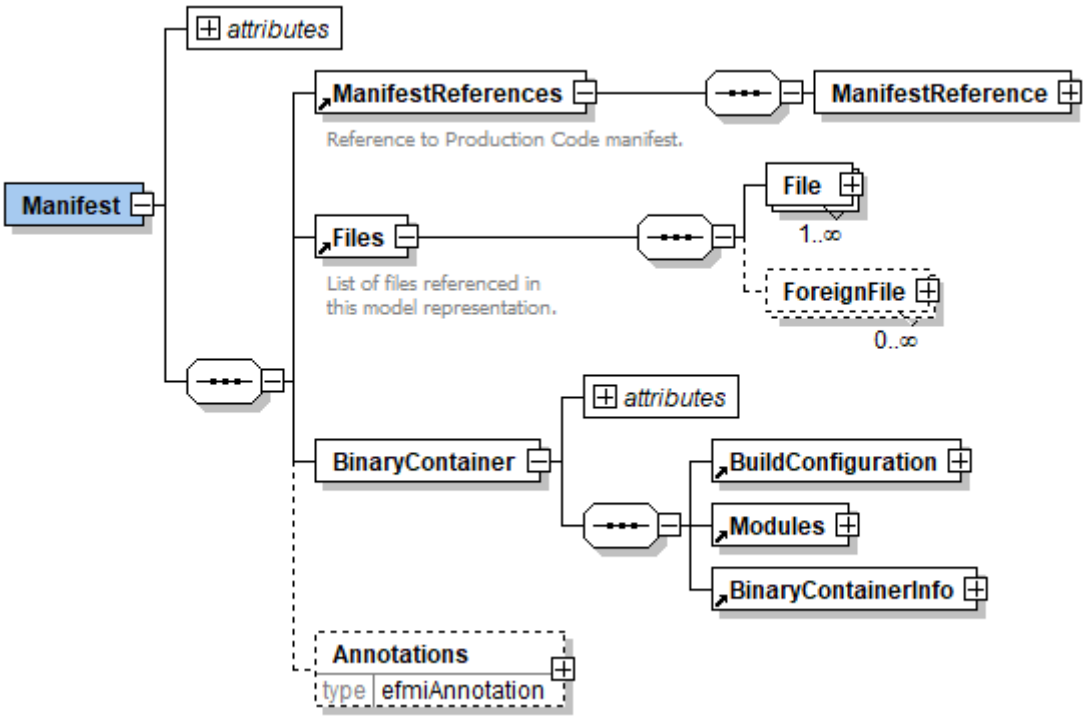
- information about the run time behavior
- meta information regarding the source code (e.g. MISRA Compliance, Code Quality reports, etc.)
- Calibration

The Binary Code manifest is an XML file with structured information about the Binary Model Representation.

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.

6.2.1. Structure of the Manifest

The Binary Code manifest:



consists of the following elements:

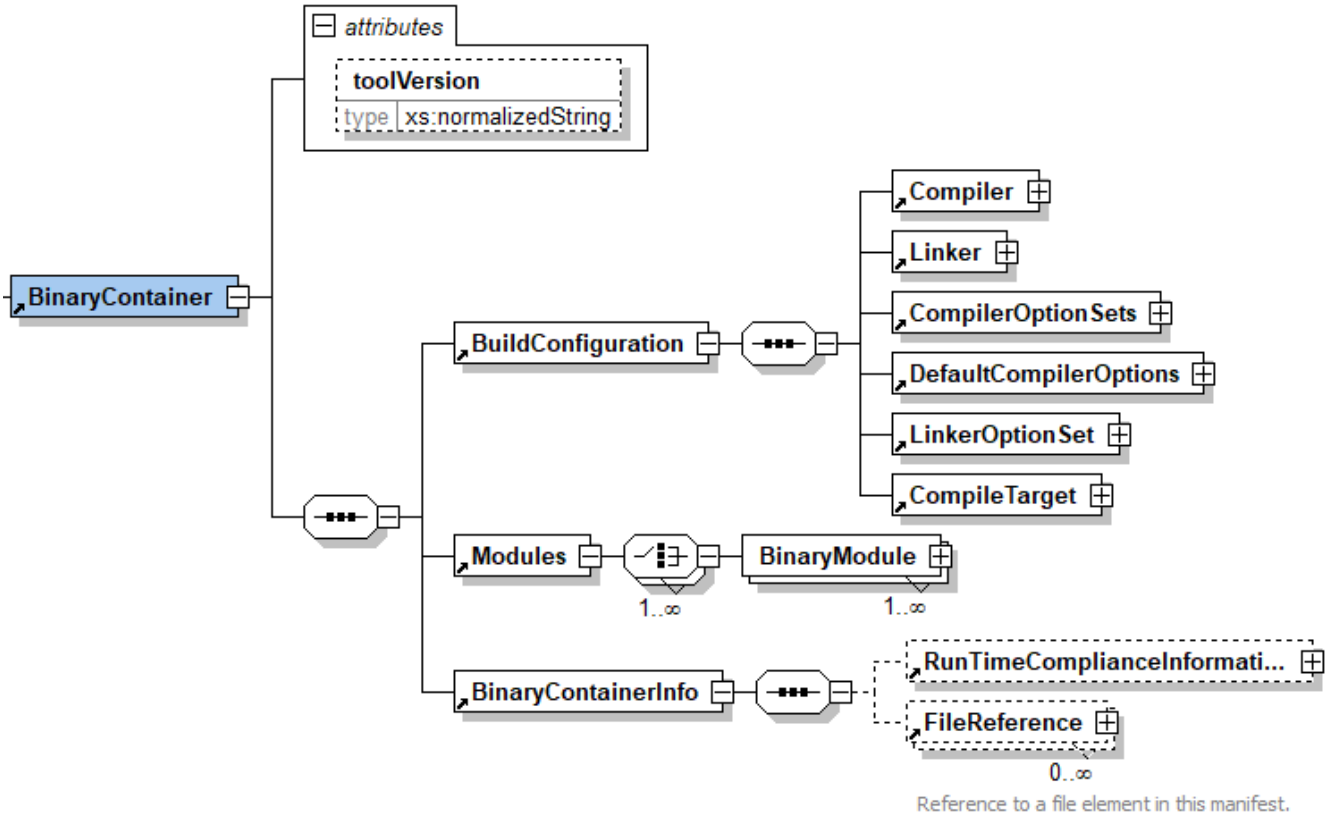
On the top level, the schema consists of the following elements:

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 6.2.2.
`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.

6.2.2. Binary Container

Element `BinaryContainer`



consists of the following elements:

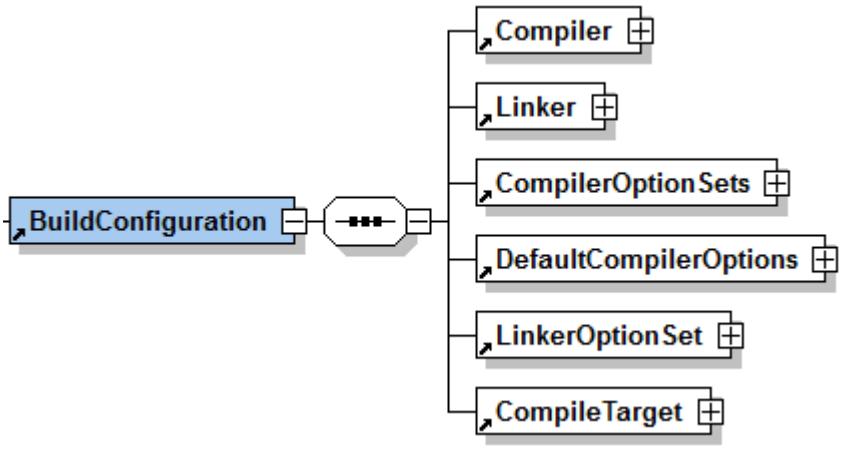
Name	Description
`BuildConfiguration`	The `BuildConfiguration` describes the actual build environment used to create the binary objects in the container. For more details see Section 6.2.2.1.
`Modules`	The `Modules` section describes all relevant binaries and source code references required or available for the binary model representation container. For more details see Section 6.2.3.
`BinaryContainerInfo`	The `BinaryContainerInfo` element contains additional and optional information relevant to the end user. For more details see Section 6.2.4.

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:



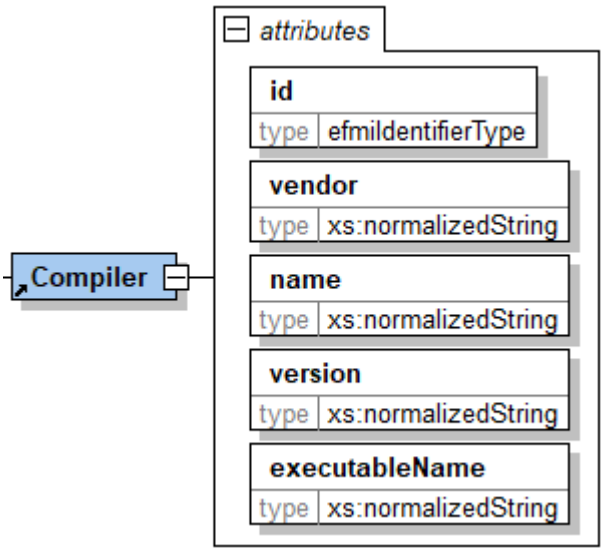
This element contains *exactly one* of each of the following elements:

Name	Description
`Compiler`	This element unambigously describes the compiler that has been used to create the binary artifacts. For details see Section 6.2.2.2.
`Linker`	This element unambigously describes the linker that has been used to create the binary artifacts. For details see Section 6.2.2.3.
`CompilerOptionSets`	This element stores all possible compiler settings used to create any binary element in the container. For details see Section 6.2.2.4.
`DefaultCompilerOptions`	This element refers to a `CompilerOptionSet` that has to be used to create the binary. For details see Section 6.2.2.5.
`LinkerOptionSet`	This element describes the relevant linker option for the above linker that has been used to create the binary object. For details see [definition-of-linker-option-set].
`CompileTarget`	This element describes the target platform, the binary has been compiled for. For details see [definition-of-compile-target].

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).

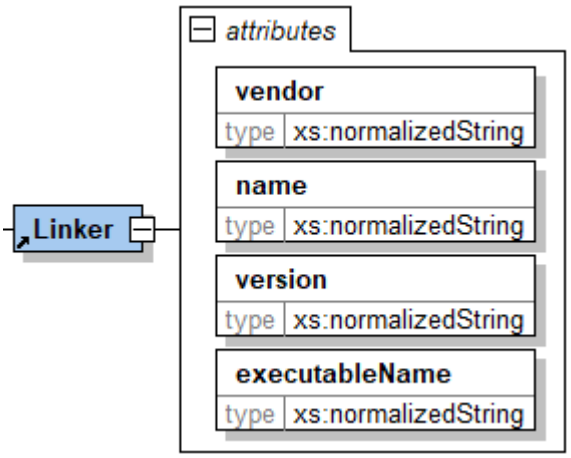
The attributes `vendor`, `name` and `version` must clearly identify a particular compiler. Furthermore, it should be possible to use the value `executableName` together with a matching `CompilerOptionSet` to automatically compile a source file.

The following example depicts a compiler configuration for a target compiler for the TriCore processor archtecture.

```
<Compiler id="ID_1000001" vendor="Altium"
          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.
`executableName`	The name of the actual executable of the linker (suite).

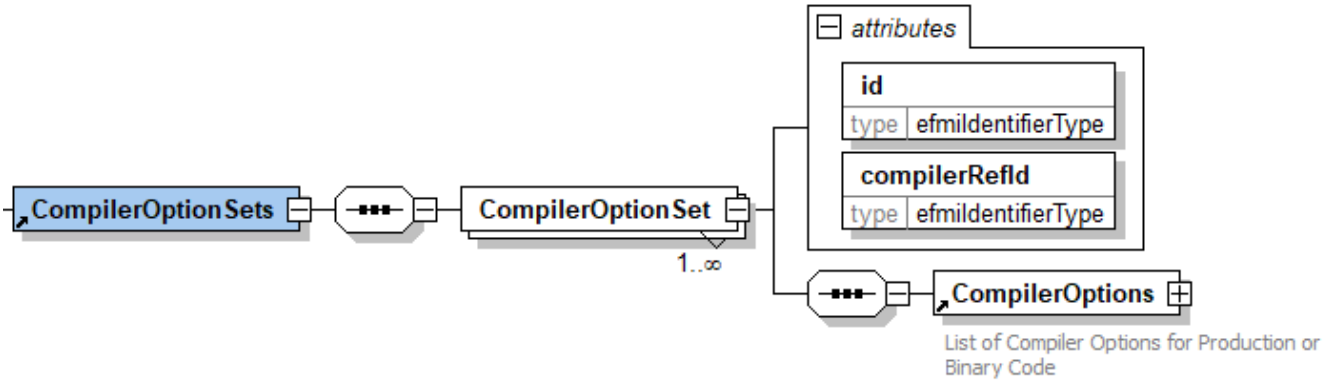
The attributes `vendor`, `name` and `version` must clearly identify a particular linker. Furthermore, it should be possible to use the value `executableName` together with the below defined `LinkerOptionSet` to automatically link object files together.

The following example depicts an linker configuration for the TriCore processor architecture.

```
<Linker id="ID_1000002" vendor="Altium" name="TASKING VX-toolset for TriCore: object
linker" version="v4.2r2" executableName="ltc"/>
```

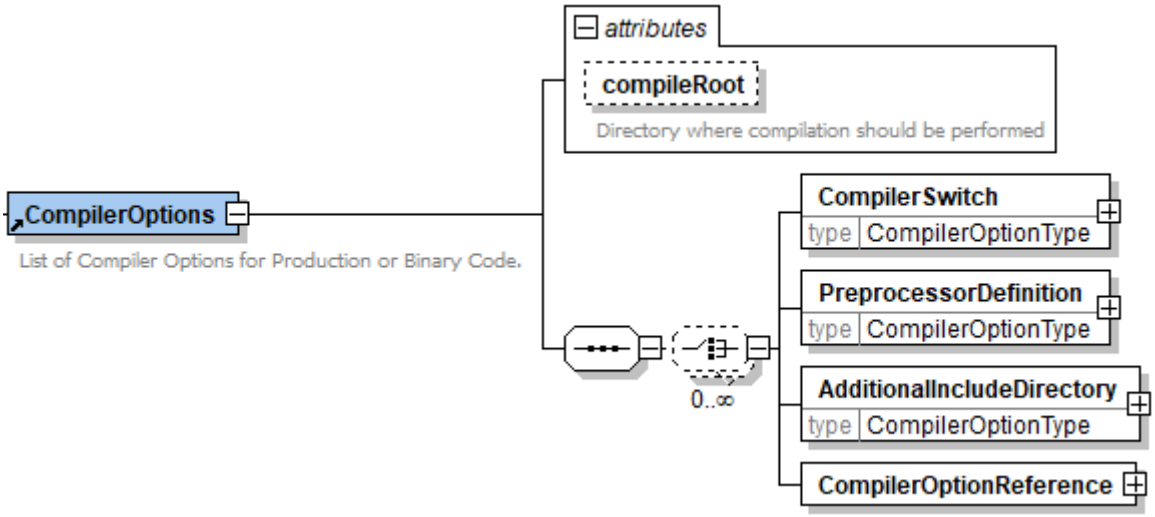
CompilerOptionSets

The `CompilerOptionSets` contains one or more `CompilerOptionSet` which defines settings and switches used to create at least one of the contained binary artifacts.



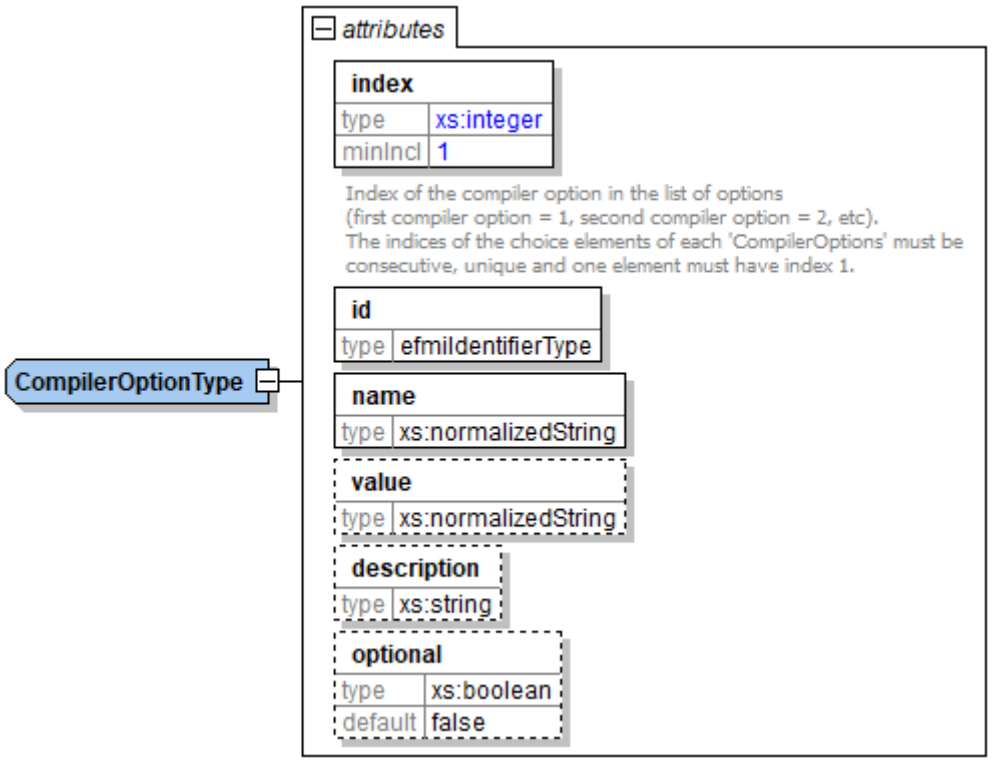
Name	Description
`id`	The unique identifier of the the `CompilerOptionSet` within the manifest.
`compilerRefId`	A reference to a configured compiler for the Compilers Section.
`CompilerOptions`	List of compiler options for Production or Binary Code, see [CompilerOptions]

The `CompilerOptions` list is defined as:



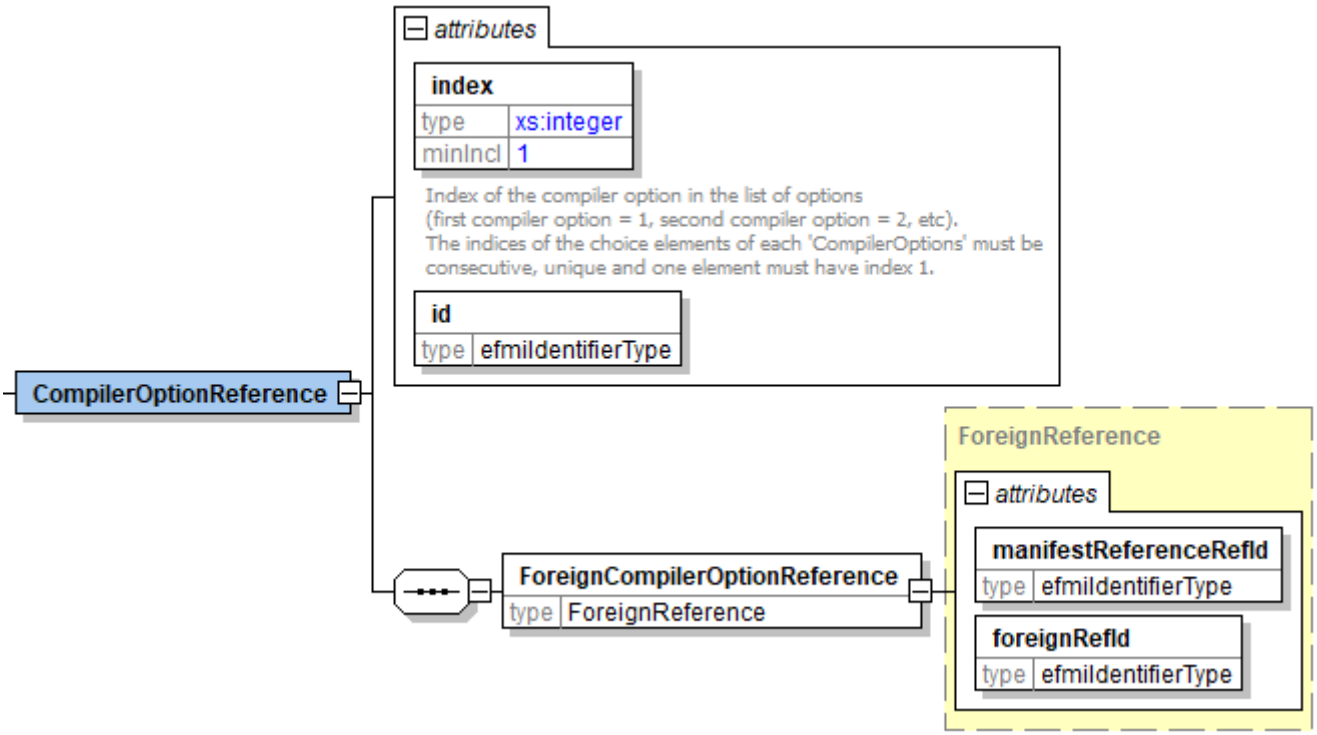
Name	Description
`compileRoot`	Directory where compilation should be performed.
`CompilerSwitch`	The compiler switches of type [CompilerOptionType].
`PreprocessorDefinition`	Preprocessor definitions of type [CompilerOptionType].
`AdditionalIncludeDirectory`	Additional include directory of type [CompilerOptionType].
`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.

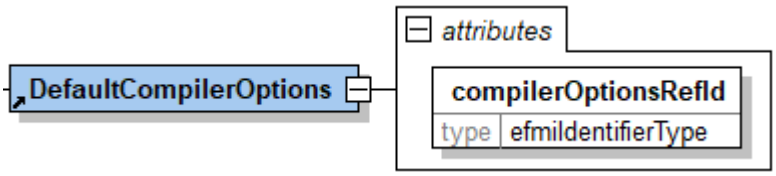
The following example depicts some of the options that have to be provided in order to compile code for the Infineon Tricore TC27x family. Most options are special to this compiler family.

```
<CompilerOptionSets>
  <CompilerOptionSet id="ID_1001" compilerRefId="ID_1000001">
    <CompilerOptions>
      <CompilerSwitch>
        <id>ID_100010</id>
        <name>--iso</name>
        <value>90</value>
      </CompilerSwitch>
      <CompilerSwitch>
        <id>ID_100011</id>
        <name>--align</name>
        <value>4</value>
      </CompilerSwitch>
      <CompilerSwitch>
        <id>ID_100012</id>
        <name>--optimize</name>
        <value>3</value>
      </CompilerSwitch>
      <CompilerSwitch>
        <id>ID_100013</id>
        <name>--tradeoff</name>
        <value>4</value>
      </CompilerSwitch>
      <CompilerSwitch>
        <id>ID_100014</id>
        <name>--source</name>
      </CompilerSwitch>
      <CompilerSwitch>
        <id>ID_100015</id>
        <name>--error-file</name>
      </CompilerSwitch>
      <CompilerSwitch>
        <id>ID_100016</id>
        <name>--rename-sections=sect</name>
      </CompilerSwitch>
      <CompilerSwitch>
        <id>ID_100017</id>
        <name>--core</name>
        <value>tc1.6.x>
      </CompilerSwitch>
      <CompilerSwitch>
        <id>ID_100018</id>
        <name>-Hsfr/regtc27x.sfr</name>
      </CompilerSwitch>
      <CompilerSwitch>
        <id>ID_100019</id>
        <name>--default-near-size</name>
        <value>0</value>
      </CompilerSwitch>
<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:



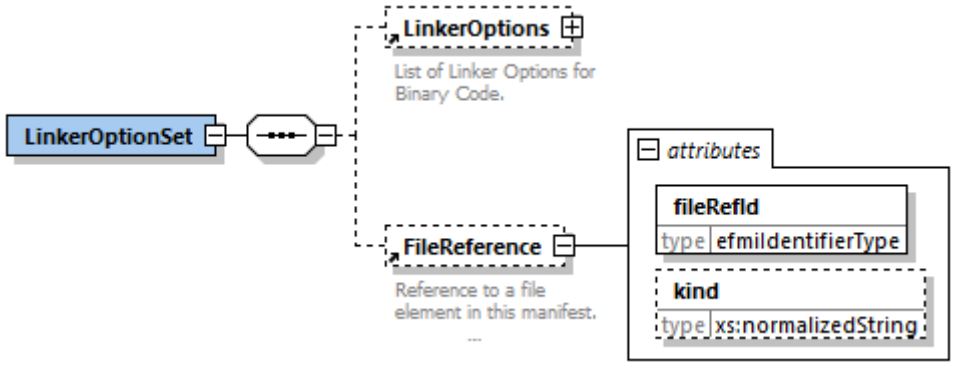
Name	Description
`compilerOptionsRefId`	Reference to a previously defined `CompilerOptionSet` to be used as default.

The following example depicts an default option set that refers to the `CompilerOptionSet` defined in the parent `BinaryContainer` element.

```
<DefaultCompilerOptions compilerOptionsRefId="ID_1001" />
```

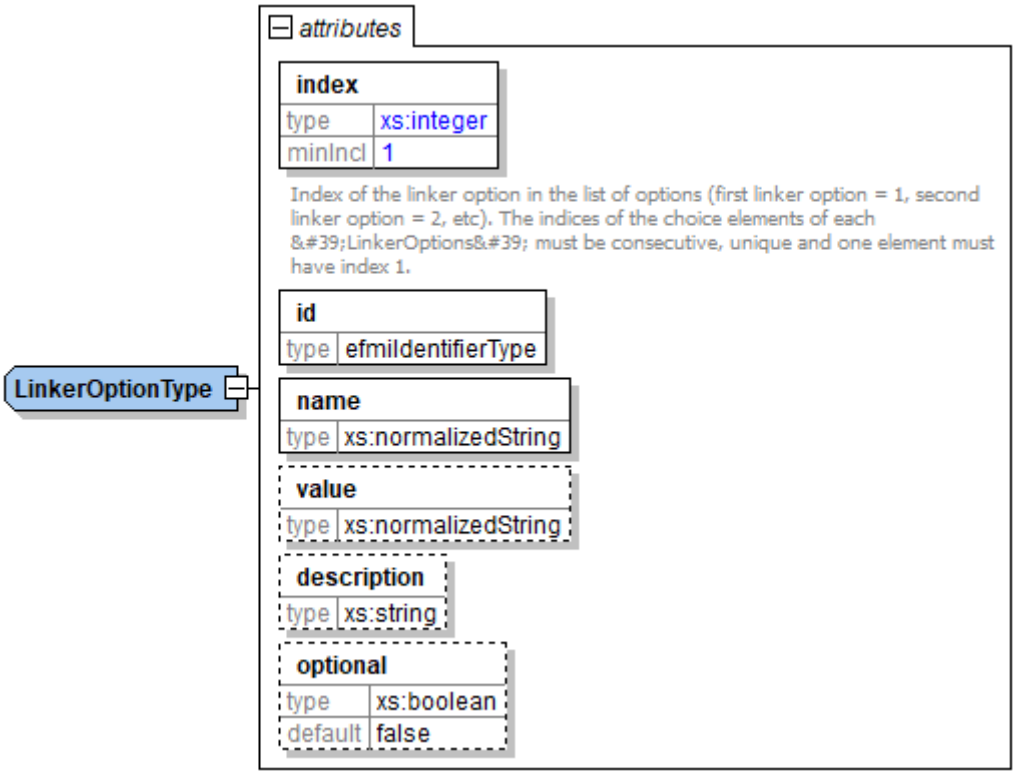
LinkerOptionSet

The `LinkerOptionSet` contains one `LinkerOptions` which defines linker settings and switches.



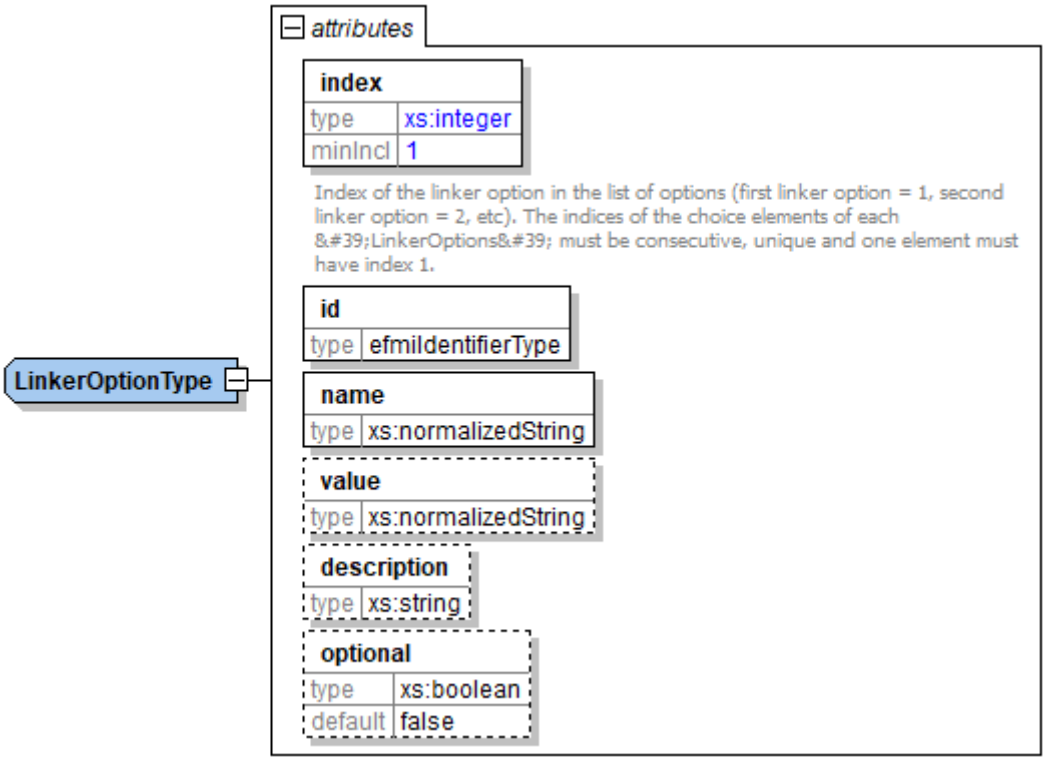
Name	Description
`LinkerOptions`	List of linker options for Production or Binary Code, see [LinkerOptions]
`FileReference`	The linker script is referenced with a `FileReference` element.

The `LinkerOptions` list is defined as:



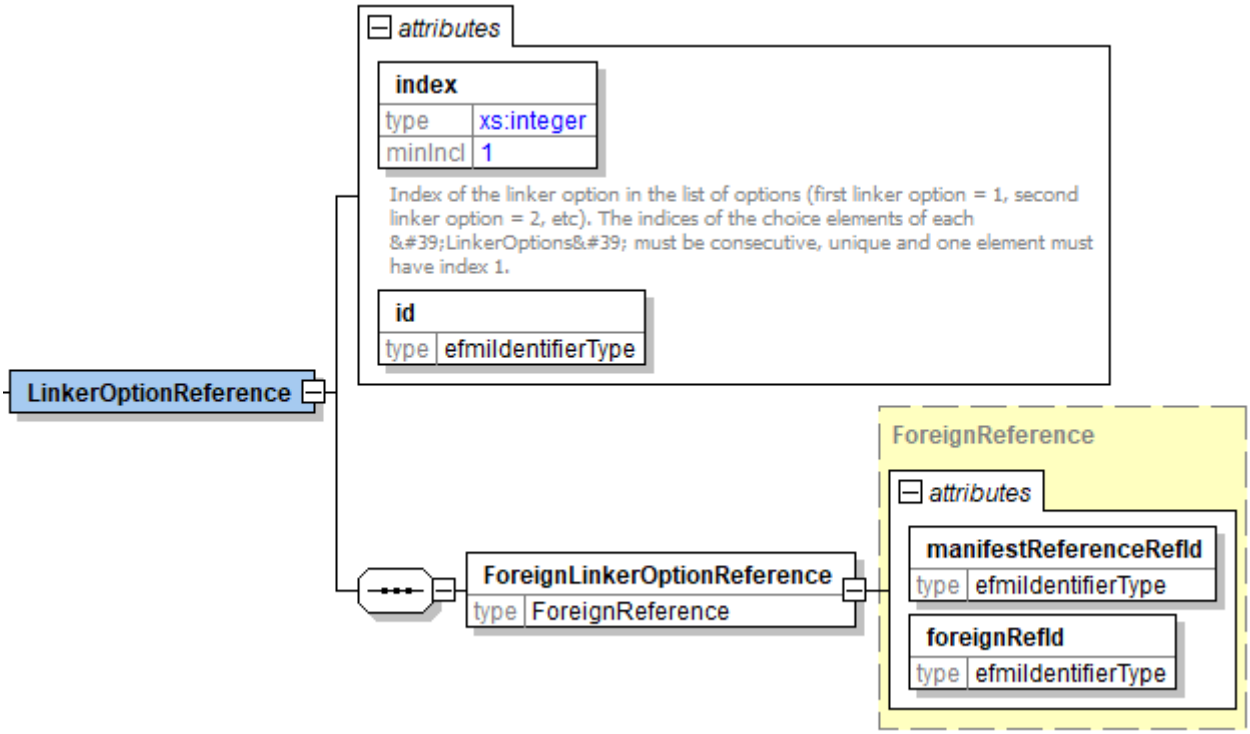
Name	Description
`LinkerSwitch`	The linker switches of type [LinkerOptionType].
`Library`	Library of type [LinkerOptionType].
`AdditionalLibraryDirectory`	Additional library directory of type [LinkerOptionType].
`LinkerOptionReference`	A list of option references, see [LinkerOptionReference].

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:



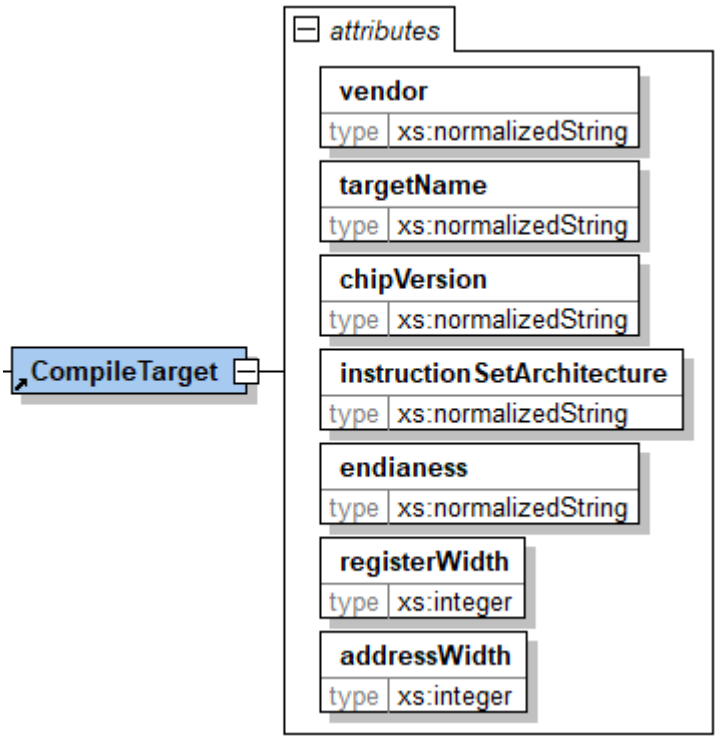
Name	Description
index	Index of the option in the linker command line.
id	Unique id of option reference.
ForeignOptionReference	Reference to another manifest file of type ForeignReference. For details see Section 2.3.4.3.

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.

```
<LinkerOptionSet>
  <LinkerOptions>
    <LinkerSwitch>
      <id>ID_100010</id>
      <name>output</name>
      <value>dummy.elf:ELF</value>
    </LinkerSwitch>
    <LinkerSwitch>
      <id>ID_100011</id>
      <name>no-warnings</name>
    </LinkerSwitch>
    <LinkerSwitch>
      <id>ID_100012</id>
      <name>incremental</name>
    </LinkerSwitch>
    <LinkerSwitch>
      <id>ID_100013</id>
      <name>lsl-file</name>
      <value>TC277.lsl</value>
    </LinkerSwitch>
    <LinkerSwitch>
      <id>ID_100014</id>
      <name>map-file</name>
      <value>mapfile.map</value>
    </LinkerSwitch>
  <LinkerOptions>
  <FileReference fileRefId="ID_999915" kind="LinkerScript" />
</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.

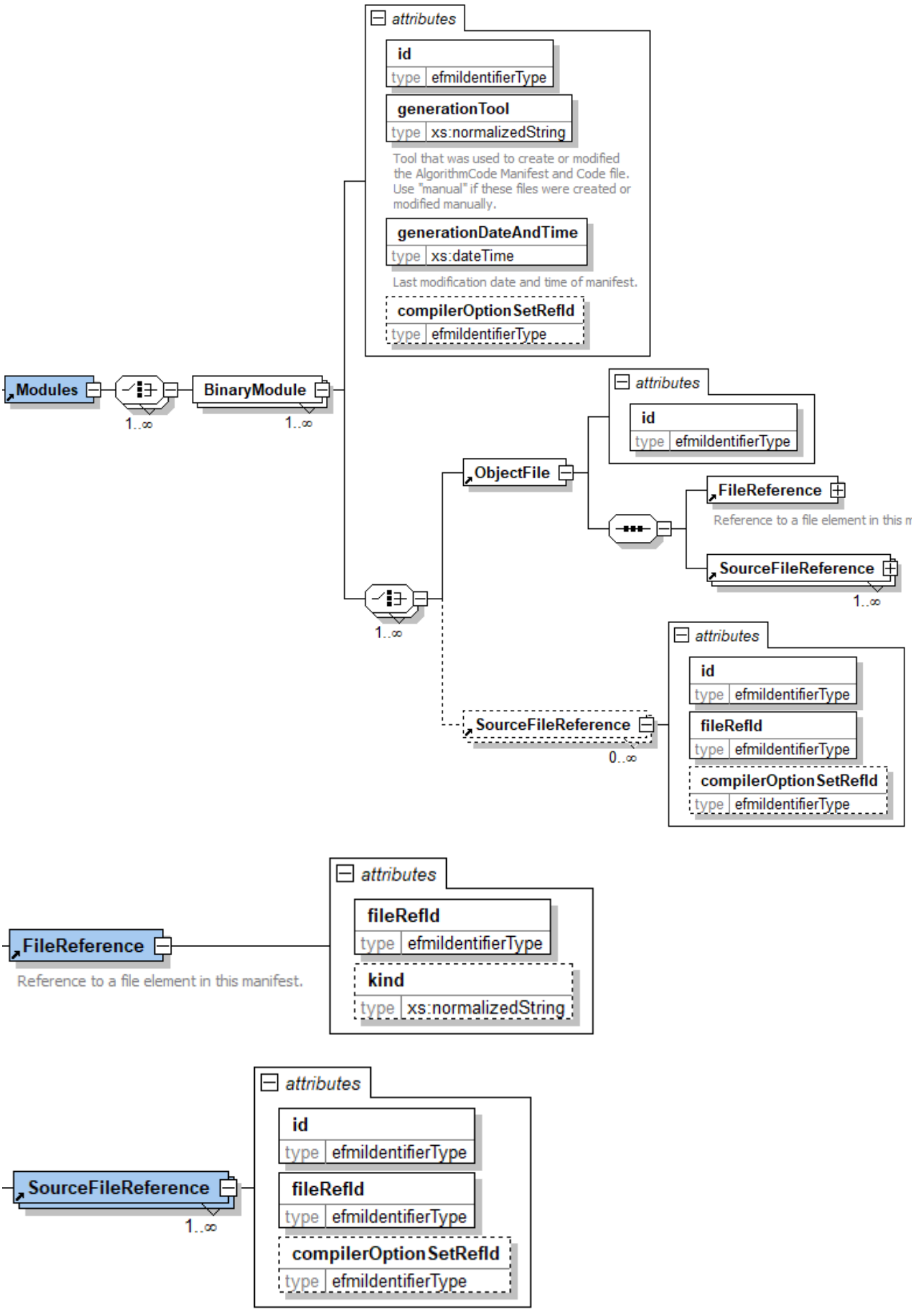
The following example depicts the target information needed for a TC277 Processor within a TriCore embedded target.

```
<CompileTarget id="ID_100001" vendor="Infineon" targetName="TriCore"
chipVersion="TC277 C-Step" instructionSetArchitecture="TC1.6E" endianess="LITTLE"
registerWidth="32" addressWidth="32"/>
```

6.2.3. 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.
`compilerOptionSetRefId`	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.
`SourceFileReference`	Each element refers to a code file in production Code manifest.

`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.
`CompilerOptionSetId`	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
`FileReference`	Reference to the actual binary object file. The `kind` of the `FileReference` is either *"RelocatableObjectFile"* or *"ExecutableObjectFile"*. This element is mandatory.

Name	Description
ForeignSourceFileReference	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.

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.

```
<ForeignFile id="ID_999920">
  <ForeignReference foreignRefId="ID_9" manifestReferenceRefId="ID_0000001" />
</ForeignFile>
<ForeignFile id="ID_999921">
  <ForeignReference foreignRefId="ID_10" manifestReferenceRefId="ID_0000001" />
</ForeignFile>
<ForeignFile id="ID_999922">
  <ForeignReference foreignRefId="ID_5" manifestReferenceRefId="ID_0000001"/>
</ForeignFile>
<ForeignFile id="ID_999923">
  <ForeignReference foreignRefId="ID_1" manifestReferenceRefId="ID_0000001"  />
</ForeignFile>
<ForeignFile id="ID_999924">
  <ForeignReference foreignRefId="ID_3" manifestReferenceRefId="ID_0000001"  />
</ForeignFile>
[...]
<Modules>
  <BinaryModule id="ID_4" creator="JDoe" creationDate="2018-08-09">
    <ObjectFile id="ID_10">
      <FileReference fileRefId="ID_01" kind="RelocatableObjectFile" />
      <SourceFileReference id="ID_02" fileRefId="ID_999920" />
      <SourceFileReference id="ID_03" fileRefId="ID_999921" />
    </ObjectFile>
    <SourceFileReference id="ID_5"fileRefId="ID_999922" />
    <SourceFileReference id="ID_1" compilerOptionSetRefId="ID_46"
fileRefId="ID_999923" />
    <SourceFileReference id="ID_3" compilerOptionSetRefId="ID_46"
fileRefId="ID_999924" />
  </BinaryModule>
</Modules>
```

6.2.4. Binary Container Info (optional)

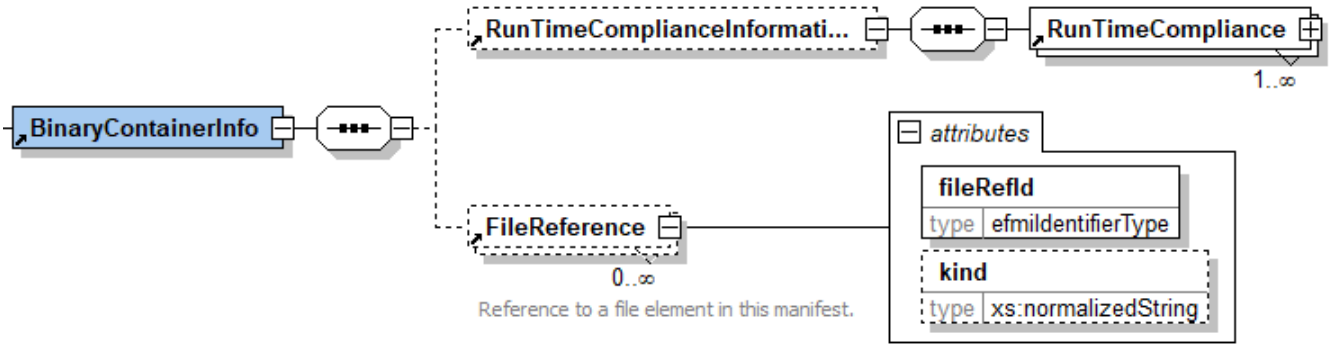
The previously described elements of the manifest for the Binary Code Model Representation are mandatory. However, there is also information that might not be necessary to describe a binary but very helpful in the actual use cases for the Binary Code Model Representation such as integration or

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.

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"`.

```
<File id="ID_999913" path="/objects/" name="SpeedController.map" role="other"
needsChecksum="true"
       checksum="A43C0994FAD1247988C2AA8A90CCA2E241CF5687" />
[...]
<BinaryContainerInfo>
  <FileReference fileRefId="ID_999913" kind="MapFile" />
</BinaryContainerInfo>
```

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.

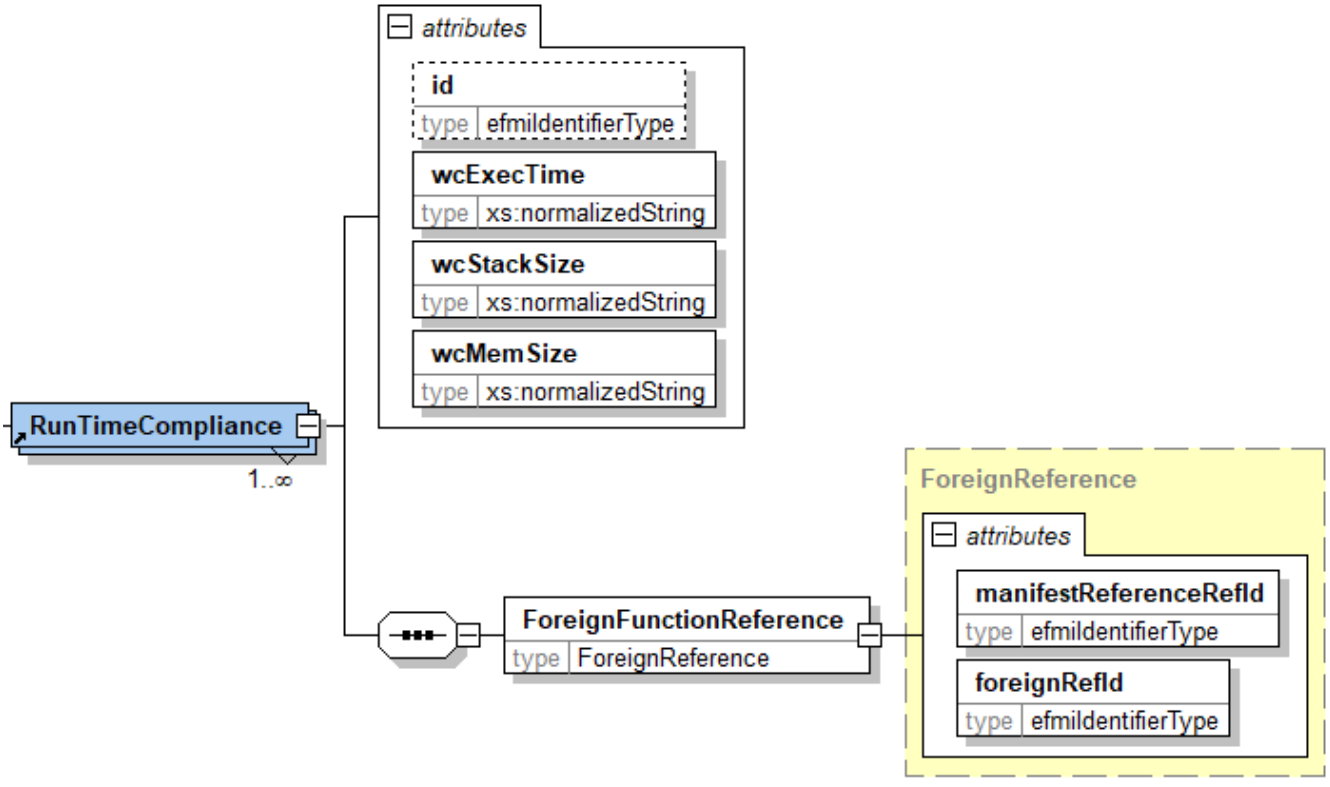
Run Time Behavior

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.

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.

The following example shows how the `RunTimeComplianceInformation` can be defined for some function.

```
<BinaryContainerInfo>
  <RunTimeComplianceInformation>
    <RunTimeCompliance id="ID_100301" wcExecTime="8.4ms" wcStackSize="70kb"
wcMemSize="840kb">
      <ForeignFunctionReference foreignRefId="ID_41"
manifestReferenceRefId="ID_0000001" />
    </RunTimeCompliance>
  </RunTimeComplianceInformation>
</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"`.

The following code snippet shows how a calibration file can be provided.

```
<File id="ID_999912" path="/" name="myFunction.a2l" role="other"
checksum="0DC09613F414FFCE10865AF3AD3EC31D3ED61EA8" needsChecksum="true" />
[...]
<BinaryContainerInfo>
  <FileReference fileRefId="ID_999912" kind="CalibrationInformationFile" />
</BinaryContainerInfo>
```

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"`.

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.

```
<File id="ID_999912" path="/" name="myFunction.a2l" role="other"
checksum="0DC09613F414FFCE10865AF3AD3EC31D3ED61EA8" needsChecksum="true" />
[...]
<BinaryContainerInfo>
  <FileReference fileRefId="ID_999912" kind="MeasurmentInformationFile" />
</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"`.

The following code snippet shows how a diagnosis information file can be provided.

```
<File id="ID_999914" path="/" name="myFunction.cdd" role="other"
checksum="E7A58CD816076EE26DE1D6BF2F13630000675FB2" needsChecksum="true" />
[...]
<BinaryContainerInfo>
  <FileReference fileRefId="ID_999914" kind="DiagnosisInformationFile" />
</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"`.

The following code snippet shows how a compliance information file can be provided.

```
<File id="ID_999910" path="/doc/" name="MISRA.doc" role="other"
checksum="27D8D7BB69E1D7E98C7A278C5A48199CE7B65399" needsChecksum="true" />
[...]
<BinaryContainerInfo>
  <FileReference fileRefId="ID_999910" kind="ComplianceInformationFile" />
</BinaryContainerInfo>
```

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 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"`.

The following code snippet shows how a licenese file can be provided.

```
<File id="ID_999911" path="/license/" name="BSD.TXT" role="other"
checksum="A7549D084CFD2F9C6DEFA940B9BD5DA402B8341D" needsChecksum="true" />
[...]
<BinaryContainerInfo>
  <FileReference fileRefId="ID_999910" kind="LicenseFile" />
</BinaryContainerInfo>
```

Validation & Verification

For Verification and Validation, additional files can be provide using one or more `FileReference` elements with the `kind="ValidationAndVerificationFile"`.

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.

```
<File id="ID_999920" path="/v_n_v/" name="scenario1.mdf" role="other"
checksum="DB1A8489D88604A5C896BAB2B35631314B257036" needsChecksum="true" />
<File id="ID_999921" path="/v_n_v/" name="equivalenceclasses.xml" role="other"
checksum="F61E2D36002DD140653334E4871DEBE6EE3B721A" needsChecksum="true" />
[...]
<BinaryContainerInfo>
  <FileReference fileRefId="ID_999910" kind="ValidationAndVerificationFile" />
</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"`.

The following code snippet shows how a SOME/IP stack configuration for Adaptive AUTOSAR application is referenced.

```
<File id="ID_999910" path="/adaptive/" name="someip.json" role="other"
checksum="DB1A8489D88604A5C896BAB2B35631314B257036" needsChecksum="true" />
[...]
<BinaryContainerInfo>
  <FileReference fileRefId="ID_999910" kind="ConfigurationFile" />
</BinaryContainerInfo>
```

6.3. 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].

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].

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.

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.

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).

Chapter 9. Tool Support

This eFMI version was evaluated with prototypes of the following tools (alphabetical list):

Tool	Vendor	eFMI support
AUTOSAR Builder	Dassault Systèmes	Generation of Adaptive AUTOSAR from eFMI Production and eFMI Binary Code
Astrée	AbsInt Angewandte Informatik GmbH	Verification of eFMI Production Code
CSD	Siemens NV	Test of eFMI Production Code with eFMI Behavioral Model; integration in existing code and verification of code
Dymola	Dassault Systèmes	Generation of eFMI Algorithm Code and eFMI Behavioral Model (reference results) from Modelica model
ESP	Dassault Systèmes	Generation of eFMI Production Code from eFMI Algorithm Code; Generation of eFMI Binary Code from eFMI Production Code
Simcenter Amesim	Siemens Digital Industries Software	Generation of eFMI Algorithm Code from neural network approximation of Amesim model
SCODE CONGRA	ETAS GmbH	Generation of eFMI Production Code from eFMI Algorithm Code; test of eFMI Production Code with eFMI Behavioral Model
SimulationX	ESI ITI GmbH	Generation of eFMI Algorithm Code from Modelica model
TargetLink	dSPACE GmbH	Generation of eFMI Production Code from eFMI Algorithm Code; test of eFMI Production Code with eFMI Behavioral Model
TPT	PikeTec GmbH	Test of eFMI Production Code with eFMI Behavioral Model

Literature

[] 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

[] 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

[] **The C Programming Language**. https://en.wikipedia.org/wiki/C_(programming_language), last visited 2019-02-06.

[] **C++** Programming Language. https://en.wikipedia.org/wiki/C%2B%2B, last visited 2019-02-06.

[] **Dynamic-Link Libraries**. https://docs.microsoft.com/en-us/windows/desktop/Dlls/dynamic-link-libraries, last visited 2019-03-29.

[] **Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification**. http://refspecs.linuxfoundation.org/elf/elf.pdf, last visited 2019-03-28.

[] Kernighan Brian W., Ritchie Dennis M. (1978): **The C Programming Language** (1st ed.), Englewood Cliffs, NJ: Prentice Hall. ISBN 0-13-110163-3.

[] 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

[] MISRA C++:2008: **Guidelines for the use of the C++ language in critical systems**. ISBN 978-906400-04-0, MIRA Limited, Nuneaton, March 2013

[] 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

[] **Secure Hash Algorithm**. https://en.wikipedia.org/wiki/Secure_Hash_Algorithms, last visited 2019-02-08.

[] Stroustrup Bjarne (1997), **The C++ Programming Language** (Forth ed.), Addison-Wesley, ISBN 0-32-156384-0.

Appendix A: eFMI Revision History

Version	Date	Release Status	Notes
0.0.1	April 01, 2019	EMPHYSIS internal	Initial sketch.
0.6.0	Aug. 04, 2020	EMPHYSIS internal	Incomplete draft (for tool development).
1.0.0-alpha.1	Nov. 12, 2020	EMPHYSIS internal + shared with FMI group	Draft of specification.
1.0.0-alpha.2	Jan. 26, 2021	EMPHYSIS internal + shared with FMI group	Status before Equation Code Model representation was moved to appendix
1.0.0-alpha.3	Jan. 27, 2021	Publicly available	Equation Code Model representation moved to appendix. New section *Tool Support*. License of document changed to *Creative Commons Attribution-ShareAlike 4.0 International* and of accompanying code and data to *2-Clause BSD License*.
1.0.0-alpha.4	Feb. 22, 2021	Publicly available	Remaining Equation Code references removed. Images of schema files updated. License of accompanying code and data changed to *3-Clause BSD License*. Minor improvements of some descriptions.

Version 1.0.0

Contributors of Specification

The eFMI specification was developed within the ITEA EMPHYSIS project (https://itea3.org/project/emphysis.html) that was initiated and organized by Oliver Lenord, Christian Bertsch (Robert Bosch GmbH), Pacôme Magnin (Siemens) and Martin Otter (DLR-SR).

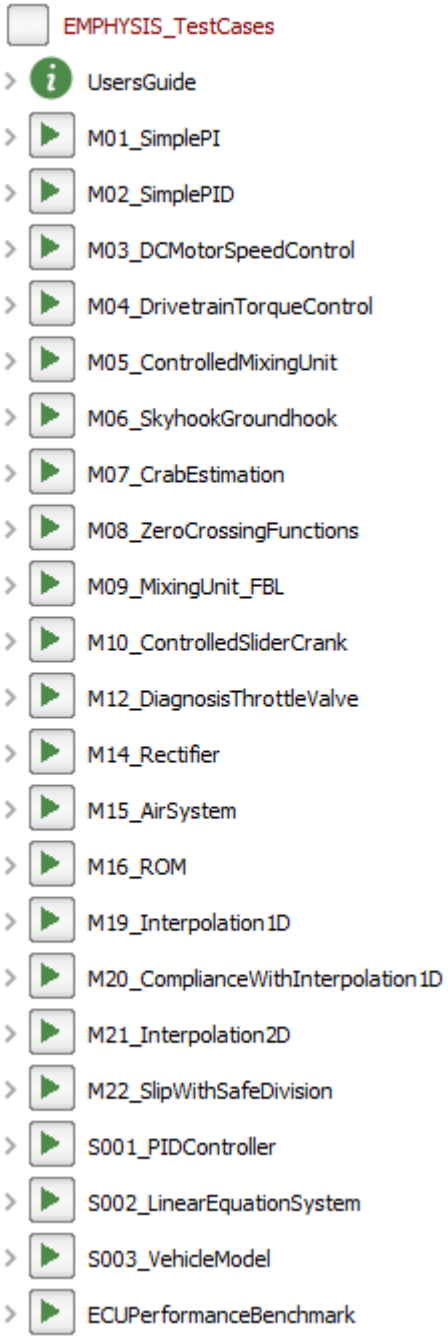
The development of the eFMI specification was headed and managed by Oliver Lenord (Robert Bosch GmbH). The essential part of the design of this version was performed by the following core development groups that closely worked together (alphabetical listings in the respective subgroups) and that utilized feedback and input from Benchmark Test Cases, Tool Assessment, as well as Demonstrators:

- **Behavorial Model**
  Yuri Durodié (Siemens NV)
  Andreas Pfeiffer (DLR-SR)
  Robert Reicherdt (PikeTec)
- **Rudimentary Equation Code**

Andreas Pfeiffer (DLR-SR)
Robert Reicherdt (PikeTec)

- **Algorithm Code**
  Christoff Bürger (Dassault Systèmes AB)
  Martin Otter (DLR-SR)
  Andreas Pfeiffer (DLR-SR)
- **Production Code**
  Jörg Niere (dSPACE GmbH)
  Michael Hussmann (dSPACE GmbH)
  Kai Werther (ETAS GmbH)
- **Binary Code**
  David Brenken (EFS)
  Pierre Le Bihan (Dassault Systèmes)
  Robert Reicherdt (PikeTec)

Benchmark Test Cases



The specification was assessed with benchmark tests cases provided in the Modelica library *EMPHYSIS_TestCases* and with Simcenter Amesim models. The *EMPHYSIS_TestCases* library was managed by Andreas Pfeiffer (DLR-SR) and Christoff Bürger (Dassault Systèmes AB).

The benchmark test cases have been developed by:

- **Robert Bosch GmbH**
  Siva Sankar Armugham
  Christian Bertsch
  Oliver Lenord
  Naresh Mandipalli
  Jonathan Neudorfer
  Christian Potthast
  Vishnupriya Veeraragavan
- **DLR-SR**

Jonathan Brembeck
Ricardo de Castro
Michael Fleps-Dezasse
Martin Otter
Andreas Pfeiffer
Jakub Tobolar

- **Siemens Digital Industries Software**
  Jérôme André

Tool Assessment

The eFMI specification was assessed by implementing eFMI support in various tools whose interoperability as a tool chain was evaluated. To that end, more than a hundred test models and variants of the benchmark test cases provided by the *EMPHYSIS_TestCases* library have been used to validate tool interoperability and correctness.

The developed and bechmarked tools are, in alphabetic order:

AUTOSAR Builder (Dassault Systèmes)

- Production and Binary Code → Adaptive AUTOSAR
- Developers: Fabien Aillerie

Astrée (AbsInt Angewandte Informatik GmbH)

- Verification of Production Code
- Developers: Reinhold Heckmann

CSD (Siemens NV)

- Test of Production Code with Behavioral Model, integration in existing code and verification of code
- Developers: Jishnu Jayaram

Dymola (Dassault Systèmes AB)

- Modelica → Algorithm Code
- Modelica → Behavioral Model
- Developers: Christoff Bürger

ESP (Dassault Systèmes)

- Algorithm Code → Production Code
- Production Code → Binary Code
- Developers: Samuel Devulder, Pierre Le Bihan, Laurent Le Goff

SCODE CONGRA (ETAS GmbH)

- Algorithm Code → Production Code
- Test of Production Code with BehavioralModel

- Developers: Kai Werther

Behavioral Model Scripts (DLR-SR)

- Generation of Behavioral Model
- Developers: Andreas Pfeiffer

Simcenter Amesim (Siemens Digital Industries Software)

- Amesim model → neural network approximation as Algorithm Code
- Developers: Jérôme André

SimulationX (ESI ITI GmbH)

- Modelica → Algorithm Code
- Developers: Gerd Kurzbach

TargetLink (dSPACE GmbH)

- Algorithm Code → Production Code
- Test of Production Code with Behavioral Model
- Developers: Michael Hussmann, Jörg Niere

TPT (PikeTec)

- Test of Production Code with Behavioral Model
- Developers: Robert Reicherdt

Demonstrators

The eFMI specification and the developed tools have been assessed by industrial demonstrators:

Performance assessment (Robert Bosch GmbH)

Comparing generated Production Code of nine benchmark test cases of the *EMPHYSIS_TestCases* library with manually developed code. This includes comparison of execution performance on the Bosch ECU MDG1.

- Tooling: Performance Test Environment
- Developer: Vishnupriya Veeraragavan

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 (www.openmodelica.org), SCODE-CONGRA, ECU Test Environment

- Contributors: Oliver Lenord, Christian Potthast

Virtual sensor for hybrid drivetrain (Siemens)

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 (DLR-SR)

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 EFS 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 (EFS)
  - Jonathan Brembeck (DLR-SR)
  - Daniel Baumgartner (DLR-SR)
  - Christoff Bürger (Dassault Systèmes AB)
  - David Brenken (EFS)
  - Dario Celan (EFS)
  - Georg Hofstetter (EFS)
  - Michael Hussmann (dSPACE GmbH)
  - Konrad Krauter (EFS)
  - Severin Kirpal (EFS)
  - Jorg Niere (dSPACE GmbH)
  - Andreas Pfeiffer (DLR-SR)
  - Raik Ritter (EFS)
  - Julian Ruggaber (DLR-SR)
  - Christina Schreppel (DLR-SR)
  - Jakub Tobolar (DLR-SR)
  - Johannes Ultsch (DLR-SR)
  - Christoph Winter (DLR-SR)

Dual-clutch use case (Daimler 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
  - Hardware-in-the-loop tests: TargetLink, ConfigurationDesk (dSPACE GmbH) and PROVEtech (Akka Technologies)
- Contributors:
  - Zdenek Husar (Daimler AG)
  - Jan Röper (Daimler AG)
  - Emmanuel Chrisofakis (Daimler AG)
  - Klaus Riedl (Daimler AG)
  - Christoff Bürger (Dassault Systèmes AB)
  - Hans Olsson (Dassault Systèmes AB)

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 AB)

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: ESP
    - AUTOSAR Adaptive Platform component generation: AUTOSAR Builder
- Contributors:
    - Christoff Bürger (Dassault Systèmes AB)
    - 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 B: 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).
`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).

Function-Name	Description
`remainderHalfTowardsZero(i1,i2)`	Integer remainder of roundHalfTowardsZero(i1/i2).
`remainderHalfAwayZero(i1,i2)`	Integer remainder of roundHalfAwayZero(i1/i2).
`remainderHalfToEven(i1,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(r1,r2)`	Real remainder of roundHalfUp(r1/r2).
`realRemainderHalfTowardsZero(r1,r2)`	Real remainder of roundHalfTowardsZero(r1/r2)
`realRemainderHalfAwayZero(r1,r2)`	Real remainder of roundHalfAwayZero(r1/r2)
`realRemainderHalfToEven(r1,r2)`	Real remainder of roundHalfToEven(r1/r2)
`realRemainderHalfToOdd(r1,r2)`	Real remainder of roundHalfToOdd(r1/r2)

Definition of the reserved built-in functions

The following functions are appended to $C_{builtin1}$:

```
/*****************************************************************************************
********
    Direct rounding to an integer:
*****************************************************************************************
*******/

function roundTowardsZero
    input  Real r;
    output Real i;
algorithm /*
    Also known as: truncation, round away from infinity.
    i := (if r >= 0.0 then roundDown(r) else roundUp(r));
*/ end roundTowardsZero;

function roundAwayZero
    input  Real r;
```

```
    output Real i;
algorithm /*
    Also known as: round towards infinity.
    i := (if r <= 0.0 then roundDown(r) else roundUp(r));
*/ end roundAwayZero;

/*******************************************************************************
********
    Rounding to the nearest integer (using a tie-breaking rule):
*******************************************************************************
*******/

function roundHalfDown
    input  Real r;
    output Real i;
algorithm /*
    Also known as: round half towards negative infinity.
    i := roundUp(r - 0.5);
*/ end roundHalfDown;

function roundHalfUp
    input  Real r;
    output Real i;
algorithm /*
    Also known as: round half towards positive infinity.
    i := roundDown(r + 0.5);
*/ end roundHalfUp;

function roundHalfTowardsZero
    input  Real r;
    output Real i;
algorithm /*
    Also known as: round half away from infinity.
    i := roundAwayZero(r - sign(r) * 0.5);
*/ end roundHalfTowardsZero;

function roundHalfAwayZero
    input  Real r;
    output Real i;
algorithm /*
    Also known as: round half towards infinity.
    i := roundTowardsZero(r + sign(r) * 0.5);
*/ end roundHalfAwayZero;

function roundHalfToOdd
    input  Real r;
    output Real i;
algorithm /*
    i := (if roundHalfDown(r) < roundHalfUp(r)
          then (if integer(remainder(r + 0.5, 2.0)) == 0 then r - 0.5 else r + 0.5)
          else roundHalfDown(r));
```

```
*/ end roundHalfToOdd;

/************************************ END OF LISTING
************************************/
```

The following functions redefine $C_{builtin2}$, which defines builtin functions for *Integer* division. For every function named `roundα` of $C_{builtin1}$ with `α` an arbitrary sequence of characters, $C_{builtin2}$ contains the character sequence:

```
/************************************ BEGIN OF LISTING
************************************/

function divisionα
    input Integer dividend;
    input Integer divisor;
    output Integer quotient;
algorithm /*
    quotient := integer(roundα(real(dividend) / real(divisor)));
*/ end divisionα;

function remainderα
    input Integer dividend;
    input Integer divisor;
    output Integer remainder;
algorithm /*
    remainder := dividend - divisor * divisionα(dividend, divisor);
*/ end remainderα;

/************************************ END OF LISTING
************************************/
```

Further, $C_{builtin2}$ contains the following character sequence:

```
/************************************ BEGIN OF LISTING
************************************/

function divisionEuclidean
    input Integer dividend;
    input Integer divisor;
    output Integer quotient;
algorithm /*
    quotient := integer((if divisor > 0
        then roundDown(real(dividend) / real(divisor))
        else roundUp(real(dividend) / real(divisor))));
*/ end divisionEuclidean;

function remainderEuclidean
    input Integer dividend;
    input Integer divisor;
    output Integer remainder;
algorithm /*
    remainder := dividend - divisor * divisionEuclidean(dividend, divisor);
*/ end remainderEuclidean;

/************************************ 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:

```
/************************************ BEGIN OF LISTING
************************************/

function realRemainderα
    input Real dividend;
    input Real divisor;
    output Real remainder;
algorithm /*
    remainder := dividend - divisor * round_α(dividend / divisor);
*/ end realRemainderα;

/************************************ END OF LISTING
************************************/
```

Above functions are in lexical order w.r.t. their names; they constitute $C_{builtin3}$ in its entirety.

Appendix C: Equation Code Model Representation

This section describes rudimentary support for the planned Equation Code model. It is not part of the eFMI standard, because the development is not yet finalized. This appendix summarizes the status of the development. An improved version might be added to a future version of the eFMI standard.

Introduction

The Equation Code model shall describe the mathematical model of the acausal, *continuous-time* physical system with a standardized, intermediate language (a subset of the Modelica language (https://www.modelica.org/modelicalanguage), often also referred to as *Flat Modelica*).

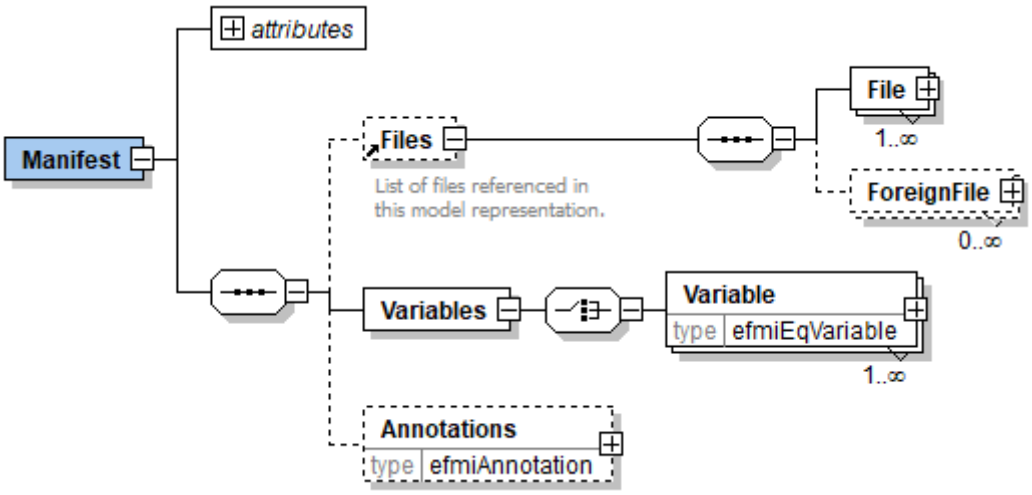
Conceptually, the Equation Code model representation depicts the earliest stage of the model analyses. Here any language specific analyses, e.g. such as syntax checks are already done. However, the model is still acausal, i.e. the inputs and outputs are not yet fixed, the states not yet selected and the equations are not yet sorted and discretized.

This representation form is currently under developement and is not yet defined in this specification, with exception of a very rudimentary manifest file that is needed to connect Behavioral Model and Algorithm Code representations.

Manifest schema

The rudimentary *manifest file* of the Equation Code model representation is an instance of an XML schema definition and defines the names and types of the variables that are used in the interface of the model.

Definition of an eFMU Equation Code (efmiEquationCodeManifest.xsd)

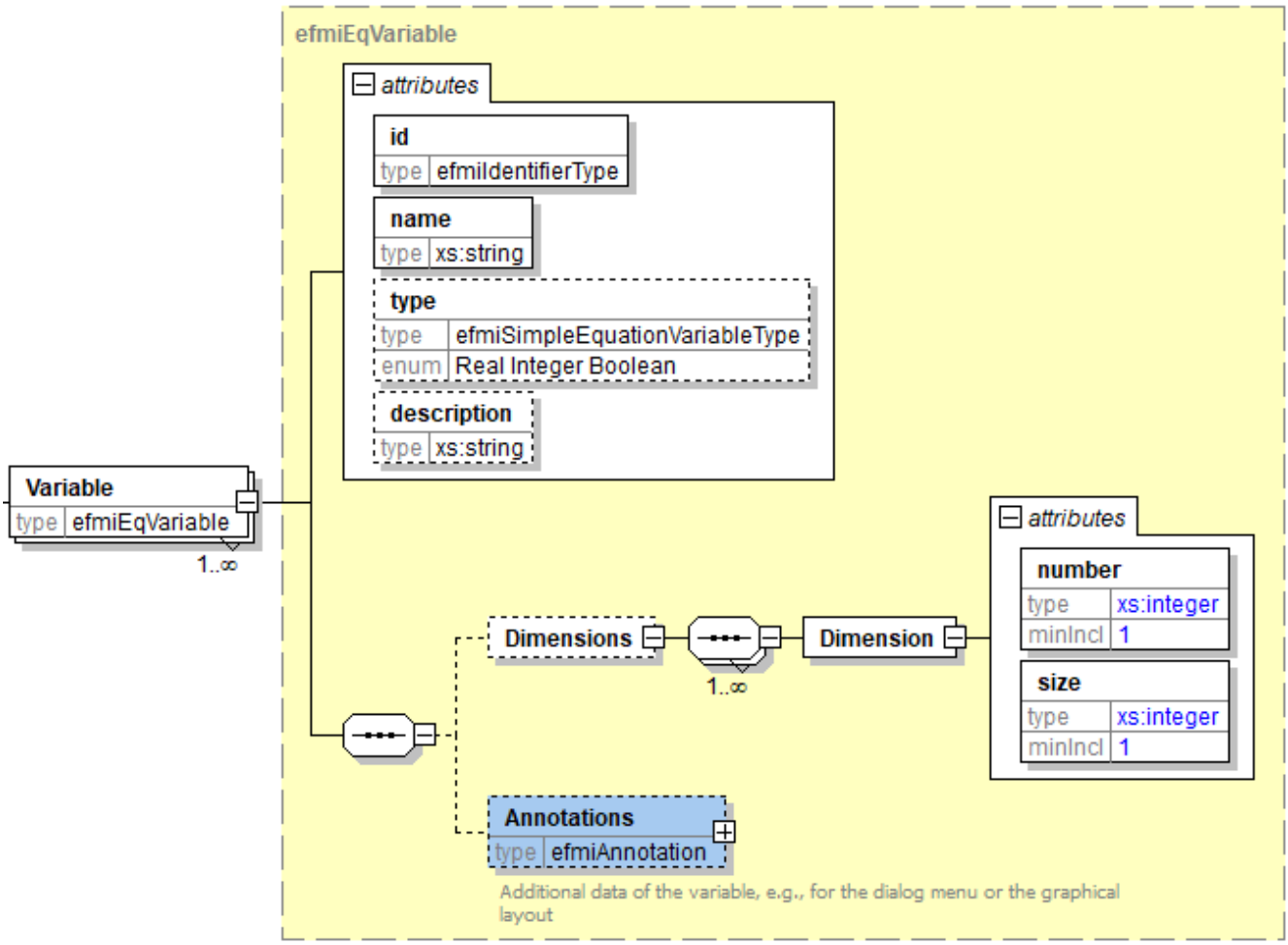


On the top level, the schema consists of the following elements:

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 = "EquationCode"`, `xsdVersion` (value is the current xsd version of the schema for the Equation Code model manifest).
`Files`	List of files referenced in this model representation. Currently, no Files are defined. This element is the same for all manifest kinds and is defined in section Section 2.3.3.
`Variables`	A list of the discrete-time interface variables of the model. A variable might be a scalar or an array of an elementary type. For details see Definition of an Equation Code Variable (efmiEqVariable.xsd).
`Annotations`	Additional data that a vendor might want to store and that other vendors might ignore. For details see Section 2.3.4.5.

Definition of an Equation Code Variable (efmiEqVariable.xsd)

An Equation Code defines a set of Variables. A Variable is defined in the following way:



The schema consists of the following elements:

Name	*Description*
`id`	The *unique* identification of the variable with respect to the EquationCode 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 EquationCode instance by this name.
`type`	The base type of the variable. Valid values are: `Real`, `Integer`, Boolean`.
`description`	An optional description string describing the meaning of the variable.
`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.

```
<script>
// hide / show the Table of Contents (TOC)
function toggleTOC() {
  var toc = document.getElementById("toc");
  var body = document.getElementsByTagName("body")[0];

  if (toc.style.display === "none") {
      toc.style.display = "block";
      body.classList.remove("toc-hidden");
  } else {
      toc.style.display = "none";
      body.classList.add("toc-hidden");
  }
}

// toggle the TOC when "t" key is pressed
document.addEventListener('keydown', (event) => {
  if (event.key == 't') {
    toggleTOC();
  }
});
</script>
```