eFMI: An open standard for physical models in embedded software

Oliver Lenord[1] Martin Otter[2] Christoff Bürger[3] Michael Hussmann[4]
Pierre Le Bihan[5] Jörg Niere[4] Andreas Pfeiffer[2] Robert Reicherdt[6] Kai Werther[7]

[1]Robert Bosch GmbH, Germany, [2]DLR-SR, Germany,
[3]Dassault Systèmes AB, Sweden, [4]dSPACE GmbH, Germany, [5]Dassault Systèmes SE, France,
[6]PikeTec GmbH, Germany, [7]ETAS GmbH, Germany

Abstract

This paper summarizes the final research results of the ITEA3 project EMPHYSIS (***em***bedded systems with ***phys***ical models ***i***n the production code ***s***oftware). Its core achievement is the new open *eFMI Standard* enabling automated workflows from high-level mathematical models of physical systems (referred to as physical models) to automotive compliant embedded software. eFMI (FMI for embedded systems) defines a container architecture for model exchange and testing. Multiple representations from an intermediate representation of sampled algorithms (GALEC) to production and binary code for specific embedded targets are maintained in a traceable workspace. The successful integration of the developed eFMI tooling is demonstrated by a comprehensive open source Modelica test cases library and industrial demonstrators. The readiness of the proposed approach is proven by compliance checks according to common automotive code quality standards like MISRA C:2012 and a performance benchmark in terms of runtime and resource demand in comparison with state-of-the-art hand coded solutions.

Keywords: embedded software, model-based development, code generation, model exchange, Modelica, FMI, eFMI, GALEC

1 Introduction

1.1 Motivation

Software has become an innovation driver not only but especially in the automotive industry. This has been leading to new challenges in terms of maintainability of the growing software stack for a growing number of ECUs (Electronic Control Units) in vehicles.

In the field of application software, it is the growing variability of the vehicles, increasingly demanding regulations and new powertrain solutions that add to the complexity of control and diagnosis functions. Original equipment manufacturers (OEM) as well as Tier 1 suppliers are aiming to cope with these demands by using mathematical models of physical systems (referred to as physical models) as part of the control software. For example, in Zimmermann et al. (2015) physical models are considered essential to manage the growing complexity of Diesel engine control as map-based approaches lead to an overwhelming calibration effort to satisfy the requirements of real driving emissions (RDE). Englert et al. (2019) present a framework for embedded non-linear model predictive controllers (NMPC) as a very generic approach of integrating physical models into a controller. With the increased computational power of modern multi-core ECUs, like the Bosch MDG1 (Rüger et al. 2014), these advanced approaches become relevant for industrial applications. In addition, e.g. Bosch is holding patents on methods for real-time applications of physical models (e.g. Wagner et al. 2009) stressing the fact that managing this type of applications is a differentiating selling proposition.

Model-based development (MBD) is an established paradigm for the development of control software for embedded targets deemed to ease these challenges. In practice the commonly used signal flow-oriented models, restricted to a predefined set of blocks, do not scale to the need. The models are rather a model of the software, than a model of the physical system with poor means to reflect variants of the underlying physical structure. The rather low-level description requires a high level of expertise of the modeler far beyond the physical behavior of the system including floating-point arithmetic, embedded software regulations, e.g. MISRA C:2012 rules (MISRA 2013-03) and target architectures, e.g. the AUTomotive Open System Architecture (AUTOSAR) (AUTOSAR Consortium 2021).

Despite valuable pioneering work on FMI (Functional Mock-up Interface) (Modelica Association 2021-04) in AUTOSAR, discussed in the following section, there is today no simulation solution supporting the export of physical models suitable for direct integration into controls, and no standard of any kind supporting efficient generation and integration processes from physical models to embedded software.

The goal of FMI for embedded systems (eFMI) is to overcome the known limitations of FMI, to enable new ways of model-based development of embedded software functions for arbitrary targets and architectures, based on advanced physical models of the underlying system.

1.2 State of the art of FMI in AUTOSAR

In a case study by Bertsch et al. (2015) it has been demonstrated that it is possible to wrap the C code from a Source Code FMU (Functional Mock-up Unit), that has been generated from a Modelica (Modelica Association 2021-02) physical model, as AUTOSAR ASW-C (Application Software Component) and to execute it on a Bosch ECU. This study also revealed the conceptual weaknesses of FMI when it comes to embedded automotive software and safety critical applications.

The probably most obvious weakness roots in the purposeful design of FMI being a standardized <u>interface</u> without being directive about the implementation of interface functions, except of which entry level files to provide and which headers to include. In contrast to that, the widely accepted rules of the Motor Industry Software Reliability Association (MISRA) are very specific about <u>how</u> a function is to be implemented in the C language to avoid ambiguities and runtime exceptions, while ensuring readability of the code. The goals of maximum flexibility for the exporting tool to produce any kind of *simulation code* to be wrapped into a black box running on a personal computer (PC) vs. high quality embedded *production code* being subject to thorough code reviews before being deployed to a dedicated target according to a certified procedure are contradicting. FMI is not designed to provide production code and binaries that are ready to satisfy the requirements of automotive embedded software quality gates. There are no mechanisms for the traceability of the code and no attributes about the used compilers and compiler settings to ensure the repeatability of the build process of a binary targeting a dedicated application in a specific runtime environment with the type of microcontroller already defined.

The prototypical tooling developed by Bertsch et al. (2015) enhanced by Neudorfer et al. (2017) is focusing on the technical aspect of translating a Source Code FMU into an AUTOAR SW-C. The aspects of which meta data has to be provided to support an end-to-end tool chain from the original model to the deployed binary is not discussed and remains as an unresolved issue of the prototypical work not intended for productive usage.

In terms of code quality Bertsch et al. (2015) state that none of the inspected source codes of the evaluated tools fulfilled their requirements and that the "C-code from many commercial tools is not suitable to run on an ECU due to its size and complexity since it was not intended to run on an embedded system".

The tooling presented by Bertsch et al. (2015) translates in a first step the FMI `modelDescription.xml` file into a corresponding AUTOSAR `.arxml` file, based on a number of design decisions made on the desired representation as SW-C. The second step involves the translation of the C code of the Source Code FMU into C code that can be processed by build tools from Bosch for engine control software. After inspection of the source code generated by three different Modelica tools certain patterns were identified, such as: moving declarations to public or private headers, exclusion of functions from, e.g., `stdio.h` and `math.h` (ISO/IEC 2018-06) which are not supported on embedded devices, taking care of proper assignment of float values and avoiding implicit type casts. This process had been automated to a large extent, but made assumptions on the structure of the code and was finally still relying on the expertise of an embedded software developer to inspect the translated code and to fix remaining issues before further compilation.

This involved procedure illustrates that, if the generator of the C code is not giving any guarantees on its code, then it is very difficult, if not impossible, for the consuming tool to enforce these afterwards. From a workflow perspective, it is also highly objectionable when only after importing and processing of an FMU deficits are revealed that have to be addressed by the exporting tool.

All these issues are only about just compiling the code; but to fulfill the requirements of an embedded software the following aspects regarding the behavior and resource demand of the code have to be addressed as well:

- Limited data memory and code memory.
- Limited computation power of the target.
- Limitations on supported data types: 32-bit vs. 64-bit floating-point precision, fixed-point arithmetic etc.
- Static memory allocation only.
- Guaranteed exception freeness: Programs never fail due, e.g., unavailable/busy external devices, writing read-only memory, accessing multi-dimensions out-of-bounds, running out of stack memory etc.
- Guaranteed execution time within the limits of the available target system resources and the minimal sampling period required for correct physical behavior.
- Proper error handling: Handling of Not a Number (NaN) (IEEE 2019-07), mathematical functions that are undefined for some arguments, linear systems without unique solution etc.
- In bound guarantees of signals.

Simulation code generated by today's FMU generating tools is optimized for runtime performance on a PC. The memory required by the FMU is allocated dynamically. The sizes of the data and program code are not considered limiting factors and therefore not minimized. This renders existing FMI solutions unsuited for application in the embedded domain.

For co-simulation FMUs (CS-FMU) there are no restrictions for the type of solver being used. For real-time applications, variable-step-size solvers are not suitable, as they cannot guarantee the execution time on an equidistant time grid. Event handling and non-linear algebraic loops are particularly challenging, since self-evident unbounded iterative solutions of such cannot be used; instead, such have to be expressed as upper-bounded iterative algorithms with execution time guarantees. If this is not possible, the system is not suited for embedded real-time.

Hence, the consumer of an FMU fully depends on the modeler and exporting tool to have made appropriate choices in setting up the model and making the settings for the solver and the compiler to meet the basic runtime requirements. An FMU by itself gives no guarantees further processing can rely upon.

Furthermore, FMUs are expected to run in a simulation or co-simulation environment that provides exception handling mechanisms. Messages may be dumped into a log file in case of exceeded value ranges based on the assert statements in the code for the simulation engineers to verify that they can trust the results. In safety critical applications, there is no second attempt. The software must guarantee exception free execution and a predictable behavior under all circumstances. FMI does not provide any means to cope with this requirement.

Despite the fact that one can find ways to translate the C code from a Source Code FMU to compile and being executed on an embedded target, one has to state that FMI has no means to guarantee that the source code fulfills basic prerequisites for embedded real-time execution, nor does it provide a rich enough model description to support an end-to-end build chain in terms of traceability, transparency and repeatability of the process.

1.3 eFMI vs. FMI

eFMI helps to overcome the shortcomings of FMI, explained in Section 1.2, for the development of embedded software based on physical models. eFMI is not just an extension of FMI; it is an orthogonal, new standard that is maintained and further developed in a separate Modelica Association Project. Entirely new concepts are introduced, but at the same time, a high degree of consistency with FMI has been achieved. Whereas FMUs are ready-made "consumer" products for exchanging simulation models, an *FMU for embedded systems* (eFMU) is a shared development workspace for step-wise, semi- and full-automatized refinement from a high-level intermediate representation of a sampled algorithm (in the GALEC language, see Section 2.4) to an implementation of the algorithm for an embedded target. The development of a single eFMU is shared between varying eFMI tools and developers. Throughout its development, the eFMU very likely is in intermediate stages not suited for simulation. The developed final solution can be wrapped by a respective FMI interface such that it behaves like a regular FMU. Doing so, the outer FMI shell allows any FMI supporting tool to load and execute the eFMU as CS-FMU, accessing the actual production code through appropriate wrappers. This enables verification and validation (V&V) of the target code in a Software-in-the-Loop (SiL) environment without restricting the target code to satisfy a static C interface.

1.4 eFMI Workflow

Key differentiator of the proposed eFMI workflow (see Figure 1) is a multistep approach reflected by different types of so-called *model representations* that are stored in containers within the same eFMU. Each model representation addresses a different aspect and refinement step of a physical model to embedded software in the development process. The intention is to provide an automatable workflow, where the model representations can be generated, with tool-supported refinement along the "*Transform*" boxes in Figure 1.

The *Algorithm Code* model representation (see Section 2.4) provides a target independent, intermediate representation for upper-bounded algorithmic

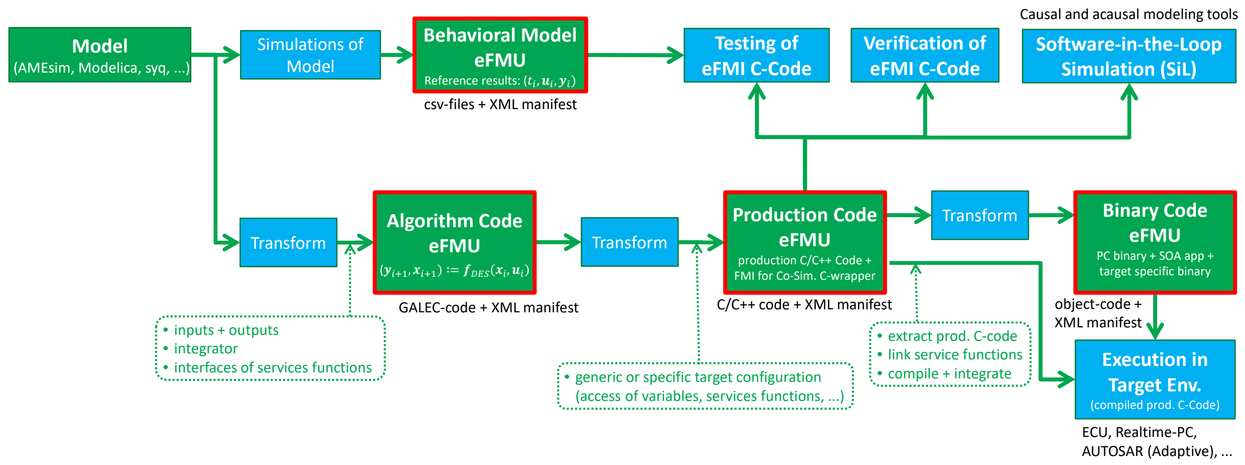
Model
(AMEsim, Modelica, syq, ...)
Simulations of Model
Behavioral Model eFMU
Reference results: (t_i, u_i, y_i)
csv-files + XML manifest
Causal and acausal modeling tools
Testing of eFMI C-Code
Verification of eFMI C-Code
Software-in-the-Loop Simulation (SiL)
Transform
Algorithm Code eFMU
$(y_{i+1}, x_{i+1}) := f_{DES}(x_i, u_i)$
GALEC-code + XML manifest
Transform
Production Code eFMU
production C/C++ Code + FMI for Co-Sim. C-wrapper
C/C++ code + XML manifest
Transform
Binary Code eFMU
PC binary + SOA app + target specific binary
object-code + XML manifest
Execution in Target Env.
(compiled prod. C-Code)
ECU, Realtime-PC, AUTOSAR (Adaptive), ...
• inputs + outputs
• integrator
• interfaces of services functions
• generic or specific target configuration (access of variables, services functions, ...)
• extract prod. C-code
• link service functions
• compile + integrate

Figure 1. eFMI workflow with its different model representations (red boxes).

computations. It serves as code generation target for (physics-based) modeling tools, allowing them to concentrate on the general concern of finding a sequence of computations that satisfies real-time constraints, i.e., is algorithmically well-defined with an upper-bound of computational steps.

Such algorithmic solutions are further refined to actual implementation code that is best suited for a specific target environment in terms of performance, memory consumption, software architecture and applicable rules and regulations by *Production Code* model representations (see Section 2.5). For a single Algorithm Code container, several Production Code containers can be given to address varying target platforms, e.g., alternative chip sets or customer specific code guidelines.

For each production code, arbitrarily many compiled binaries, tailored for embedded system integration within a specific ECU/target platform, can be included via *Binary Code* model representations (see Section 2.6). Besides defining a target specific build and integration, Binary Code containers can also protect intellectual property by only providing the binaries as such without sources.

Finally, validation and verification (V&V) is covered by means of *Behavioral Model* model representations (see Section 2.3), which allow to store reference results for later back-to-back testing of other model representations like Production and Binary Code containers.

Starting from an Algorithm Code container, there is no further stringent order in which traceable containers must be provided or updated throughout eFMU development iterations. This enables full flexibility in the software development process supporting all kinds of model and software sharing schemes for OEMs and suppliers, with eFMI as open standard giving maximum freedom in the choice of tools. An eFMU is a standardized workspace for collaborative development of embedded solutions from (physical) models.

1.5 Structure of the paper

This paper gives a coarse introduction to the basic concepts of the eFMI Standard (Section 2) and highlights the achieved goals in terms of readiness and applicability to industrial grade problems (Sections 3, 4), before summarizing the future work and the conclusions (Sections 5, 6).

2 eFMI Standard

The following description of the eFMI Standard is according to version 1.0.0-alpha.4 (EMPHYSIS 2021-07) published in February 2021; on the same web page an example eFMU can be downloaded.

2.1 Mathematical description of eFMI

As described in Section 2.6, the starting point of eFMI workflows are typically physical models for some independent modeling and simulation environment, e.g., a Modelica tooling (Modelica Association 2021-02). Such original models can be described by (unsorted) equations, algorithms or functions, which have to be transformed to eFMI Algorithm Code model representations. This requires a causal, discretized algorithmic solution to be found for the acausal physics equations – hence the modeling environment must provide a GALEC code generation backend (see Section 2.6).

Mathematically, an algorithmic solution can be described as a *sampled input/output block* with *one* (potentially varying) sample period $T_i = t_{i+1} - t_i$ for the whole block. Inputs $\boldsymbol{u}_i = \boldsymbol{u}(t_i)$ and previous block internal states $\boldsymbol{x}_i$ are provided at sample time t_i whereas outputs $\boldsymbol{y}_i = \boldsymbol{y}(t_i)$ and new states $\boldsymbol{x}_{i+1}$ are computed in the block (see Figure 2).

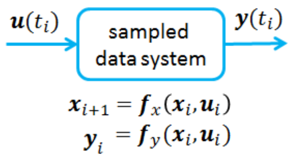
$\boldsymbol{u}(t_i)$
sampled data system
$\boldsymbol{y}(t_i)$
$\boldsymbol{x}_{i+1} = \boldsymbol{f}_x(\boldsymbol{x}_i, \boldsymbol{u}_i)$
$\boldsymbol{y}_i = \boldsymbol{f}_y(\boldsymbol{x}_i, \boldsymbol{u}_i)$

Figure 2. Mathematical description of an algorithmic solution as supported by eFMI Algorithm Code model representations and the GALEC language.

All variables of the block have a defined type and all statements of the block are sorted and explicitly solved for a particular variable. Functions are provided to execute the relevant parts of the block, especially to initialize it (`Startup()` function) and to perform one step (`DoStep()` function).

To find an upper-bounded algorithmic solution for acausal equation systems is a non-trivial task, with reasonable solution strategies highly depending on characteristics of the original modeling language. The eFMI Standard therefore is not covering, or in any way restricting on how to find suited solutions; it solely concentrates on defining how such solutions must look like (see Section 2.4) to be processable by further eFMI tooling like production code generators.

2.2 eFMU Container Architecture

The basic file structure of eFMUs is:

```
/eFMU ; eFMU root directory
   /<directories> ; Model representations
   /schemas ; eFMI XML Schema definitions
   __content.xml ; eFMU content-manifest
```

The only mandatory file is the eFMU content-manifest (`__content.xml`) at the root of the `eFMU` folder. In addition, an eFMU includes the XML Schema files defined by the eFMI Standard (`/schemas`) to become self-contained; these XML Schemas restrict the structure and content of the manifests of the varying eFMI model representations and thereby enable automatic processing of eFMUs and their content. All other directory and file names within an eFMU can be freely chosen by the tools generating and processing the varying eFMI model representations. The directories containing some model representation, like an Algorithm Code or Production Code container, are denoted in the eFMU content-manifest, which also lists the type of model representation and other meta information like checksums. Each model representation further supplies its own manifest according to its type of representation; manifests of model representations typically list all files of the representation, their checksums and other representation specific meta information, like the in- and outputs of GALEC programs (Algorithm Code containers) or the compiler settings used to produce some binary code (Binary Code containers).

The structure of a typical eFMU could look like this:

```
/eFMU
   /BehavioralModel
      manifest.xml ; Container manifest
      <other files>
   /AlgorithmCode
      manifest.xml ; Container manifest
      <other files>
   /ProductionCode_Generic_C_Float32
      manifest.xml ; Container manifest
      <other files>
   /ProductionCode_Generic_C_Float64
      manifest.xml ; Container manifest
      <other files>
   /ProductionCode_Autosar_Float32
      manifest.xml ; Container manifest
      <other files>
   /schemas
   __content.xml ; eFMU content-manifest
```

An eFMU can be packed in different formats.

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 git or SVN.
2. The eFMU root directory is zipped with the eFMU-content and is stored in a zip-file with the extension `.efmu`. This is useful to ship or distribute an eFMU.
3. The eFMU root directory is path `extra/org.efmi-standard` inside a standard FMU. The path is defined according to the current FMI-3.0-beta.1 pre-release of the FMI specification (Modelica Association 2021-04). With attribute `activeFMU` inside the eFMU content-manifest 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 Model/Software/Hardware-in-the-Loop (MiL/SiL/HiL) simulation by further FMU tooling.

Note, that Algorithm Code, Production Code and Binary Code representations can optionally store associated FMUs. For example, Algorithm Code representations can store a Model-in-the-Loop FMU and Production Code representations for different targets can store Software-in-the-Loop FMUs. To execute these FMUs, they must be extracted from the respective model representation – manually or by a tool. If an eFMU is organized according to package format (3), a selected FMU from a model representation has to be copied to the root level so that the eFMU behaves as an ordinary FMU and can be simulated by any FMI tool.

2.3 Behavioral Model Representation

The Behavioral Model representation of eFMI is designed to describe functional and behavioral aspects of the original (physical) model/system/controller with the goal of enabling the validation of other generated model representations within the same eFMU. The central question is how to define and include reference behavior for varying model representations and their respective software? As an example, one can think about a controller model in Modelica represented by ordinary or differential algebraic equation systems that shall be exported as eFMU. Reference result data may be important for several use case scenarios, e.g. simulation runs of the controller model in a Modelica tool:

- with a complex variable step-size integration algorithm to define a highly accurate reference solution,
- with a fixed step-size algorithm like the Explicit Euler method and a fixed step-size to get a reference solution of the discretized sampled data system,
- in different model setups like open and/or closed loop scenarios to cover a broad range of possible input value combinations,
- in other test scenarios to test specific features/requirements of the controller – like unit tests in software development – and
- all the tests may be run using the floating-point precision typically used in offline simulations of 64-bit or/and the more common precision for embedded systems of 32-bit.

To structure this variety of possible reference data a rather simple format with only two hierarchies (scenarios and scenario parts) has been designed: An eFMI Behavioral Model representation consists of scenario parts grouped in one or more use case scenarios. Each scenario part represents a single behavioral aspect of the original model given by different time dependent input/output data in a comma-separated values (CSV) file, e.g. a closed loop scenario with an Explicit Euler discretization of the controller model. Absolute and relative error tolerances or time dependent lower and upper bounds can be defined for each variable in the Behavioral Model manifest, to account for deviations resulting from discretization methods, implementation data types and floating-point imprecision.

Since variable names as well as data types might differ between the different eFMI representations within the same eFMU (e.g., different production code variants might map GALEC variables to different C identifiers to satisfy naming conventions of varying target environments), every variable of a Behavioral Model container is linked with its corresponding variable of the Algorithm Code container such that reference result data can be provided easily and fully automatic for each model representation. This means for example, that later added Production Code containers can be automatically tested using existing Behavioral Model containers thanks to the trace-links between variables of the manifests of Behavioral Model and Production Code containers to the manifest of the Algorithm Code container.

A typical validation approach consists of the execution of compiled eFMI production or binary code with input data from the CSV files. These simulation results are compared with the expected output data of the Behavioral Model representation according to the given error tolerances or bounds to validate the contained eFMI representations against the "behavior" of the source of the generated eFMU. This approach enables tools (such as testing tools) to perform a fully automatic validation of other eFMI representations (Algorithm Code, Production Code and Binary Code representations) within an eFMU w.r.t. behavioral and functional equivalence to the physical model using a back-to-back testing approach.

2.4 Algorithm Code Model Representation

All containers in an eFMU are related to the Algorithm Code container of which exists exactly one in each eFMU. It provides a high-level intermediate representation of a sampled algorithm by means of a GALEC block and a description of the block's interface by means of an XML manifest. The manifest is used by other containers to trace their dependencies on the block and avoid processing GALEC programs if just in need of a description of the block interface. The GALEC block can define any kind of finite sampled computation that is subject to embedded integration (controller, virtual sensor etc.). In terms of the Modelica language, a GALEC block is a causal solution for the sampled system defined by a clocked partition.

GALEC is a new imperative programming language part of the eFMI Standard. Its name is an abbreviation for ***g***uarded ***a***lgorithmic ***l***anguage for ***e***mbedded ***c***ontrol. It is an intermediate representation between the (physics) modeling and embedded programming domains.

GALEC Characteristics: The language characteristics C1-11 of GALEC, making it a suitable intermediate representation between modeling and embedded software, are:

(C1) Target independence: Arithmetic and algorithms are on an ideal machine, with built-in functions for abstract handling of target dependent operations like retrieving the fraction part of a real or checking if such is Not a Number (NaN) (IEEE 2019-07).

(C2) Explicit language semantics: No implicit casts between integer and real, no hidden side effects and no default arguments; simple name space without shadowing, overloading or polymorphic functions. These restrictions are kind of language-enforced MISRA C:2012 rules.

(C3) Multi-dimensional arithmetic: Support for vectors, matrices and higher dimensions with respective scalar and matrix multiplication, addition etc. Production code generators can leverage on (C4) to map multi-dimensional operations to efficient implementations, for example Streaming SIMD Extensions 4 (SSE4) machine instructions (Intel Corporation 2021-06).

(C4) Powerful static evaluation: GALEC expressions are separated into three kinds: (1) declarative sizes, (2) algorithmic indexing and (3) algorithmic runtime computations, with the former two being subject to static evaluation for mandatory well-formedness analyses. Algorithmic indexing hereby includes `for`-loop iterators and static evaluation of their ranges. Indexing expressions can depend on loop iterators but must not refer to any other variables for their value. Otherwise, statically evaluated expressions can be arbitrary complex, including calls of built-in functions.

(C5) Upper-bounded: GALEC programs must be non-recursive; the only iteration construct are `for`-loops, which according to (C4) can be unrolled. Thus, every program can be unfolded to an iteration-free sequence of conditional assignments defining an upper bound of algorithmic steps. This characteristic enables worst time execution analyses and advanced optimizations.

(C6) Computational-safe: The static unfolding-characteristics of (C4) and (C5) are used to guarantee all indexing is within bounds. Production code generators can avoid dynamic memory allocation, optimize the memory mapping and eventually ensure a target's resources are always sufficient for exception-free program execution.

(C7) Control-flow integrated error signal handling: Language constructs to signal errors and handle signaled errors using ordinary control-flow conditions. All potentially signaled errors must be handled or explicitly exposed to the runtime environment; they cannot slip through unnoticed. Automatic error signal propagation enables delayed error handling, avoiding the need for immediate checks of each operation that might fail.

(C8) Safe floating-point numerics: Guaranteed quiet NaN and infinity propagation according to IEEE 754-2019 (IEEE 2019-07), with relational operations signaling pre-defined errors when called with NaN arguments. Such integration with the error signaling concept of (C7) means, that NaNs can never slip through unnoticed.

(C9) Safe built-in functions: Rich set of safe built-in functions for casting, numeric limits, rounding, trigonometric operations, 1/2/3D interpolation, solving systems of linear equations etc. If arguments are out of

range, the error signaling of (C7) is used to denote so and returned values are precisely defined (typically NaN or infinity) to avoid undefined, implementation dependent, behavior; in line with (C8), NaN arguments are preferably silently propagated.

(C10) Call-by-value semantic with well-defined side effects: Function arguments are passed by value; and, although an imperative language, GALEC has well-defined side-effect rules that guarantee (1) each expression is free of competing side effects and (2) which statements are mutual independent. Production code generators can leverage on these characteristics for automatic, lock-free program parallelization and to avoid unnecessary copying of multi-dimensional values.

(C11) Block life-cycle with well-defined layers of modification: GALEC supports a layered modification concept distinguishing constants from semi-constant tunable parameters from dependent parameters, parameters from block in- and outputs and such from inner states. Only tunable parameters and inputs can be directly changed by the runtime environment, but only in-between two sampling steps, never while sampling. Whenever tunable parameters are changed, dependent parameters must be recomputed based on the new tunable parameters only (`Recalibrate()` interface function); dependencies on states, in- or outputs are forbidden. In addition, initialization is clearly encapsulated (`Startup()` interface function) with mandatory data-flow analyses guaranteeing every block variable is assigned an initial value based only on literal values or already initialized variables. Initialization code can contain arbitrary complex algorithms; its clear encapsulation enables static evaluation. The actual sampling code, which computes new outputs for given inputs considering the current block state, is encapsulated in the `DoStep()` interface function; its implementation must not change inputs nor parameters. All block interface functions automatically saturate variables with declared ranges (ranged variables) at the very beginning and ending of their execution. This guarantees, for example, that inputs and outputs are always within their ranges when `DoStep()` starts and terminates, yielding the behavior of a saturated controller. Not only block interface variables are saturated, but also inner states if respectively ranged, or ranged parameters when recalibrating. The whole block life cycle is formally defined via a state machine.

For details, readers are encouraged to consult the public alpha draft of the eFMI specification (EMPHYSIS 2021-07).

GALEC Example: To give at least a glimpse on how GALEC programs look like, particularly error handling, a short artificial example is given in the following. A typical GALEC block looks like the following (`[[...]]` denotes removed code snippets):

```
block Controller
  // Block interface variables:
  input  Real u[10] (min = -1.5, max = 1.5);
  output Real y[20] (min = -1.0, max = 1.0);
  parameter Real tP;    // tunable parameter
  parameter Real tV[20];// tunable parameter

protected
  // Internal block variables and functions:
  parameter Real dP; // dependent parameter
  Real M1[20,10]     // state
    (min = -1.0, max = 1.0);
  Real M2[10,20]     // state
    (min = -1.0, max = 1.0);
  function checked_transpose
    signals UNDERFLOW, NAN [[...]];
  function sum [[...]];

public
  // Block interface functions:
  method Recalibrate
    signals INVALID_ARGUMENT [[...]];
  method Startup [[...]];
  method DoStep
    signals NO_SOLUTION_FOUND [[...]];
end Controller;
```

First, the block interface variables that can be set (inputs and tunable parameters) and read (outputs) by the runtime environment are declared. Like any variable, such can be multi-dimensions and ranged. E.g., `y` is an output vector of size 20, with each of its elements in the range [-1.0, 1.0]. Then the section with the internal block variables and functions follows, first the dependent parameters, then the states and finally functions. Note, that any errors a function can signal to callees are part of its interface. `checked_transpose` for example can signal `UNDERFLOW` and `NAN` error signals. Finally, the section with the block interface functions follows. These are `Recalibrate()`, `Startup()` and `DoStep()`. Note, that in the example, a sampling step can signal that no solution has been found via the `NO_SOLUTION_FOUND` signal; the signal is used in the example to denote that a fallback controller has been used due to an unexpected error. The `INVALID_ARGUMENT` of the `Recalibrate()` function is used to denote to the runtime environment that given new tunable parameters are invalid and another recalibration is required. Of course, these error signals are just examples; the interface functions of other blocks may signal different, or no errors at all.

Assume `checked_transpose` is defined as follows:

```
function checked_transpose
  signals UNDERFLOW, NAN;
  input Real In[:, :];
  output Real Out[size(In, 2), size(In, 1)];
algorithm
  for i in 1 : size(I, 1) loop
    for j in 1 : size(I, 2) loop
      Out[j, i] := In[i, j];
      // Signals NAN if any argument is NAN:
```

```
      if absolute(In[i, j]) <
        epsReal() * self.dP
      then
        signal UNDERFLOW;
      end if;
    end for;
  end for;
end checked_transpose;
```

Its in- and output are any matrices of reversed dimensionality. If used in a context where the output is not an $n \times m$ matrix for an $m \times n$ input, static dimensionality analyses will fail with an error (cf. (C6)). The `for`-loop uses the matrix dimensions to traverse all elements; it computes the transpose of `In` in `Out`. Thereby every element is checked to be non-zero around an epsilon based on the target machine's minimal precision (`epsReal()` built-in function) and the block's dependent parameter `dP`; the latter is accessed directly via `self.dP`. If the check fails, `UNDERFLOW` is signaled. The `<` check itself will signal `NAN` if any of its arguments is NaN. This behavior is guaranteed by GALEC (cf. (C8)). Since neither of both signals is handled within `checked_transpose`, both must be exposed to callees as denoted in the function's interface (the `signals UNDERFLOW, NAN;` following the function name).

Assume the sampling function is:

```
method DoStep
  signals NO_SOLUTION_FOUND;
algorithm
  self.M1 := sum(self.u) /
    real(size(self.u, 1)) * self.M1;
  self.y := solveLinearEquations(
    self.dP * self.M1 * self.M2,
    self.tV);
  self.M2 := checked_transpose(self.M1);
  // Catch any error signals
  // or NaN/∞ in self.y:
  if signal or not(allFinite(self.y)) then
    [[ ...fallback controller code... ]]
    // Expose use of fallback controller:
    signal NO_SOLUTION_FOUND;
  end if;
end DoStep;
```

At the very end of all computations, a simple conditional control-flow checks for any kind of errors, and in case of any error, uses some fallback controller and signals its usage by exposing the `NO_SOLUTION_FOUND` signal to the runtime environment. This delayed error handling is achieved by the conditional:

```
 if signal or not(allFinite(self.y))
```

The `if signal` construct can be used to check for any, only specific or any except certain error signals (`if signal`, `if signal in E`$_1$`, E`$_2$`, …, E`$_n$ and `if signal not in E`$_1$`, E`$_2$`, …,E`$_n$ respectively). The body of the check is executed if any of the checked signals was set; if so, the signals are automatically unset. In the example's case, all error signals are handled. The `or` condition is optional; it is used in the example to check if any value of the block's output vector `y` is NaN or +/-∞ via the `allFinite` built-in function. Error signal checks are ordinary control-flow conditionals and can be combined with any other `if`, `elseif` and `else` conditioned branches. In the example, errors might be signaled by the `checked_transpose` call or the `solveLinearEquations` call. The latter built-in function fails with a predefined error if the linear equation system `Ax = b` cannot be solved, with `A` being its first argument, `b` the second and `x` its result. Note, that in the example, the first argument is computed using multi-dimensional arithmetics: the scalar dependent parameter `dP` is multiplied to the 20×10 matrix `M1`, the resulting 20×10 matrix in turn is multiplied to the 10×20 matrix `M2` yielding a quadratic 20×20 `A` matrix as required by `solveLinearEquations`. Finally, `DoStep()` will according to (C11) implicitly, at the very end, saturate the block output `y` and all elements of matrices `M1` and `M2` to be in the range [-1.0, 1.0], as it will implicitly saturate the block input `u` to be in range [-1.5, 1.5] at its very beginning (since these block variables are declared with ranges). Of course, only non-NaN values can be saturated; NaNs stay.

Tool challenges: A Modelica tool (or any other modeling tool) targeting GALEC for code generation can concentrate on the actual computation by leveraging on its high-level abstractions for multi-dimensional arithmetic, whereas embedded tooling benefits from the inherent language guarantees every GALEC program will satisfy and these are of uttermost importance for embedded software (like guaranteed termination with upper-bound of algorithmic steps or exception-freeness). Of course, a major challenge for Modelica tools is to actually find an upper-bounded causal solution for a given acausal equation system; this is a non-trivial task with many challenges on developing suited integration schemes. Once developed, respective approaches are however naturally/conveniently expressed as GALEC programs.

2.5 Production Code Model Representation

The previously described formal representation of a control algorithm in the GALEC language must be transformed into executable code for a target machine. For the generation of this code (production code) there are many degrees of freedom. In contrast to FMI, eFMI does not enforce a strict code and API format but allows the actual Production Code representation of an algorithm to be adjusted to the context in which the code is to be integrated into. The container architecture of an eFMU allows to hold several such Production Code model representations. An integrator of production code can pick the one that is suitable for his integration context.

The integration context determines several characteristics including the available bit size (e.g. integer as well as single or double precision floating-point), different data interfaces (e.g. functions without arguments working on global data vs. functions with arguments), target and compiler specific optimization, as well as completely different platforms (such as e.g. AUTOSAR vs. plain C code). The different production codes for the same GALEC program and their integration contexts are described in their respective manifest. Each manifest contains all information to enable an integration of its production code into a test or execution environment.

We would like to illustrate the impact of the integration context onto the actual production code by giving alternative C18 (ISO/IEC 2018-06) code realizations for the GALEC example of Section 2.4. In the first "software architecture", all block variables are realized as global variables:

```
float u[10];
float y[20];

float tP;
float tV[20];
float dp;
float M1[20][10];
float M2[10][20];

unsigned int DoStep(void)
{
  unsigned int signals = 0U;
  [[...]]
  return signals; /* NO_SOLUTION_FOUND? */
}
```

In the second architecture below, the input-output notion of the block is mapped to an input/output of the `DoStep` function itself and the block state (parameters and inner states) is encapsulated in a passed `struct` which must be allocated by the embedded runtime environment (note that the block state and output are passed by reference, thus are writeable by `DoStep`):

```
typedef struct
{
  unsigned int signals;
  float tP;
  float tV[20];
  float dp;
  float M1[20][10];
  float M2[10][20];
} BlockState;

void DoStep(
  BlockState* const state,
  const float const u[10],
  const float y[20])
{
  state->signals = 0U;
  [[...]]
}
```

This scheme allows multiple, independent instances of the block to be allocated by the runtime environment (`DoStep` itself is stateless).

A third architecture could be the AUTOSAR Classic Platform. Here, variables are accessed using macros provided by a centrally generated middleware.

Note, that independent of the software architecture, many characteristics of GALEC (e.g. no need for dynamic memory allocation, bounded execution time with bounded loop iterations) are intrinsically also properties of derived production code.

Other characteristics such as the handling of error signals or the support of multi-dimensional arithmetic let more room for production code generating tools to exploit different solution alternatives and are not straightforward. For multi-dimensional arithmetic, specific libraries could be included, and for error signal handling a low-level mapping using bit masking logic can be performed in the transformation process from GALEC to production code.

The representation of error signals using bit-masking logic, for example, allows for fast (simultaneous) check of several conditions and the efficient setting and resetting of all concerned signal values. A GALEC snippet like

```
if signal in OVERFLOW, NAN then
  y := y + 1.0;
end if;
```

could be translated into C18 production code like

```
if ((signals & 0x6U) != 0U)
{
  /* First, reset the checked signals: */
  signals = signals & 0xfffffff9U;
  /* Then, proceed with body: */
  y = y + 1.0F;
}
```

with proper encoding of the signal values for `OVERFLOW` and `NAN` in bit positions 2 and 3.

For multi-dimensional arithmetic and interpolation routines, usually libraries optimized for the target platform will be used. The production code generator has to make sure that used data structures of matrices and vectors match the format of the used libraries. Furthermore, the production code generator has to take care that the value-semantic of the GALEC language is preserved by taking adequate precaution like copying data when using library algorithms that alter the input data (e.g. when solving linear systems). Data flow analysis on the GALEC program enables optimizations that can avoid unneeded copy operations. Operations that are "simple/atomic" in GALEC code (like chained arithmetic expressions on multidimensional elements) may require a "flattening" in the generated production code and a proper non-trivial management of intermediate results.

In the example of Section 2.4, the multi-dimensional arithmetic expression

```
self.y := solveLinearEquations(
  self.dP * self.M1 * self.M2,
  self.tV)
```

for example requires to store the intermediate result of multiplying the scalar `dP` with the 20×10 matrix `M1`; the resulting 20×10 matrix has to be multiplied with the 10×20 matrix `M2`, yielding a temporary 20×20 matrix which is passed as `A` argument to `solveLinearEquations`.

Other optimizations on the production code generator side include analysis of value ranges to be able to omit unnecessary saturation operations for in-, outputs and states in case it can be determined that the values are always within bounds.

Another aspect of the production code generation step is to make the generated production code accessible and interpretable by consumers of it in subsequent phases like testing or integrating into an ECU SW. With the large degrees of freedom in generating production code for a given GALEC program, the structure of the production code may vary greatly, but must be described unambiguously at least in the interface parts and must be made accessible to anyone who would like to interact with it. This is made possible by the Production Code manifest, which precisely describes the code structure and interfaces (e.g. types, variables, functions) as well as their association to the corresponding elements of the GALEC code. This association is important to map information available only on the GALEC level also to the production code elements and enable traceability of the multi-step generation process. For example, stimulation data in a Behavioral Model container that is mapped to GALEC block variables can be applied also to their counterparts in the production code, or attributes of these variables (like ranges, units) can be associated to their respective production code counterparts. The required cross-referencing between different eFMU containers, e.g., to the manifest of the Algorithm Code container, uses a unified referencing scheme.

Besides the integration interface, Production Code manifests give, for example, a precise description of the target (like target language, target platform, target type, compiler and linker options) and the code files that make up the production code. The content of such code files is described in terms of XML elements and attributes like *Includes*, *TypeDefs*, *Macros*, *Variables* and *Functions* with *FormalParameter* and *ReturnParameter*, including both a mapping to target specific realizations (e.g. target types) as well as a “backward” reference to the corresponding elements in the Algorithm Code container.

With the help of the meta information of Production Code manifests, other widely used standards like AUTOSAR and FMI can be supported. In case of an AUTOSAR platform, the code files are complemented with specific description files that contain all information to integrate the production code w.r.t. the used AUTOSAR standard. In case of the AUTOSAR Classic Platform for example, such description files are the `.arxml` files shipped with the software component.

2.6 Binary Code Model Representation

The eFMI Binary Code model representation contains binaries that have been derived from a Production Code representation for a dedicated target architecture. It mainly serves two purposes:

1. Support the creation (build process) and integration of binaries on an embedded ECU target.
2. Protect intellectual properties when software artifacts are shared in a collaborative development process with multiple parties.

The first purpose is achieved by providing (a) the actual binaries and (b) the relevant build information like compilation and linking steps to create these binaries for a certain target platform. (b) is done in the manifest of the respective Binary Code container and can also be used to rebuild the binaries in case they are stale due to later production code changes, whereas production code cross-referencing with mandatory checksums enables to automatically deduce if binaries are stale. Note, that existing compiler and linker information of production code manifests can be referenced and further refined by the manifests of Binary Code containers, enabling a stepwise specialization and dedication towards a target platform. Integrators can use the build information to integrate the binaries on their embedded target ECUs. Additionally, the manifest can list run time compliance information such as execution times and further information relevant for the integration (e.g., a calibration file describing memory addresses and value ranges for calibration).

Intellectual property protection is achieved by removing the source code of the Algorithm Code and Production Code containers a Binary Code container is derived from, such that they are left with their manifest files only. Doing so, binary implementations can be shared without exposing actual source codes to third parties, whereas the meta information required for embedded software integration are still provided by the manifests.

An example for the stepwise derivation of dedicated binaries is the AEBS demonstrator mentioned in Section 4.1, where a generic production code is refined with integration code for the AUTOSAR Adaptive Platform, from which eventually platform-specific binaries with accompanying AUTOSAR Adaptive Platform manifests are generated.

3 eFMI Readiness

3.1 eFMI Tool Support

The EMPHYSIS Consortium (EMPHYSIS 2021) with its 25 partners from Belgium, Canada, France, Germany and

Sweden covered the entire value chain from vendors of modeling and simulation tools, code generators and V&V tools over embedded software developers and integrators to automotive Tier 1 suppliers and OEMs. This allowed to develop the eFMI specification along with reference implementations that have been thoroughly tested (cf. Section 3.2) and applied to challenging industrial applications (cf. Section 4).

By the end of the project in February 2021, already 13 different tools covering the entire eFMI workflow plus the open source eFMI Compliance Checker were available as prototypes. Soon after the official release of the eFMI Standard, these tools are expected to be available on the market.

3.2 Test Cases and Coverage

A set of dedicated test cases has been extensively used for testing the eFMI workflow with implementations in different prototype tools during the EMPHYSIS project. Most of the test cases are part of the Modelica library *eFMI_TestCases* that has recently been published under a 3-Clause BSD license (Modelica Association 2021-07). A few other test cases are AMEsim models or manually implemented Algorithm Code containers. For each of the test cases automatically generated reference results are provided in respective Behavioral Model containers.

By altogether 48 test cases (including variants) the following partially very advanced features are covered: non-linear inverse models, feedback linearization based controllers, explicit and implicit integration schemes, event-based re-initialization of continuous states, neural networks, error handling, implicit saturation and important built-in functions like solving linear equation systems as well as 1-D and 2-D interpolation tables. Each feature is supported by at least one eFMI prototype tool generating Algorithm Code containers.

All generated Algorithm Code containers have been successfully imported by the involved production code tools. For each Algorithm Code container, two production code variants have been generated: A double precision floating-point (64-bit) and a single precision floating-point (32-bit) version, each with respective implementations of higher-level built-in functions.

A testing tool chain has been set up to automatically check all generated production code variants, create test harnesses, compile the code, execute it and compare the results with the reference results contained in the Behavioral Model containers of each eFMU with respect to given error tolerances. In total, 538 execution runs are necessary to assess all production code variants generated by the varying combination of tools along the eFMI workflow. More than 96% of these runs successfully passed. The unsuccessful tests are all a result of a currently incomplete initialization mechanism in one test case and its variations that will be the subject of investigation in future work. Nevertheless, the very positive test rate impressively shows the maturity of the tool prototypes and their compatibility.

3.3 Performance Benchmarks

Performance benchmarks of the generated production code against state of the art manually implemented C solutions have been conducted. The target was the Bosch Multicore ECU MDG1 (Rüger et al. 2014). Six test cases addressing known difficulties of physical models on ECUs by using automatic model to code transformations have been contributed to the *eFMI_TestCases* library. In the following, these test cases are denoted by their IDs in *eFMI_TestCases*; the addressed challenges, in ascending order w.r.t. difficulty, are:

- DC motor speed control with PID controller (*M03_B*): Minimal footprint of code with saturated inputs and outputs.
- Air system controller (*M15_A*): Stiff ODE with delay operator.
- Drivetrain torque controller based on inverse model (*M04_A*): Linear inverse physical model.
- Inverse slider crank (*M10_B*): Non-linear inverse physical model (DAE Index-1).
- Reduced order model of a thermal heat transfer (*M16_A*): Efficient handling of matrix operations and large two-dimensional maps.
- Ideal rectifier (*M14_A/B*): Advanced symbolic transformation to derive a compact state space form.

All models are tested in an open loop setup using the recorded data from their Behavioral Model container as stimulus. The execution time is captured based on the CPU ticks elapsed, right from the start of calling the model interface function (e.g. `DoStep`) until the function execution is completed (note, that the MDG1 ECU enables precise and reliable counting of elapsed CPU cycles without any caching effects). As they have a significant impact, boundary and error checks are considered. Boundary checks saturate the in- and outputs to their limit values. Error handling will check for non-plausible values like NaN and infinity. More details are provided in Armugham et al. (2021).

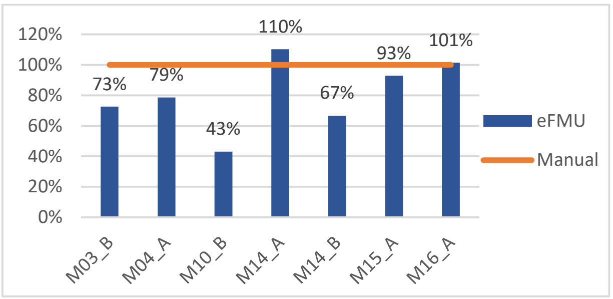

Figure 3. Run time measurements of eFMU production code with respect to manually coded solutions

The results of the performance benchmarks are shown in Figure 3. In 4 out of 6 examples (counting M14_A and M14_B as one), there is at least one eFMI tool chain setup that outperforms the manual implementation.

In case of *M10_B*, the manually derived solution of the inverse slider crank mechanism did not show a stable behavior unless two of the state variables were computed in double precision, while the auto-generated solution worked fine in single precision due to a more appropriate state selection.

The eFMU derived from the component-oriented rectifier model (*M14_A*) did not lead to the desired most compact and efficient formulation of the problem but gave very good results after reformulating the problem in the Modelica code (*M14_B*).

The reduced order model (ROM) of a thermal heat transfer test case (*M16_A*) has been processed by the tool chain starting from a manual implementation of the matrix equation system in GALEC code. As discussed in Agosta, et al. (2019), rigid scalarization, as applied by today's Modelica compilers, leads to an undesired code expansion. Here is room for improvement of the GALEC code generating tools. However, as the results show, the GALEC language is expressive enough to formulate this type of problem in a way that can be handled by the production code generating tools in a highly efficient way (the manually written GALEC code leverages on multi-dimensional arithmetic to avoid scalarization of the two-dimensional maps of the test case).

3.4 Code Quality Assessment

For all the test cases in Section 3.1, the code quality of the generated C code has been assessed by a static code analysis tool to find runtime issues such as variable overflows, possible division by zero, array index out of bounds, etc. or prove their absence. Also, the compliance of the code with the MISRA C:2012 rules has been checked. A static analysis is sound, but not necessarily complete. Hence, checked errors and rule violations are never overlooked, but may yield false alarms. Manual inspections resolved many of the false alarms so that in the 402 Production Code containers finally only 1% definite errors and 9% rule violations were detected. The main part of rule violations was detected in the implementations of built-in functions not being in the focus so far. It was assumed, that target-specific libraries realizing built-in functions will be used. Since then the tool prototypes have been further improved aiming for a full coverage of the MISRA C:2012 rules relevant for generated code.

3.5 Gain in Productivity

Aiming to put the time saving of an automated tool chain into perspective of the overall development effort of an embedded function, the working hours for modeling, implementation in C and validation of the results on the ECU have been counted for both the eFMI workflow and the manual development for the six benchmark examples.

The comparison of the results shows that in those cases based on a component-oriented modeling (*M03_B*, *M04_A* and *M10_A*) with a high level of reuse, the eFMI workflow took about 10 times less effort. For *M15_A* and *M16_A* the models have been implemented from scratch in Modelica based on a known state space formulation, but still gave a gain by a factor of 2.0 and 1.2 respectively. This stresses the high business value of eFMI for embedded software development especially for advanced physics-based control functions.

4 eFMI Applications

4.1 EMPHYSIS Demonstrators

The developed demonstrators, presented to the ITEA review board on Feb. 10, 2021 and summarized in the final demonstrator report of EMPHYSIS (2021-08), illustrate the application of the eFMI tool chain in concrete and realistic usage scenarios. These cover the domains vehicle dynamics, powertrain (internal combustion engine, battery electric vehicle, hybrid electric vehicle) and thermal systems and they are applied to advanced non-linear controllers, model-based diagnosis, virtual sensors and HiL simulation.

Renault demonstrated in two applications how a neural network trained by a high-fidelity model can be integrated as very accurate approximation into the embedded software running on a car by using the eFMI tool chain.

DLR-SR realized an advanced vertical dynamics controller and observer for semi-active damping (see Figure 4) using an inverse non-linear model and a non-linear Kalman filter running on a small series ECU in real driving tests. Never before for the institute, C code derived from a Modelica model has been directly integrated into the application software as in this case from the generated eFMI Production Code container.

GIPSA-lab demonstrated how eFMI can be utilized to derive a parametric Non-linear Model Predictive Controller (pNMPC) and deploy its production code to a dSPACE MicroAutoBox II ECU. The developed controller uses 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; this prediction is used for suspension control.

Volvo Cars demonstrated the development of an embedded 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. The transmission model physics comprise non-linearities and discrete events for handling brake-torques at low speeds, resulting in a stiff discontinuous system with mixed equations that has been successfully

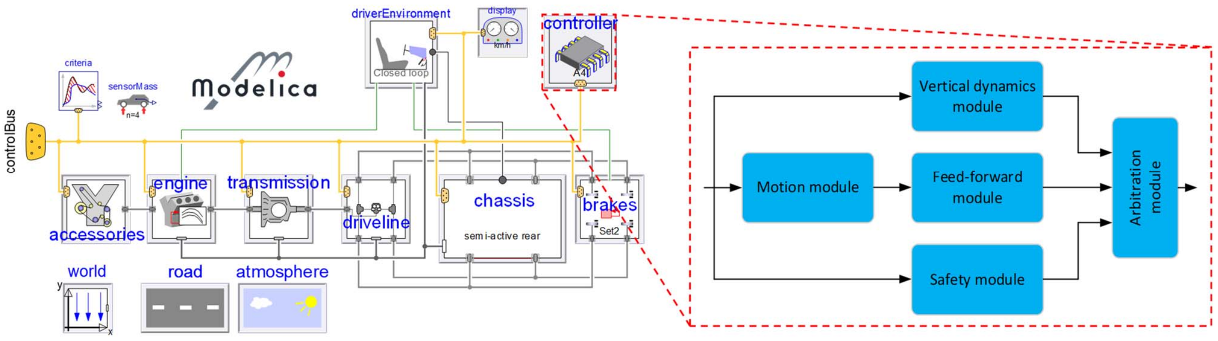

Figure 4. High fidelity vehicle model and advanced non-linear semi-active damping controller.

transformed by Modelica-tooling to a real-time suited GALEC solution.

Dassault Systèmes demonstrated the generation and validation of an AUTOSAR Adaptive Platform component starting from a Modelica model via a seamless, eFMI-based tool chain, for an advanced emergency braking (AEBS) controller. The AEBS controller is modeled in a classic block-diagram style with embedded physics. The blocks include enabled subsystems and signal locks, whereas the side effects of such are correctly handled using Modelica state machines.

4.2 OEM Advisory Board Feedback: Dual-clutch Transmission Demonstrator

The EMPHYSIS project has been accompanied by the so-called OEM Advisory Board with representatives from European and Japanese automotive OEMs. During half-day workshops, intermediate results of the project have been presented and discussed. An OEM Advisory Board usage scenario – a virtual sensor for a dual-clutch transmission – has been defined and a corresponding demonstrator implemented and evaluated in close collaboration between EMPHYSIS partners and the experts at Mercedes-Benz AG that provided the plant model of the dual-clutch transmission.

The objective of the virtual sensor is to use the physical model of a dual-clutch transmission to estimate the torque of clutches during shifting to avoid, for example, clutch over-burn and improve the driving-comfort during transmission shifting. The used transmission model had been derived from an existing high-fidelity system simulation model used in the product development of Mercedes-Benz AG; the most challenging system properties for a real-time application are therefore preserved. This includes the stiff dynamics of a hydraulic piston being tightly coupled with the discontinuous mode switching behavior of the clutches due to Coulomb friction, yielding a mixed equation system with undesired jittering even at a very small step-size of 0.1 ms with Explicit Euler.

By using a Rosenbrock method of order 1 (Hairer 1996) this problem could be drastically relaxed towards a jitter free behavior at a fixed step-size of 0.1 ms and robust but slightly jittering at a step-size of 10 ms. Compared to Explicit Euler, the Rosenbrock method therefore enables a factor 100 lower sampling rate, enabling the usage of the dual-clutch transmission model for embedded[1] real-time simulation for the very first time.

According to Mercedes-Benz AG, this result was considered a big progress towards using eFMI to derive very accurate plant models for SiL, HiL and embedded observer applications in a seamless fashion from a high-fidelity system simulation. It was confirmed that there is currently no better automated solution available for this task. As of today, a dedicated real-time model must be derived and individually fitted for each application causing significant repeated effort.

The eFMI container architecture with its built-in traceability and safety mechanisms has been praised by all members of the OEM Advisory Board as making eFMI a promising candidate to become the prescribed format for embedded software deliverables in OEM supplier collaborations. Especially for advanced functions like observers, the proposed eFMI workflow was considered as game changing technology to revolutionize the embedded software development.

5 Future Work

From the very beginning of the EMPHYSIS project (Lenord, 2019), also an Equation Code model representation has been investigated as an optional first intermediate target representation for acausal equation-based modeling tools. The motivation is that on the one hand Algorithm Code model representations can be generated from such a standardized, universal – but still simple – equation language, whereas on the other hand further equation analysis tooling could be integrated to

[1] The model has been used for real-time simulation by Mercedes-Benz AG before, but not for developing a software solution that can be deployed on the embedded device; this became possible only with eFMI.

refine equations or derive system characteristics of interest in the embedded domain like fault-behavior/safety, numeric stability etc.

To that end, a collaborative working group between EMPHYSIS partners and the Modelica Language working group has been formed, with the objective to define a proposal of a standardized *Flat Modelica* language as basis for a more restricted equation code language. A subset of Modelica keywords and a modified grammar have been proposed (Modelica Association 2021-06).

By implementing a prototypical *Flat Modelica* parser and pretty-printer for a non-Modelica tool within a few person months, it was demonstrated that other, already existing equation-based modeling tools (with their existing model representations, analyses capabilities and code generation back ends) can be integrated into the acausal modeling process with comparatively small effort. This early prototype tooling has been applied to two Bosch use cases: (1) inversion of a plant model of a drivetrain and (2) structural analysis of a thermal system to evaluate the detectability of system faults (EMPHYSIS 2021-08).

The work on a standardized *Flat Modelica* language and Equation Code model representation is planned to be continued and incorporated into a later version of the eFMI Standard.

6 Conclusions

This paper presented eFMI, a new workflow and open standard for the automatic generation of embedded software from physical models. The novelty of eFMI is its tooling-open, standardized exchange format by means of a container architecture with various standardized, traceable model representations for behavioral reference results, abstract algorithmic solutions, actual production codes and target-specific binary codes, bridging the gap between physics-modeling and embedded software.

A broad set of test cases, including technically challenging models, has been used to rigorously test and crosscheck the developed prototypical eFMI tools and their interoperability in the eFMI workflow. Together with performance benchmarks and code quality assessments, a high level of maturity has been testified.

The eFMI container architecture, with its various model representations, has been successfully applied to industrial usage scenarios. Automotive OEMs and Tier 1 suppliers confirmed the benefits of the proposed workflow over the state-of-the-art development processes in terms of repeatability, traceability and overall gain in productivity for embedded software development.

The work of EMPHYSIS and the eFMI Standard is continued in a new Modelica Association Project eFMI (MAP eFMI), successfully founded by core partners of the EMPHYSIS project. The work to further develop the eFMI specification towards a first official release according to established Modelica Association processes has already started. Companies and other organizations are encouraged to join MAP eFMI, leverage on the already developed tooling and foster the eFMI ecosystem.

Acknowledgements

This work is the result of the European ITEA3 Call2 EMPHYSIS (project number 15016). The work was funded by the German *Federal Ministry of Education and Research* (BMBF, grant numbers 01|S17023(A-H)), *Sweden's Innovation Agency* (VINNOVA, project number: 2017-05121), the French *Directorate General for Enterprise* and the Belgian agency *Flanders Innovation & Entrepreneurship*. The authors are responsible for the content of this publication.

The authors would like to thank the members of the EMPHYSIS OEM Advisory Board: BMW, Mercedes-Benz AG, Mazda, Volvo Trucks, JSAE for their use case, feedback and directions and the numerous persons that have worked in the EMPHYSIS project on the development of the eFMI Standard and/or evaluated it with tool prototypes, benchmarks and test cases as listed in appendix A of the eFMI specification.

References

Agosta, Giovanni, Emanuele Baldino, Francesco Casella, Stefano Cherubin, Alberto Leva and Federico Terraneo (2019). "Towards a High-Performance Modelica Compiler." In: *Proceedings of the 13th International Modelica Conference*. Modelica Association, pp. 313–320. DOI: 10.3384/ecp19157313.

Armugham, Siva Sankar, Christian Bertsch, Karthikeyan Ramachandran, Oliver Lenord and Kai Werther (2021). "eFMI (FMI for embedded systems) in AUTOSAR for Next Generation Automotive Software Development". In: *Symposium on International Automotive Technology 2021*. SAE International. Accepted March 31, 2021.

AUTOSAR Consortium (2021). *AUTOSAR (AUTomotive Open System ARchitecture)*. URL: http://www.autosar.org/.

Bertsch, Christian, Jonathan Neudorfer, Elmar Ahle, Siva Sankar Arumugham, Karthikeyan Ramachandran and Andreas Thuy (2015). "FMI for Physical Models on Automotive Embedded Targets". In: *Proceedings of the 11th International Modelica Conference*. Modelica Association, pp. 43–50. DOI: 10.3384/ecp1511843.

EMPHYSIS (2021). *EMPHYSIS (Embedded systems with physical models in the production code software)*. URL: https://emphysis.github.io/.

EMPHYSIS (2021-07). *Functional Mock-Up Interface for embedded systems (eFMI)*. Version 1.0.0-alpha.4, Tech. rep.: EMPHYSIS Consortium. URL: https://emphysis.github.io/downloads.

EMPHYSIS (2021-08). *eFMI for Physics-Based ECU Controllers*. Tech. rep. D7.9, EMPHYSIS project deliverables: EMPHYSIS Consortium. URL: https://emphysis.github.io/downloads.

Englert, Tobias, Andreas Völz, Felix Mesmer, Sönke Rhein and Knut Graichen (2019). "A Software Framework for Embedded Nonlinear Model Predictive Control Using a Gradient-Based Augmented Lagrangian Approach

(GRAMPC)". In: *Optimization and Engineering* 20 (3), pp. 769–809. Springer. DOI: 10.1007/s11081-018-9417-2.

Hairer, Ernst and Gerhard Wanner (1996). *Solving Ordinary Differential Equations II*. 2nd ed. Springer. ISBN: 978-3-540-60452-5.

IEEE (2019-07). *IEEE Standard for Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers. ISBN: 978-1-5044-5924-2.

Intel Corporation (2021-06). *Intel® 64 and IA-32 Architectures Software Developer's Manual – Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*. Tech. rep.: Intel Corporation. Order Number: 325462-075US.

ISO/IEC (2018-06). *ISO/IEC 9899:2018 — Information technology — Programming languages — C*. International Organization for Standardization.

Lenord, Oliver (2019). "Standardizing eFMI for Embedded Systems with Physical Models in the Production Code Software". Presented at: *Jubilee Symposium: Future Directions of System Modeling and Simulation*. Medicon Village, Lund, Sweden, September 30, 2019. URL: https://modelica.github.io/Symposium2019/.

MISRA (2013-03). *MISRA C:2012 – Guidelines for the use of the C language in critical systems*. MISRA Consortium Limited. ISBN: 978-1-906400-10-1.

Modelica Association (2021-02). *Modelica® – A Unified Object-Oriented Language for Systems Modeling – Language Specification – Version 3.5*. Tech. rep.: Modelica Association. URL: https://modelica.org/documents/MLS.pdf.

Modelica Association (2021-04). *Functional Mock-up Interface for Model Exchange and Co-Simulation*. Tech. rep.: Modelica Association. URL: https://fmi-standard.org/downloads/.

Modelica Association (2021-06). "Modelica Language Change Proposal 31 (MCP 31)". URL: https://github.com/modelica/ModelicaSpecification/tree/MCP/0031/RationaleMCP/0031.

Modelica Association (2021-07). "Official eFMI test cases for demonstrating and evaluating eFMI tooling". URL: https://github.com/modelica/efmi-testcases.

Neudorfer, Jonathan, Siva Sankar Armugham, Mathews Peter, Naresh Mandipalli, Karthikeyan Ramachandran, Christian Bertsch and Isidro Corral (2017). "FMI for Physics-Based Models on AUTOSAR Platforms". In: *Symposium on International Automotive Technology 2017*. SAE International. DOI: 10.4271/2017-26-0358.

Rüger, Johannes-Joerg, Alexander Wernet, Hasan-Ferit Kececi and Thomas Thiel (2014). "MDG1: The New, Scalable, and Powerful ECU Platform from Bosch". In: *Proceedings of the FISITA 2012 World Automotive Congress*. Vol. 6. Vehicle Electronics. Springer.

Wagner, Alexandre, Thomas Bleile, Slobodanka Lux and Christian Fleck (2009). "Method for real time capability simulation of an air system model of an internal combustion engine". Patent: United States US8321172B2, filed November 19, 2009.

Zimmermann, Michael, Thomas Bleile, Friedrun Heiber and Alexander Henle (2015). "Komplexitätsbeherrschung von Motorsteuerungs-Funktionalitäten". In: *MTZ - Motortechnische Zeitschrift* 76 (1), pp. 60–64. Springer. DOI: 10.1007/s35146-014-2003-z.