D3.3	# Concept for customizing code generation including flexible adaptation to different target systems
Access[1]:	**PU**
Type[2]:	**Report**
Version:	**1.0**
Due Dates[3]:	**M12**

Open Cyber-Physical System Model-Driven Certified Development

Executive summary[4]:

This deliverable describes a concept for code generation from the Modelica language, particularly aiming for allowing a flexible adaptation of the generated code to different target systems. The concept aims at using the Functional Mock-up Interface standard (FMI v2.0) for the interface of the generated code. It discusses code requirements due to targeting restricted embedded systems and the necessity of real-time execution, as well as trade-offs which are related to the reusability of the generated code.

[1]Access classification as per definitions in PCA; PU = Public, CO = Confidential. Access classification per deliverable stated in FPP.
[2]Deliverable type according to FPP, note that all non-report deliverables must be accompanied by a deliverable report.
[3]Due month(s) according to FPP.
[4]It is mandatory to provide an executive summary for each deliverable.

Deliverable Contributors:

	Name	Organisation	Primary role in project	Main Author(s)[5]
Deliverable Leader[6]	Bernhard Thiele	SICSEast	WP3 Leader	X
Contributing Author(s)[7]	François Beaude	RTE	T5.3 leader	
Internal Reviewer(s)[8]	Adrian Pop	SICSEast	WP5 leader	
	Magnus Eek	Saab AB	Project Coordinator	

Document History:

Version	Date	Reason for change	Status[9]
0.1	04/11/2016	First Draft	Draft
0.2	08/11/2016	Integrated improvements by François Beaude	In Review
1.0	09/11/2016	Integrated comments by reviewers	Released

[5] Indicate Main Author(s) with an “X” in this column.
[6] Deliverable leader according to FPP, role definition in PCA.
[7] Person(s) from contributing partners for the deliverable, expected contributing partners stated in FPP.
[8] Typically person(s) with appropriate expertise to assess deliverable structure and quality.
[9] Status = “Draft”, “In Review”, “Released”.

Contents

Acronyms

ACG	Acyclic Causality Graph.
CPS	Cyber-Physical Systems.
DAE	Differential Algebraic Equation.
DLL	Dynamic Link Library.
FMI	Functional Mock-up Interface.
FMU	Functional Mock-up Unit.
FPU	Floating-Point Unit.
MBD	Model-Based Development.
MBSE	Model-Based Systems Engineering.
OMC	OpenModelica Compiler.
SW-C	Software Component.

1 Introduction

This deliverable describes a concept for code generation from the Modelica language. It will be implemented in the OpenModelica tool. The deliverable is related to the OPENCPS D3.2 report "Translation validation and traceability concept from acausal hybrid models to generated code" within the same work package [TS16]. Both deliverables describe (different) aspects of the code generation from Modelica to C or C++.

Code generators transform a source language into a target language while preserving the semantics of the source language. In this report the source language is Modelica and the target languages are C and C++.

The OPENCPS project aims at improving methods and tools that support a Model-Based Systems Engineering (MBSE) approach for Cyber-Physical Systemss (CPSs). The envisioned improvements include multi-disciplinary simulation for early integration and validation of important system behavior aspects at a high-level system level, and a seamless refinement of high-level models until code for embedded systems can be generated. Going from a high-level system description to generated code that can be executed on a real system is a complex process involving multiple stages of various levels of abstraction which is depicted in Figure 1.

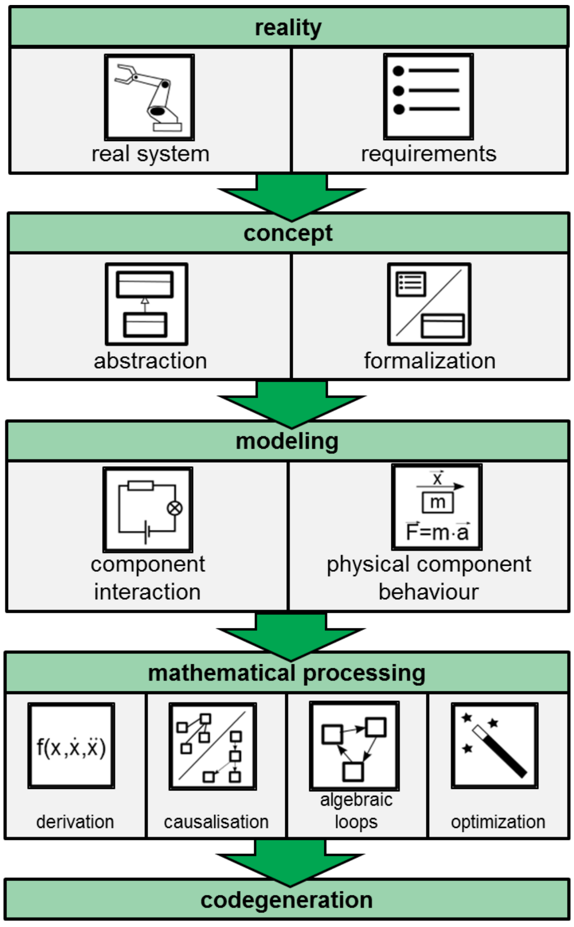


Figure 1: System modeling work-flow.

Modern control technology often uses a model of the physical process as part of advanced, digitally implemented control strategies. Modelica's excellent capabilities for physical modeling can be leveraged to synthesize such advanced controllers [AB06, TSGT08, TKOB05]. However, up to now there exists no qualified Modelica-based code generator that allows to directly use the generated code in safety-related (production) systems. The OPENCPS D3.2 report [TS16] proposes a concept for translation validation of Modelica-generated code which aims at increasing the confidence that can be placed in the generated code.

The goal of the present report is to propose a code generation concept that supports customizable code generation, so that the generated code can be easily adapted and integrated into different target systems. This concept is essentially orthogonal to the safety-related concepts developed in [TS16], *i.e.*, both concepts solve different problems but they can be (and will be) used together. Naturally, both proposal share the same state-of-the-art for translating acausal hybrid models which is already described in [TS16].

2 Background

This section presents selected background knowledge as a preparation for understanding the code generation concepts proposed in the follow-up Section 3.

2.1 Software Architecture

The following definition for a software architecture is taken from Clements et al. [CBB+10]:

> “The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.”

Software architectures are typically tailored for their application domain and address aspects that are sometimes referred to as *non-functional* requirements (*e.g.*, communication, efficiency, extensibility, maintainability, portability, reliability, scalability, scheduling, testability).

Generated code that is executed on an embedded system (in a CPS context) typically needs to integrate into an existing software architecture. Consequently, the code generator has to produce code that conforms, or is easily adaptable, to interfaces expected by the software architectures.

2.2 Functional Mock-up Interface (FMI)

Functional Mock-up Interface (FMI) is a well received tool independent standard to support both model exchange and co-simulation of dynamic models. While the FMI is primarily intended to provide a standardized exchange format for *physical* simulation models, the intention for being also usable for software components in embedded control systems was already stated in the abstract of the first version of the standard [FMI10]. In FMI terminology a system model that implements the interface defined by the FMI specification is called a Functional Mock-up Unit (FMU).

Thiele and Henriksson [TH11] explored the feasibility of using the initial version of FMI as an intermediate format in an AUTOSAR[10] software component development process. They developed a transformations between the XML schemas of the two standards which they utilized to automatically convert FMUs to AUTOSAR Software Component (SW-C). During the prototype development they also identified several missing features that they found worth considering for future versions of the standard.

More recently Bertsch et al. [BNA+15] revisited the idea of utilizing FMUs as SW-Cs on automotive embedded targets, by conducting an elaborate case study based on the current version of the standard (FMI v2.0, [FMI14]). One of the results of their work is a list of current limitation which they would like to address in a modified standard (which they term "FMI for automotive embedded systems").

For less restricted target hardware (PC compatible) FMI-based high-level process control optimization applications have already hit industrial production [Fra15]. The previous industrial application serves as a good example for the benefits of an open-source product like OpenModelica — industry applications can build on provided base functionality of the open-source product and extend it with desired specialized features [FWW+15].

Hence, despite current limitations in using FMUs for deeply embedded systems, experts believe in the potential of using FMI not only for co-simulation purposes, but also as a standardized and flexible means to integrate control-oriented SW-Cs into embedded software architectures.

2.3 Code Generation in OpenModelica

The foundations of translating Modelica models into simulation executables is already covered in Section 2 "Foundations of Modelica" of the OPENCPS D3.2 report [TS16]. For the present report it is expected that the reader is already familiar with the descriptions given there. The following sections will exceed these earlier descriptions with details which are relevant in the context of the present report.

2.3.1 Code Generation Phases

The translation of Modelica models in the OpenModelica Compiler (OMC) is divided in several phases (*cf.* [TS16, Section 2.2]):

Flattening *Lexical analysis* and *parsing*, *type checking*, *collapsing of the instance hierarchy* and *generation of connection equations* from connect equations. The result is a hybrid Differential Algebraic Equation (DAE) denoted as *Flat Modelica*.

Equation Transformation This step encompasses transforming and manipulating the equation system into a representation that can be efficiently solved by a numerical solver[11]. In OMC this representation is denoted as *DAELow*.

[10]AUTOSAR is an a development effort within the automotive industry with the goal of creating and establishing an open and standardized software architecture for automotive electronic control units

[11]Performed symbolic transformations include index reduction, matching, equation sorting, causalization, alias elimination, tearing, common subexpression elimination, *etc.*

SimCode Generation The final system of equations is transformed into an independent simulation code structure denoted as *SimCode*.

Code Generation A template-based code generator called *Susan* [FPSP09] supports code generation from SimCode. The standard target language is C, other supported targets include C++, JavaScript, and FMU.

Executable Generation The code is compiled and linked together with a corresponding *simulation run-time*. The simulation run-time is a library which is required to execute the generated model code. It contains the numerical solver required for the model simulation.

Simulation Execution of the (compiled) model. During execution, the simulation results are typically written into a file for later analysis.

2.3.2 Template Language Driven Code Generation

Figure 2 depicts OMC's approach of using templates for adapting the generated code. The text generation template language used in OMC is called Susan.

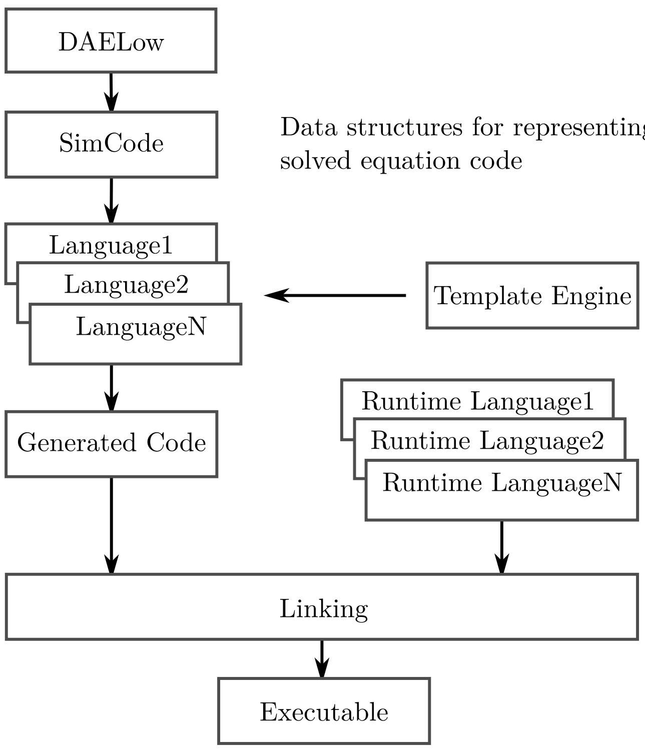


Figure 2: OpenModelica code generation using templates [FPSP09].

The Susan template language has been used for providing language support for different target languages besides the standard target language C, like C++, JavaScript and Java. Except for the C++ target the support for the alternative target languages is rather experimental.

Particularly the C++ target is an example for a code generation target which is strongly driven and adapted by industrial partners for creating customized solutions for their respective use cases [WM12, MKWM15, FWW$^+$15].

2.4 Relation to the State-of-the-Art

To the best of the authors' knowledge current Modelica tools have no designated support for exporting FMUs which are tailored for restricted embedded systems targets. Compared to the State-of-the-Art (SotA), the code-generation concept outlined in this report aims to provide designated support for producing FMUs which are *optimized for restricted embedded systems* while also providing entry points for a user to *customize the code-generation* for different target systems and use-cases.

3 Concept for Code Generation

This section describes a concept for providing improved code generation support from OpenModelica. The aim is

1. generating code in a format that can be easily integrated into existing software architectures, and
2. allowing users of the code generator to customize the code generation to their needs in a rather straight-forward manner.

3.1 Interface of Generated Code

The default interface for the code generator shall be FMI compatible. The FMI standard is already supported by many tools which facilitates the development process since it allows to import the generated code into already available tools for easy co-simulation with other systems and for testing purposes. Furthermore, the FMI format is quite suitable for integration into an existing software architecture since it provides a clean interface for which required wrapper functions can be generated in an automated way. A brief discussion of the state-of-the-art of using FMI in this context is provided in Section 2.2.

More specifically it is planned to generate code which uses the FMI v2.0, which specifies a *model exchange* and a *co-simulation* interface [FMI14]. Compared to the model exchange interface, the co-simulation interface has the advantage that it integrates more naturally into a typical execution scheme in which a digital control function is activated within a periodic task. The reason for this is that a co-simulation FMU basically contains everything which is needed for its execution and it suffices to provide it its required input values, call its step function `fmi2DoStep(..)`, and retrieve the output values.

In contrast, model exchange based FMUs require that the embedding environment provides suitable solvers, particularly for numerical integration of its continuous-time states. Hence, integration of model exchange FMUs has higher demands on the embedding environment. However, it has the advantage that it allows using solvers that have been optimized for a particular target platform. Another advantage of the model exchange interface is a clean execution model for handling discrete-time events based on a *super dense time* description (see [TS16, Section 2.5]). The handling of events is less safe if using co-simulation FMUs, because the (simpler) interface doesn't support more involved constellations.

The differences between model exchange and co-simulation start to blur if inline integration methods [EOC95] are used when generating model exchange FMUs. In that case, the FMU will not expose any continuous-time states and thus the embedding environment does not need to provide an integrator. Inline integration techniques are known to be well suited for real-time simulation applications and their favorable real-time capabilities make them also suitable choices for using them for real-time code on embedded targets. Indeed, standard methods for discretization of continuous-time controllers are closely related to inline integration techniques.

Despite the differences between the model exchange and the co-simulation interface, there are also many elements which they share. This suggests to support both interfaces, so that the more suitable option can be picked on an application specific basis.

3.2 Structure of Generated Code

Embedded target systems often have restriction that need to be considered when writing or generating code for them:

- (hard) real-time requirements
  $\Longrightarrow$ No algorithms with unpredictable upper execution time, *e.g.*,
  - no dynamic allocation of memory,
  - restrictions on event iteration,
  - restriction on iterative numerical methods to solve nonlinear equations,
  - restrictions on numerical solver methods,
- restricted amount of memory,
- restricted computational power,
- restricted availability of standard library functions (*e.g.*, not all functions from `math.h` might be available),
- restrictions on hardware support for floating point numbers (*e.g.*, only single precision support or no Floating-Point Unit (FPU) at all),
- cross-compiler specific restrictions and extensions,
- target specific low-level hardware aspects,
- code compliance to domain and project specific coding guidelines.

For the envisioned code generation it is neither the intention, nor is it feasible, to support every possible use case and target architecture. Instead, the code generator should produce real-time capable (language standard compliant) C/C++ code, which strives to be memory and run-time efficient without resorting to target specific low-level optimizations. It is expected that developers use the flexibility of the template language driven code generation to adapt the code generation to target specific requirements (see Section 2.3.2).

3.3 Variable Dependencies and Calling Sequences

Adapting an FMI (v2.0) compatible interface prescribes certain properties of the generated code, but it allows quite different internal implementation and optimizations. An important case is the implementation of the `fmi2SetXXX` and `fmi2GetXXX` functions. The implications will be explained in the following section using an example which is introduced below.

3.3.1 Running Example

The following discussion will assume that acausal Modelica equations have already been partly elaborated, so that a variable assignment has been made to every equation and an acyclic directed graph (acyclic *causality graph*) of the equation system is internally available in the tool[12]. Required algorithms are implemented in OMC (as well as in other Modelica tools). Further information about involved algorithms can be found in [Fri14, Part IV. Technology and Tools].

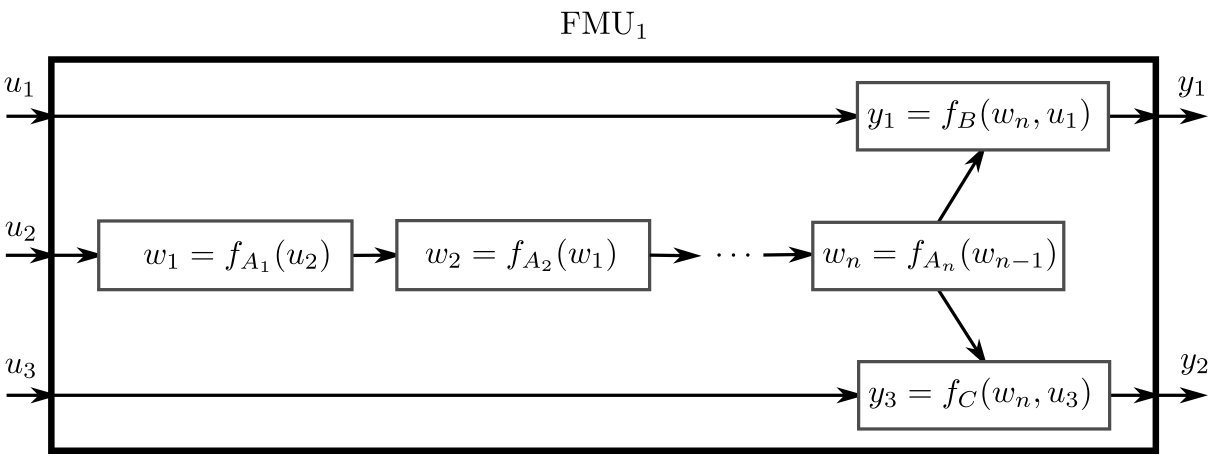
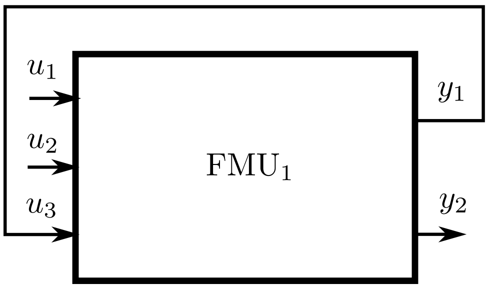


Figure 3: Running example FMU$_1$: Internal variable dependencies.

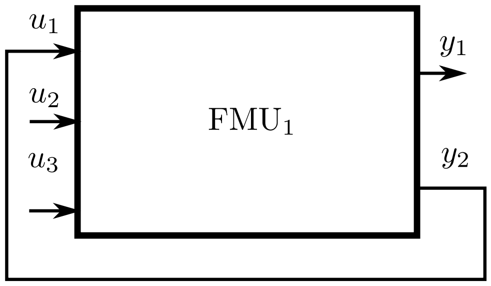
Figure 3 gives an example for internal variable dependencies within an FMU with three inputs (u_1, u_2, u_3), two outputs (y_1, y_2) and internal variables ($w_1, \cdots, w_n$)[13]. We assume that these dependencies have been derived from acausal Modelica equations by standard approaches as

[12] There are applications, *e.g.*, in the domain of power-system simulation (see RTE's use-cases and benchmark problems [Bea16]), for which one would like to generate acausal equations and employ an internalDAE solver which derives the causality internally. However, "DAE FMIs" are currently not supported by the FMI standard and the necessary research and development is beyond the scope of the work discussed here.

[13] The example is structurally similar to the example used by Lublinermann et al. [LST09, Figure 4] which facilitates the discussion and comparison to their work in Section 3.3.4



(a) Feed-back composition 1



(b) Feed-back composition 2

Figure 4: Two compositions of FMU$_1$ without algebraic loop.

mentioned above. These dependencies correspond to following equations:

$$y_1 = f_B(w_n, u_1) \tag{1a}$$
$$w_1 = f_{A_1}(u_2) \tag{1b}$$
$$w_2 = f_{A_2}(w_1) \tag{1c}$$
$$\vdots$$
$$w_n = f_{A_n}(w_{n-1}) \tag{1d}$$
$$y_2 = f_C(w_n, u_3), \tag{1e}$$

leading to following input and output dependencies on the FMU level

$$y_1 = f(u_1, u_2), \qquad y_2 = f(u_2, u_3). \tag{2}$$

These dependencies allow reusing the FMU in feed-back compositions without introducing an algebraic loop. Figure 4 shows the two possible algebraic-loop free feed-back configurations.

From the viewpoint of equation dependencies, FMU$_1$ can be reused in different algebraic-loop free feed-back configurations as shown in Figure 4. Notice, that feed-back composition 1 from Figure 4a will result in the input and output dependencies

$$y_1 = f(u_1, u_2), \qquad y_2 = f(u_1, u_2), \tag{3}$$

while feed-back composition 2 from Figure 4b will result in the input and output dependencies

$$y_1 = f(u_2, u_3), \qquad y_2 = f(u_2, u_3). \tag{4}$$

3.3.2 Straight-Forward Sequential Code-Generation

Code-generation needs to translate the equations into a sorted sequence of assignment statements. Depending of the particular code-generation approach this has implications on actual dependencies which may restrict the admissible compositions further than the theoretical dependencies. The following sequence of assignments is one valid sorting of system (1):

$$\begin{aligned}
w_1 &:= f_{A_1}(u_2) && \text{(5a)}\\
w_2 &:= f_{A_2}(w_1) && \text{(5b)}\\
&\vdots\\
w_n &:= f_{A_n}(w_{n-1}) && \text{(5c)}\\
y_1 &:= f_B(w_n, u_1) && \text{(5d)}\\
y_2 &:= f_C(w_n, u_3). && \text{(5e)}
\end{aligned}$$

Notice that sorting (5) allows for feed-back composition 1 (Figure 4a), but not for feed-back composition 2 (Figure 4b). Hence, the sorting induces the dependencies

$$y_1 = f(u_1, u_2), \qquad y_2 = f(u_1, u_2, u_3). \tag{6}$$

Another valid sorting of system (1) is

$$\begin{aligned}
w_1 &:= f_{A_1}(u_2) && \text{(7a)}\\
w_2 &:= f_{A_2}(w_1) && \text{(7b)}\\
&\vdots\\
w_n &:= f_{A_n}(w_{n-1}) && \text{(7c)}\\
y_2 &:= f_C(w_n, u_3) && \text{(7d)}\\
y_1 &:= f_B(w_n, u_1), && \text{(7e)}
\end{aligned}$$

which induces the dependencies

$$y_1 = f(u_1, u_2, u_3), \qquad y_2 = f(u_2, u_3) \tag{8}$$

and allows for feed-back composition 2, but not for feedback composition 1.

The FMI standard allows to specify variable dependency information in its FMI Description Schema (an XML file). The relevant element is named "`modelStructure`" [FMI14, p. 55]. For a straight-forward implementation [FMI14, p. 76] proposes to provide the dependency information not according to the "real" functional dependency, but according to the sorted equations in the generated code. For example, assuming sorting (5) we could generate the pseudo C-code shown in Listing 1. For brevity only one internal variable w_1 is considered, error handling is omitted, initialization is omitted, etc.

Listing 1: Pseudo C-code for fmi2SetXXX and fmi2GetXXX calls.

```
/* value references ordered according to sorted sequence of
   assignment statements */
#define FMU1_u1 0
#define FMU1_u2 1
#define FMU1_u3 2
#define FMU1_w1 3
#define FMU1_y1 4
#define FMU1_y2 5
#define FMU1_nvars 6
```

```c
#define FMU1_ninputs 3

typedef enum { fmi2OK, fmi2Warning, fmi2Discard, fmi2Error,
  fmi2Fatal, fmi2Pending } fmi2Status;

typedef struct {
    double time;
    double v[FMU1_nvars]; /* Array of variable values */
    /* Boolean condition array denoting if (cached) value of v[
       index] is available */
    int c[FMU1_nvars];
} FMU1;

fmi2Status FMU1_fmi2SetTime(FMU1* m, double time) {
  if (abs(time - m->time) > eps) {
    /* check whether time != m->time */
    m->time = time;
    /*  re-initialize caching of variables */
    for (int i=0; i<FMU1_nvars; i++ ) {
      m->c[i] = 0;
    }
  }
  return fmi2OK; /* omitting error checking */
}

fmi2Status FMU1_fmi2SetReal(FMU1* m, const unsigned int vr[],
                            int nvr, const double value[]) {
  for (int i=0; i<nvr; i++) {
    int vi = vr[i];
    if (m->c[vi] == 1) {
      /* re-setting of an already set input -> conservatively
         invalidate all cached non-input variables */
      for (int j=FMU1_ninputs; j<FMU1_nvars; j++ ) {
        m->c[j] = 0;
      }
    }
    m->v[vi] = value[i];
    m->c[vi] = 1;
  }
  return fmi2OK; /* omitting error checking */
}

fmi2Status FMU1_fmi2GetReal(FMU1* m, const unsigned int vr[],
                            int nvr, double value[]) {
  for (int i=0; i < nvr; i++) {
    int vi = vr[i];
    if (vi >= FMU1_w1 && c[vi] == 0) {
      if (c[FMU1_u2] == 0) return fmi2Error;
      m->v[FMU1_w1] = m->v[FMU1_u2];
      m->c[FMU1_w1] = 1;
    }
58      if (vi >= FMU1_y1 && c[vi] == 0) {
59        if (c[FMU1_u1] == 0) return fmi2Error;
60        m->v[FMU1_y1] = m->v[FMU1_w1] + m->v[FMU1_u1];
61        m->c[FMU1_y1] = 1;
62      }
63      if (vi >= FMU1_y2 && c[vi] == 0) {
64        if (c[FMU1_u3] == 0) return fmi2Error;
65        m->v[FMU1_y2] = m->v[FMU1_w1] + m->v[FMU1_u3];
66        m->c[FMU1_y2] = 1;
67      }
68      value[i] = m->v[vi];
69    }
70    return fmi2OK; /* omitting most error checking */
71  }
```

Listing 1 uses a caching strategy to avoid re-evaluation of already computed values. The evaluation logic in the `FMU1_fmi2GetReal` function requires that the value references for the variables are ordered according to the sorted sequence of assignment statements (5). The condition array `c` in line 18 indicates whether the value of a variable is already available. The array is re-initialized when the FMU time instant is different to the previous one (line 27). The if-then constructs check whether a variable is already cached before computing it (lines 53, 58, 63). Lines 54, 59, 64 check if the input required at that instant is set.

The advantages of this code-generation approach are

- a rather *straight-forward implementation*, and
- a *caching strategy* which prevents re-evaluation of already performed computations.

Disadvantages are

- *restricted reusability* of the FMU, since variable dependencies for the generated FMU are not necessarily equal to the dependencies that can be expected due to the equations,
- *arbitrariness*, since the utilized equation sorting algorithms will constrain the dependencies of the FMU in a rather arbitrary way,
- *extensive condition checking* (compare lines 53, 58, 63) can have negative impact on execution performance. A mitigation is to collect the computation of internal variables w_i into the branches of the output variables, *e.g.*, merge branches 53 and 58.

3.3.3 Current Approach in OpenModelica

For OMC generated FMUs it is currently assumed that any output depends on every input. Thus, no direct feed-back configuration as displayed in Figure 4 can be scheduled in a sequential calling sequence. Hence, every feed-back loop is either required to cross an explicit delay, or it must be assumed that an algebraic loop needs to be solved using an iterative calling sequence, *e.g.*, the algebraic loop is solved by a Newton iteration as described in [FMI14, p. 73].

The advantage of OMC's current approach is its simplicity. However, the restricted reusability is a disadvantage which motivates the investigation and development of more sophisticated approaches for handling variable dependencies and calling sequences.

3.3.4 Reusability Optimized Code-Generation

One considerable restriction in the straight-forward approach to code generation is the restricted reusability due to equation sorting (see Section 3.3.2).

The problem is related to the well-known problem of modular code-generation for synchronous data-flow languages. Lublinerman et al. provide a good discussion on implications of different modular code-generation approaches on *modularity*, *reusability*, and *code size* [LT08, LST09]. The following paragraphs will define and explain this terms and illustrate the consequences using the FMU_1 example.

Modularity [LST09] defines *modularity* in terms of generated interface functions for a *block* with inputs and outputs similar to the notation used in Figure 3. Their interface functions have a signature that maps block inputs to function arguments and block outputs to return values, *e.g.*, if single (*monolithic*) interface function would be generated for FMU_1, its signature in their notation would read

$$\text{FMU}_1.\text{step}(u_1, u_2, u_3) \quad \text{returns } (y_1, y_2) \tag{9}$$

The smaller the number of interface functions, the higher the degree of modularity[14].

Reusability [LST09] define *reusability* as the ability to embed generated code in any context. In terms of the FMU_1 example and associated terminology, this corresponds to maintaining the theoretical composition flexibility as described by dependencies (2) as much as possible after sequential code-generation. In this context, *maximal reusability* denotes that the composition flexibility for the generated sequential code is equal to the theoretical composition flexibility as described by (2).

Code size As will be illustrated later, increasing reusability may increase the resulting code size. Hence, there is a potential trade-off between reusability and resulting code size which needs to be considered, particularly, if code shall be generated for a more restricted embedded system.

The introduced definition for reusability translates straight-forwardly into the FMI world. However, the modularity definition from above is not as directly transferable, since the FMI does not have function signatures similar to (9).

[14]Lublinerman et al. [LST09] justify this definition by complexity considerations. They note that the number of interface functions (within their framework) is related to the complexity of algorithms such as cycle detection or clustering (used by their method). Hence, they argue that minimizing the number of interface functions is essential for scalability.

Causality graph oriented code-generation As mentioned when introducing the FMU$_1$ example in Section 3.3.1, the example is constructed in a way that it can be straight-forwardly be interpreted as a causality graph. The construction of a causality graph from acausal equations is part of the standard code-generation process in OpenModelica (as well as in other Modelica tools), thus such a structure is internally available in OMC. In the following we assume that this causality graph is available.

To enable a reusability optimized code-generation it suggests itself to utilize the causality graph structure more directly instead of committing to a particular topological sorting during the code generation (the “straight-forward” approach described in Section 3.3.2). Listing 2 adopts that strategy by providing an internal function for every node in the (acyclic) causality graph (lines 62, 70, 79). The cache invalidating strategy in the `FMU1_fmi2SetReal(..)` function is conservative. It could be optimized by a more sophisticated implementation which invalidates only those cached variables that depend on the changed input. However, typical control algorithms read each input only once during one sample period so that the simple approach of Listing 2 should be sufficient for such use-cases. When the value of a variable is requested (indiscriminate whether declared as output or local variable) a **`switch`** statement dispatches to the respective internal node function. Every node function knows its causality dependencies and calls the respective nodes. Caching ensures that nodes are not evaluated several times. The pseudo code omits error handling, but hints it at some places (lines 51, 55, 59).

The approach allows maximal reusability of the generated code. Its disadvantage is the abundance of generated functions (one for each node) which increase the code size and also impair the readability of the generated code in case of larger systems (*e.g.*, imagine that $n \gg 1$ internal w_i variables).

Listing 2: Causality graph oriented pseudo C-code with maximal reusability.

```
1  #define FMU1_u1 0
2  #define FMU1_u2 1
3  #define FMU1_u3 2
4  #define FMU1_w1 3
5  #define FMU1_y1 4
6  #define FMU1_y2 5
7  #define FMU1_nvars 6
8  #define FMU1_ninputs 3
9
10
11 typedef enum { fmi2OK, fmi2Warning, fmi2Discard, fmi2Error,
12   fmi2Fatal, fmi2Pending } fmi2Status;
13
14 typedef struct {
15     double time;
16     double v[FMU1_nvars]; /* Array of variable values */
17     /* Boolean condition array denoting if (cached) value of v[
          index] is available */
18     int c[FMU1_nvars];
19 } FMU1;
20
21 fmi2Status FMU1_fmi2SetTime(FMU1* m, double time) {
22   if (abs(time - m->time) > eps) {
```

```c
    /* check whether time != m->time */
    m->time = time;
    /*  re-initialize caching of variables */
    for (int i=0; i<FMU1_nvars; i++ ) {
      m->c[i] = 0;
    }
  }
  return fmi2OK; /* omitting error checking */
}

fmi2Status FMU1_fmi2SetReal(FMU1* m, const unsigned int vr[],
                            int nvr, const double value[]) {
  for (int i=0; i<nvr; i++) {
    int vi = vr[i];
    if (m->c[vi] == 1) {
      /* re-setting of an already set input -> conservatively
         invalidate all cached non-input variables */
      for (int j=FMU1_ninputs; j<FMU1_nvars; j++ ) {
        m->c[j] = 0;
      }
    }
    m->v[vi] = value[i];
    m->c[vi] = 1;
  }
  return fmi2OK; /* omitting error checking */
}

void intern_u1(FMU1* m) {
  if (m->c[FMU1_u1] == 0)
    error("Input u1 required at instant when it was not set");
}
void intern_u2(FMU1* m) {
  if (m->c[FMU1_u2] == 0)
    error("Input u2 required at instant when it was not set");
}
void intern_u3(FMU1* m) {
  if (m->c[FMU1_u3] == 0)
    error("Input u3 required at instant when it was not set");
}

void intern_w1(FMU1* m) {
  if (m->c[FMU1_w1] == 0) {
    intern_u2(m);
    m->v[FMU1_w1] = m->v[FMU1_u2];
    m->c[FMU1_w1] = 1;
  }
}

void intern_y1(FMU1* m) {
  if (m->c[FMU1_y1] == 0) {
    intern_u1(m);
```

```
 73      intern_w1(m);
 74      m->v[FMU1_y1] = m->v[FMU1_w1] + m->v[FMU1_u1];
 75      m->c[FMU1_y1] = 1;
 76    }
 77  }
 78
 79  void intern_y2(FMU1* m) {
 80    if (m->c[FMU1_y2] == 0) {
 81      intern_u3(m);
 82      intern_w1(m);
 83      m->v[FMU1_y2] = m->v[FMU1_w1] + m->v[FMU1_u2];
 84      m->c[FMU1_y2] = 1;
 85    }
 86  }
 87
 88  fmi2Status FMU1_fmi2GetReal(FMU1* m, const unsigned int vr[],
 89                              int nvr, double value[]) {
 90    for (int i=0; i < nvr; i++) {
 91      int vi = vr[i];
 92      switch (vi) {
 93        case FMU1_w1:
 94          intern_w1(m);
 95          break;
 96        case FMU1_y1:
 97          intern_y1(m);
 98          break;
 99        case FMU1_y2:
100          intern_y2(m);
101          break;
102      }
103      value[i] = m->v[vi];
104
105    return fmi2OK; /* omitting error checking */
106  }
```

Optimal disjoint clustering Lublinerman et al. [LST09] propose a method that allows clustering of nodes in a way that optimizes modularity while maintaining maximal reusability. The clusters produced by applying their method on the FMU_1 example is depicted in Figure 5. For the code-generation they would synthesize one function for each of the determined cluster (hence, a total of three interface function for the running example). As explained previously their interface functions have no direct mapping into the FMI world. However, we can consider a mapping to *internal* FMU functions to which we dispatch from the `FMU1_fmi2GetXXX` functions.

In the pseudo code in Listing 3 each disjoint cluster is mapped to a internal functions (lines 56, 67, 77). The example code considers three local variables (w_1, w_2, w_3) whose nodes are combined in the "cluster function" `intern_w1w2wn(..)`. A **`switch`** statement (line 91) dispatches to the respective "cluster function" in which the requested value is computed. Again, the pseudo code omits error handling, but insinuates it at some places (lines 58, 69, 79).

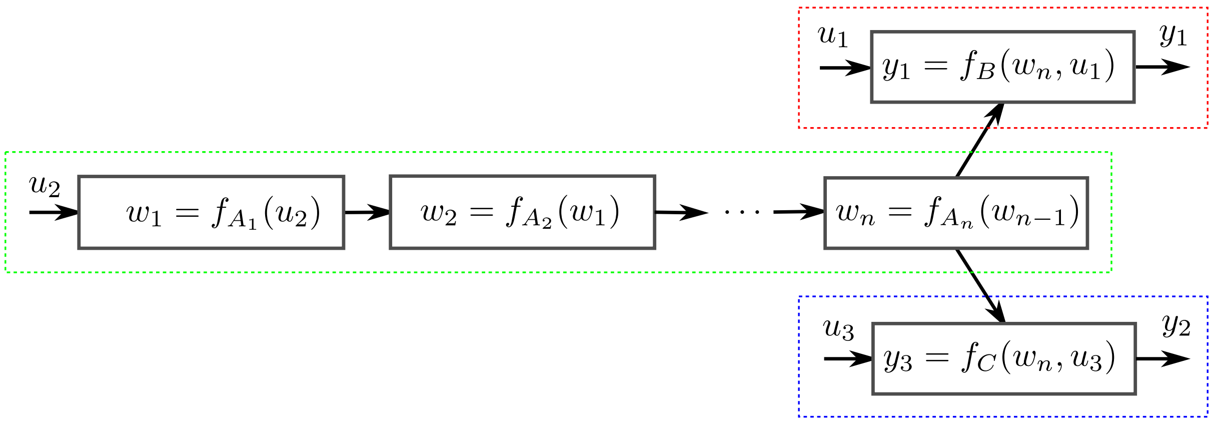


Figure 5: Clustering produced by optimal disjoint clustering [LST09].

Similarly to previous example (Listing 2), the optimal disjoint clustering approach allows maximal reusability of the generated code. In addition it reduces the amount of generated functions, which reduces the code size and should allow for better readability of the generated code in case of larger systems. Its disadvantage is the computation complexity for the clustering problem. Lublinerman et al. [LST09] show that their optimal disjoint clustering approach is intractable in the sense that it belongs to the family of NP-complete problems. However, they also report that their experimental results on applying their method to real-world models gave encouraging results (only a few seconds computation time) which indicated that the computation time may be acceptable in practice. A similar observation is reported by Pouzet and Raymond [PR10] who propose another computation approach for the same problem and validated it on several industrial examples.

Listing 3: Optimal disjoint clustering pseudo C-code with maximal reusability.

```
#define FMU1_u1 0
#define FMU1_u2 1
#define FMU1_u3 2
#define FMU1_w1 3
#define FMU1_w2 4
#define FMU1_w3 5
#define FMU1_y1 6
#define FMU1_y2 7
#define FMU1_nvars 8
#define FMU1_ninputs 3
#define FMU1_nclusters 3


typedef enum { fmi2OK, fmi2Warning, fmi2Discard, fmi2Error,
  fmi2Fatal, fmi2Pending } fmi2Status;

typedef struct {
    double time;
    double v[FMU1_nvars]; /* Array of variable values */
    /* Boolean condition array denoting if (cached) value(s) are
       available */
    int c_inp[FMU1_ninputs];
    int c_clusters[FMU1_nclusters];
} FMU1;

fmi2Status FMU1_fmi2SetTime(FMU1* m, double time) {
  if (abs(time - m->time) > eps) {
    /* check whether time != m->time */
    m->time = time;
    /*  re-initialize caching of variables */
    for (int i=0; i<FMU1_ninputs; i++ ) {
      m->c_inp[i] = 0;
    }
    for (int i=0; i<FMU1_nclusters; i++ ) {
      m->c_clusters[i] = 0;
    }
  }
  return fmi2OK; /* omitting error checking */
}

fmi2Status FMU1_fmi2SetReal(FMU1* m, const unsigned int vr[],
                            int nvr, const double value[]) {
  for (int i=0; i<nvr; i++) {
    int vi = vr[i];
    if (m->c[vi] == 1) {
      /* re-setting of an already set input -> conservatively
         invalidate all cluster caching indicators */
      for (int j=0; j<FMU1_nclusters; j++ ) {
        m->c_clusters[j] = 0;
      }
    }
    m->v[vi] = value[i];
    m->c_inp[vi] = 1;
  }
  return fmi2OK; /* omitting error checking */
}

void intern_w1w2wn(FMU1* m) {
  if (m->c[FMU1_w1w2w3] == 0) {
    if (m->c[FMU1_u2] == 0)
      error("Input u2 required at instant when it was not set");
    m->v[FMU1_w1] = m->v[FMU1_u2];
    m->v[FMU1_w2] = m->v[FMU1_w1];
    m->v[FMU1_w3] = m->v[FMU1_w2];
    m->c[FMU1_w1w2w3] = 1;
  }
}

void intern_y1(FMU1* m) {
  if (m->c[FMU1_y1] == 0) {
    if (m->c[FMU1_u1] == 0)
      error("Input u1 required at instant when it was not set");
    intern_w1w2w3(m);
    m->v[FMU1_y1] = m->v[FMU1_w3] + m->v[FMU1_u1];
    m->c[FMU1_y1] = 1;
  }
}

void intern_y2(FMU1* m) {
  if (m->c[FMU1_y2] == 0) {
    if (m->c[FMU1_u3] == 0)
      error("Input u3 required at instant when it was not set");
    intern_w1w2w3(m);
    m->v[FMU1_y2] = m->v[FMU1_w3] + m->v[FMU1_u2];
    m->c[FMU1_y2] = 1;
  }
}

fmi2Status FMU1_fmi2GetReal(FMU1* m, const unsigned int vr[],
                            int nvr, double value[]) {
  for (int i=0; i < nvr; i++) {
    int vi = vr[i];
    switch (vi) {
      case FMU1_w1:
      case FMU1_w2:
      case FMU1_w3:
        intern_w1w2w3(m);
        break;
      case FMU1_y1:
        intern_y1(m);
        break;
      case FMU1_y2:
        intern_y2(m);
        break;
    }
    value[i] = m->v[vi];

  return fmi2OK; /* omitting error checking */
}
```

4 Conclusion

This report discussed a concept for code generation from the Modelica language, particularly aiming for allowing a flexible adaption of the generated code to different target systems. The interface of the generated code shall be FMI compatible (FMI v2.0 as implementation base, but open for experimenting with extensions or version updates if needed during the project) since it allows importing the generated code into already available tools for easy co-simulation with other systems and for testing purposes. Furthermore, the FMI format seems quite suitable for integration into existing software architecture since it provides a clean interface for which required wrapper functions can be generated in an automated way.

FMI already supports FMUs which are shipped with C source code (instead, or in addition, of Dynamic Link Librarys (DLLs) or shared objects). In order to be adaptable to different target systems (*e.g.*, by cross-compilation) it is planned to generate exactly these source-code FMUs. However, embedded target systems often have restrictions which require generating different code than for desktop machines. Such restrictions have been identified and fixed in the report so that they are kept in mind for the prototype development in task T3.2 and T3.3.

An interesting sub-problem for FMI compatible code is related to (input/output) variable dependencies and calling sequences for maximizing the reusability of an FMU in different contexts. This problem has been discussed in more detail, since it has interesting impacts on modularity, reusability, and code size for which different trade-offs can be explored. Hence, several code-generation approaches were presented (partly building on reported research results in the area of modular code generation for synchronous block diagrams) and their trade-offs were discussed. Due to different trade-offs, it is not clear which code-generation method suits best and the follow-up prototype development might need to support several alternatives (task T3.2 and T3.3). Furthermore, it is expected that the standard FMU generation (*i.e.*, export of binary FMUs for desktop simulation), will eventually benefit from the prototype development within this work package (transfer of improvements concerning code efficiency and reusability of FMUs).

To the best of the authors' knowledge current Modelica tools have no designated support for exporting FMUs which are tailored for restricted embedded systems targets. Compared to the SotA, the code-generation concept outlined in this report aims to provide designated support for producing FMUs which are *optimized for restricted embedded systems* while also providing entry points for a user to *customize the code-generation* for different target systems and use-cases.

References

[AB06] J. Andreasson and T. Bünte. Global chassis control based on inverse vehicle dynamics models. *Vehicle System Dynamics*, 44:321–328, 2006. doi:10.1080/00423110600871459.

[Bea16] François Beaude. Benchmark network models. Technical Note D5.3, OPENCPS project, ITEA3, Project 14018, December 2016.

[BNA+15] Christian Bertsch, Jonathan Neudorfer, Elmar Ahle, Siva Sankar Arumugham, Karthikeyan Ramachandran, and Andreas Thuy. FMI for physical models on automotive embedded targets. In Peter Fritzson and Hilding Elmqvist, editors, 11th *Int. Modelica Conference*, Versailles, France, September 2015. doi:10.3384/ecp1511843.

[CBB+10] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2010.

[EOC95] Hilding Elmqvist, Martin Otter, and Françoise E. Cellier. Inline Integration: A New Mixed Symbolic/Numeric Approach for Solving Differential-Algebraic Equation Systems. In *European Simulation Multiconference*, 1995.

[FMI10] FMI development group. Functional Mock-up Interface for Model Exchange v1.0. Modelica Association Project "FMI", January 2010. Standard Specification. URL: https://www.fmi-standard.org/.

[FMI14] FMI development group. Functional Mock-up Interface for Model Exchange and Co-Simulation v2.0. Modelica Association Project "FMI", October 2014. Standard Specification. URL: https://www.fmi-standard.org/.

[FPSP09] Peter Fritzson, Pavol Privitzer, Martin Sjölund, and Adrian Pop. Towards a Text Generation Template Language for Modelica. In Francesco Casella, editor, 7th *Int. Modelica Conference*, Como, Italy, September 2009. doi:0.3384/ecp09430124.

[Fra15] Rüdiger Franke. Mathematical Optimization of Dynamic Systems with OpenModelica. OpenModelica Annual Workshop 2015, February 2015. Talk at workshop. URL: http://www.modprod.liu.se/openmodelica-2015/1.620216/OpenModelica2015-talk02-Franke_Optimization.pdf.

[Fri14] Peter Fritzson. *Principles of Object Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley IEEE Press, 2014.

[FWW+15] Rüdiger Franke, Marcus Walther, Niklas Worschech, Willi Braun, and Bernhard Bachmann. Model-based control with FMI and a C++ runtime for Modelica. In Peter Fritzson and Hilding Elmqvist, editors, 11th *Int. Modelica Conference*, Versailles, France, September 2015. doi:10.3384/ecp15118339.

[LST09] Roberto Lublinerman, Christian Szegedy, and Stavros Tripakis. Modular code generation from synchronous block diagrams: Modularity vs. code size. In *ACM SIGPLAN Notices*, volume 44, pages 78–89. ACM, 2009. doi:10.1145/1594834.1480893.

[LT08] Roberto Lublinerman and Stavros Tripakis. Modularity vs. Reusability: Code Generation from Synchronous Block Diagrams. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 1504–1509, New York, NY, USA, March 2008. ACM. doi:10.1145/1403375.1403736.

[MKWM15] Nils Menager, Rüdiger Kampfmann, Niklas Worschech, and Lars Mikelsons. Suitability of Different Real-Time Solvers for a Model-Based Engineering Toolchain using Industrial Rexroth Controllers. In Peter Fritzson and Hilding Elmqvist, editors, 11th *Int. Modelica Conference*, Versailles, France, September 2015. doi:10.3384/ecp15118883.

[PR10] Marc Pouzet and Pascal Raymond. Modular Static Scheduling of Synchronous Data-flow Networks: An efficient symbolic representation. *Journal of Design Automation for Embedded Systems*, 3(14):165–192, 2010. URL: http://www.springerlink.com/content/0929-5585/14/3/.

[TH11] Bernhard Thiele and Dan Henriksson. Using the Functional Mockup Interface as an Intermediate Format in AUTOSAR Software Component Development. In Christoph Clauß, editor, 8th *Int. Modelica Conference*, Dresden, Germany, March 2011. doi:10.3384/ecp11063484.

[TKOB05] M. Thümmel, M. Kurze, M. Otter, and J. Bals. Nonlinear inverse models for control. In 4th *Int. Modelica Conference*, pages 267–279, 2005.

[TS16] Bernhard Thiele and Per Sahlin. Translation validation and traceability concept from acausal hybrid models to generated code. Technical Note D3.2, OPENCPS project, ITEA3, Project 14018, December 2016.

[TSGT08] E. D. Tate, Michael Sasena, Jesse Gohl, and Micheal Tiller. Model embedded control: A method to rapidly synthesize controllers in a modeling environment. In 6th *Int. Modelica Conference*, pages 493–502, Bielefeld, Germany, March 2008.

[WM12] Niklas Worschech and Lars Mikelsons. A Toolchain for Real-Time Simulation using the OpenModelica Compiler. In Martin Otter and Dirk Zimmer, editors, 9th *Int. Modelica Conference*, pages 839–846, Munich, Germany, September 2012. doi:10.3384/ecp12076839.