Modeling, Identification and Control, Vol. 36, No. 1, 2015, pp. 23–52, ISSN 1890–1328

Towards Qualifiable Code Generation from a Clocked Synchronous Subset of Modelica

B. Thiele [1] A. Knoll [2] P. Fritzson [1]

[1] *PELAB, Linköping University, SE-581 83 Linköping, Sweden. E-mail:*{bernhard.thiele,peter.fritzson}@liu.se

[2] *Institute for Robotics and Embedded Systems, Technische Universität München, 85748 Garching bei München, Germany. E-mail:* knoll@in.tum.de

Abstract

So far no qualifiable automatic code generators (ACGs) are available for Modelica. Hence, digital control applications can be modeled and simulated in Modelica, but require tedious additional efforts (*e.g.*, manual reprogramming) to produce qualifiable target system production code. In order to more fully leverage the potential of a model-based development (MBD) process in Modelica, a qualifiable automatic code generator is needed.

Typical Modelica code generation is a fairly complex process which imposes a huge development burden to any efforts of tool qualification. This work aims at mapping a Modelica subset for digital control function development to a well-understood synchronous data-flow kernel language. This kernel language allows to resort to established compilation techniques for data-flow languages which are understood enough to be accepted by certification authorities.

The mapping is established by providing a translational semantics from the Modelica subset to the synchronous data-flow kernel language. However, this translation turned out to be more intricate than initially expected and has given rise to several interesting issues that require suitable design decisions regarding the mapping and the language subset.

Keywords: Modelica, Automatic Code Generation, Model-Based Development, Safety-Relevant Systems

1. Introduction

Deliberate use of modeling and simulation (M&S) is an effective strategy to cope with increasing product complexity and time to market pressure. A particular challenge is software functions running on distributed, networked computing devices which are in feedback loops with physical processes (cyber-physical systems). For the development and validation of these functions, it is no longer sufficient to solely consider a single, self-contained component without taking the interaction of this part within the whole system (including the physical parts!) into account.

For example, modern automotive vehicles can have more than 60 ECUs (electronic control units) communicating over a heterogeneous network of automotive buses like CAN, FlexRay, LIN, and MOST, as well as other communication buses used for infotainment purposes. The complexity[1] of the utilized embedded software increased about fiftyfold within 15 years (Ebert and Jones, 2009). Currently the development phase for new vehicles is estimated to be about three years (Schäuffele and Zurawka, 2010). Increased use of virtual prototyping methods has the potential to further curb the required development time span. Nowadays, model-based development (MBD) has become a well established development approach in the domain of em-

[1] Taking the size in object instruction as a measure of code complexity.

doi:10.4173/mic.2015.1.3

bedded (control) systems and the original promise of model-based development, to provide a more rapid and economic development process, seems to be confirmed in industrial practice (Broy et al., 2011).

In order to optimize the benefits gained by a model-based development process as, it is crucial that formally specified high-level applications can be automatically transformed (usually by using generated embedded C-code) into executable binary code for respective embedded platforms, thus eliminating error prone and expensive manual re-programming of the application using a general-purpose programming language. This requires automatic code generators (ACGs). However, if the generated code affects safety related functions the additional effort to safeguard the generated code diminishes the initial benefit of model-based development. A remedy is to rely on (automatic code generation) tools that are *qualifiable i.e.*, tools for which a certain degree of confidence in their correctness can be established for the relevant use cases. Specialized standards (despite conceptual similarities and shared base standards) apply for different industrial domains, *e.g.*, ISO 26262 (Automotive), ISO 13849: (Machinery Control Systems), DO-178 (Aircraft), *etc.*

There are tools on the market which fulfill (for specific industrial domains) the necessary requirements for safety related developments and that support control system modeling in an adequate manner (*e.g.*, Simulink Coder from MathWorks, ASCET from ETAS, SCADE from Esterel, TargetLink from dSpace). However, they share two drawbacks: (i) they do not offer state-of-the-art physical modeling[2], and (ii) they are based on proprietary model formats. Hence, more integrated modeling approaches (integrated modeling of software and physical aspects, facilitated integration in an existing software eco system) are considerably impeded.

Extensions of the non-proprietary, multi-domain cyber-physical system modeling language Modelica (Fritzson, 2014) aim at closing that gap by integrating language elements that are motivated from the point of view of clocked synchronous data-flow languages[3] for safety-relevant, sampled-data systems (Elmqvist et al., 2012; Otter et al., 2012).

However, so far no Modelica tool for high-integrity, embedded code generation has appeared on the market. Typical compilation techniques for Modelica differ significantly from established compilation techniques for data-flow languages. Hence, available knowledge about high-integrity code generation for clocked synchronous data-flow languages are not directly transferable to Modelica, exacerbating any qualification attempts.

This work proposes to link the semantics of a Modelica subset for digital control function modeling to an established clocked synchronous data-flow kernel language (SDFK) by means of a *translational semantics*[4]. This is achieved by providing transformation equations from the Modelica language to the SDFK language. At the same time this translational semantics is close to a practical implementation of a source-to-source compiler. This opens up a path to a qualifiable Modelica compiler by leveraging established compilation techniques for clocked synchronous data-flow languages which are understood enough to be accepted by certification authorities.

2. Model-Based Development and Tool Qualification

2.1. Overview

Figure 1 shows a typical model-based development toolchain including an automatic code generator. The *specification model* (aka *physical model*) is designed using a high-level domain-oriented modeling tool. These specification models are typically enriched with implementation details (*e.g.*, a continuous-time controller model is replaced by a discrete-time approximation) resulting in so-called *code generation* or *implementation models*. A code generator transforms code generation models into C-code, that the cross compiler translates into object code. The different object codes, including legacy and basic software code are then finally linked to

[2] MathWorks reacted to this deficiency by offering several Simulink toolboxes, most notably SimScape, to enable equation-based physical modeling (SimScape made its debut in the R2007A release of Matlab/Simulink). Unfortunately, despite using the same basic concepts like Modelica, MathWorks decided to create its own, proprietary, physical modeling language. In effect, the huge pool of high-quality (free and commercial) physical modeling libraries already available for Modelica cannot be reused in SimScape.

[3] Synchronous languages (Benveniste et al., 2003) are an established technology for modeling, specifying, validating, and implementing real-time embedded applications. Prominent members of the synchronous family include the Lustre (Caspi et al., 1987), Esterel (Boussinot and De Simone, 1991) and Signal (LeGuernic et al., 1991) language. The greatest industry relevancy can be attributed to the Lustre-based commercial Scade tool (Sauvage and Bouali, 2006) that is especially used for the development of safety critical software functions. Synchronous languages have no notion of continuous time and are therefore not suited for physical modeling.

[4] Attempts to formalize semantic aspects of the Modelica language can be traced back to the very early phases of the language standard where *natural semantics* based approaches were considered as a help in the language design process (Kågedal and Fritzson, 1998) and also as a base for translator generation (Fritzson et al., 2009). However, these previous studies focused rather on general advantages of formal language specifications and were not particularly concerned with high-integrity, embedded code generation.

a binary that can be executed by an embedded target.

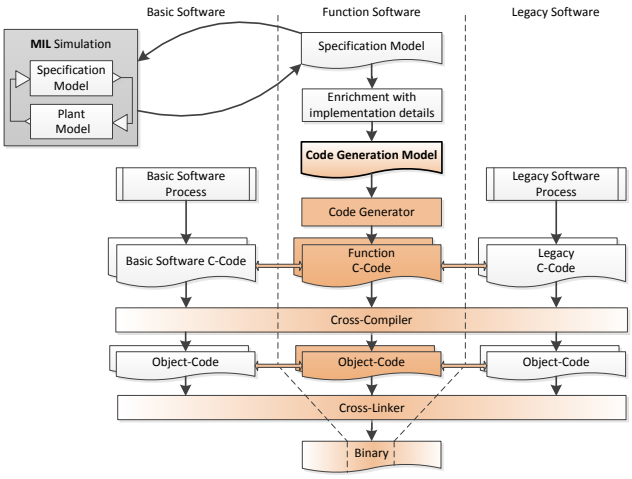


Figure 1: The generic build process for a model-based development toolchain with an automatic code generator (adapted from Schneider et al. (2009)).

The development of safety-relevant software typically needs to comply to rules described in functional safety standards. This also affects the used development tools. A development tool may need to fulfil a certain level of qualification depending on the severity of the impact a malfunction in the development tool is expected to have on a safety-related item. For example, in the automotive domain the ISO 26262-8:2011, Section *11 Qualification of software tools*, establishes a set of recommendations concerning the needed qualifications of such software tools.

The automatically generated C-code in Figure 1 can also be seen as a low-level intermediate representation of the model, because it is both the output of the code generator as well as input to the cross compiler for the target. If sufficient confidence can be put into the correct functioning of the tool chain, safeguarding measures for intermediate representations can be eliminated. For example, code reviews on automatically generated code can be omitted.

2.2. Tool Qualification

Functional safety standards typically classify development tools into categories that depend on the impact a systematic tool failure ("bug") has on a safety related item and on the probability that a bug remains undetected by downstream development steps and tools.

For example, ISO 26262 defines four categories termed *Tool Confidence Level* (TCL1 – TCL4) where TCL4 denotes the highest level of required confidence. Tools classified higher than TCL1 need to be *qualified*. Required qualification measures are further tied to the usage context, in particular to the *criticality* of the software function that is realized with the development tool. ISO 26262 further recommends four principle methods for tool qualification: (i) increased confidence from use, (ii) development process evaluation, (iii) software tool validation, and (iv) development in compliance with a safety standard. While (i) or (ii) are recommended for less critical functions and/or tool confidence levels, (iii) or (iv) are considered an adequate choice for highly critical functions and tool confidence levels.

Note that the output of tools that are not qualified can still be used, but require suitable safeguarding measures, e.g., automatically generated code can be passed into a standard assurance process where it is treated like manually written code. The benefit of qualified development tools is that certain safeguarding measures, like code reviews on generated code (Stürmer et al., 2006), can be omitted.

In order to increase the confidence in the toolchain depicted in Figure 1 it is, in principle, possible to apply a number of techniques that have been proposed for ensuring safety of compilers. The survey by Frank et al. (2008) gives an overview of existing techniques and also comments on the suitability of the presented methods for practical application. In particular Frank et al. conclude that a combination of test-based approaches (as opposed to formal verification-based approaches) is the most promising way to improve the reliability of compilers for safety-related applications within the automotive industry.

Despite plenty reports in the open literature about (in many cases formal) methods that are in principle capable of increasing confidence levels in compilers or code generators, detailed reports of actually successful tool qualification efforts for automatic code generators are rare. Schneider et al. (2009) give a rather comprehensive report on the practical qualification of an industrial-strength development tool (an integrated code generator tool with target compiler) by a *validation suite* approach (hence, by a software tool validation method) according to automotive requirements. The approach is centered around an automated test environment that allows the automated execution of large numbers of test cases. The number of required test cases is subject to scalability issues so that suitable *language restrictions* are an important prerequisite before starting qualification efforts for complex real-world languages.

Regardless of whether the tool qualification method of choice is to develop a new tool according to ap-

propriate safety standards, or to use a test-based or verification-based approach to qualify an existing tool: keeping the input language of the code generation model (see Figure 1) as *simple and well-defined as possible* is essential to keep development/qualification efforts under control. This regards the number and complexity of basic constructs in the language and also the number and complexity of performed transformation rules in the code generation process.

Furthermore, it is important to understand that technically tools need to be qualified in the context of a particular system development project and therefore only *qualifiable* tools are available on the market. Using a qualifiable tool alleviates tool qualification efforts and the project specific tool qualification is usually achieved in close collaboration of the tool user and the tool vendor (Hatcliff et al., 2014).

2.3. Typical Modelica Code Generation

Compiling Modelica code usually involves substantial code transformation. The following description is a slightly adapted reproduction of (Thiele et al., 2012, Section 4.2.5). Figure 2 gives an overview of the compilation and simulation process as described by Broman (2010, p. 29).

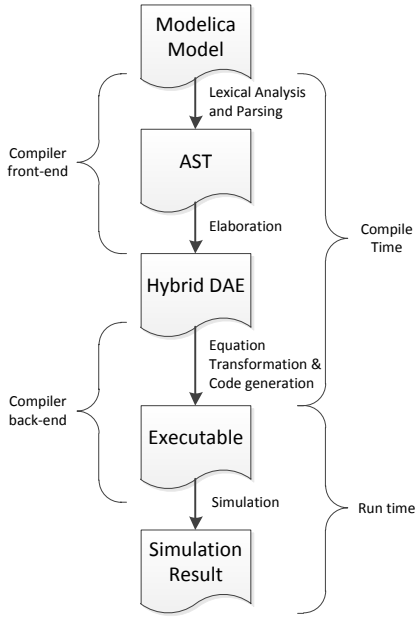


Figure 2: Outline of a typical compilation and simulation process for a Modelica language tool (Broman, 2010, p. 29).

The different phases are:

Lexical Analysis and Parsing This is standard compiler technology.

Elaboration Involves *type checking*, *collapsing the instance hierarchy* and *generation of connection equations* from connect-equations. The result is a hybrid differential algebraic equation (DAE) system consisting of variable declarations, equations from equations sections, algorithm sections, and `when`-clauses for triggering discrete-time behavior.

Equation Transformation This step encompasses transforming and manipulating the equation system into a representation that can be efficiently solved by a numerical solver. Depending on the intended solver the DAE is typically reduced to an index one problem (in case of a DAE solver) or to an ODE form (in case of numerical integration methods like Euler or Runge-Kutta).

Code generation For efficiency reasons tools typically allow (or require) translation of the residual function (for an DAE) or the right-hand side of an equation system (for an ODE) to C-code that is compiled and linked together with a numerical solver into an executable file.

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

In the context of code generation for safety relevant systems the typical processing of Modelica models has two problems:

1. In the **Elaboration phase** the instance hierarchy of the hierarchically composed model is collapsed and *flattened* into one (large) system of equations, which is subsequently translated into one (large) chunk of C-code, thus impeding modularization and traceability at the C-code level.

2. In the **Equation Transformation phase** the equations are extensively manipulated, optimized and transformed at the global model level. The algorithms used in this step are the core elements that differentiate the model compiler tools (*quality of implementation*). Although the basic algorithms are documented in the literature, the optimized algorithms and heuristics used in commercial implementations are vendor confidential proprietary information. The lack of transparency and simplicity exacerbates tool qualification efforts.

Therefore, the compilation process for *simulation* (Figure 2) may be significantly different compared to the *target code compilation* process depicted in Figure 1. Not only because different compilers are used, but also because the target code generator may (need

to) be an entirely distinct piece of software that may share only minimal to no amounts of code with the simulation code generator[5]. In particular the target code generator depicted in Figure 1 is only required to understand a Modelica subset that is sufficient for digital control function modeling.

2.4. An Approach Towards a Qualifiable Modelica Code Generator

The requirement to keep the input language of the code generation model as simple and well-defined as possible motivated Thiele et al. (2012) in a previous work to propose a subset of Modelica for safety-relevant control applications that would offer a reasonable compromise between language expressiveness and expected effort of tool qualification. The proposed language restrictions were substantiated by providing rationales. However, no evidence was given about the practical feasibility to develop a qualifiable automatic code generator for that language subset.

The Modelica subset used in this work is a modified version of the subset proposed in the former paper. In order to allow a clear and brief presentation of the translational semantics the former subset is further reduced to a kernel of representative elements. The resulting language kernel for automated code generation is denoted as *mACG-Modelica*.

The presented approach towards a qualifiable code generator relies on:

1. A target language which allows the proposition that a qualifiable code generator can be implemented for it. This language is in the following referred to as the *Synchronous Data-Flow Kernel* language (SDFK).

2. A representative language kernel for digital control function development in Modelica (mACG-Modelica).

3. A set of translation equations from mACG-Modelica to *SDFK*. The semantics of mACG-Modelica are defined in terms of this translation (*translational semantics*).

[5] The amount of code that can be shared depends on various aspects, particularly on the tool qualification approach. Tool qualification by *developing the tool in compliance with a safety standard*, needs respective evidence on the development process that is usually not available for existing code. ISO 26262 also lists *increased confidence from use* as a method that is particularly applicable for more moderate safety requirements and *tool validation* as method that is also applicable for more safety-critical software — these methods are more amenable to reuse existing code.

The existence of the translation provides: (i) a strong argument for the feasibility to develop a qualifiable code generator for the considered Modelica subset and (ii) a base that can be used to create a gateway from Modelica to a qualifiable code generator that is based on clocked synchronous data-flow language like Scade/KCG (similarly to what has been reported by Caspi et al. (2003) for a translator from Simulink to Scade).

3. Translation to a Synchronous Data-Flow Kernel Language

3.1. The Synchronous Data-Flow Kernel Language (SDFK)

Synchronous data-flow kernel languages have been used in various publications as suitable representation for languages in certain formal methods based research. Instead of reinventing syntax, this work utilizes the synchronous data-flow kernel language[6] described by Biernacki et al. (2008). The SDFK is close to Lustre/Scade so that it allows to use *well-understood and accepted compilation techniques* that have been developed for these languages.

Biernacki et al. (2008) formally describe modular code generation from that data-flow kernel into imperative code and note that "*The principles presented in this article are implemented in the RELUC compiler of SCADE/LUSTRE and experimented on industrial real-size examples*". Hence, using that particular SDFK language is attractive since it directly allows to reuse the described code generation techniques for ACG-Modelica as soon as a translational semantics is available.

In order to keep the following discussion more self-contained, the syntax and intuitive semantics described in (Biernacki et al., 2008, Section 2) are briefly reproduced in the current section:

A program is made of a list of global type ("*td*") and node ("*d*") declarations. In order to allow a clear

[6] Note that the semantics of the synchronous kernel language is given informally in (Biernacki et al., 2008). However, it can be traced back to a formal semantics if necessary. Biernacki et al. (2008) refer to (Colaço et al., 2005) for the formal semantics of the clock calculus. Actually, the extended language presented in (Colaço et al., 2005) is similar to the one used in (Biernacki et al., 2008). In (Colaço et al., 2005) this extended language is formally translated into a more "basic" data-flow kernel language by a source-to-source transformation. For this "basic" data-flow kernel language (Colaço et al., 2005, Section 3.1) refers to (Colaço and Pouzet, 2003) for a (formal) denotational Kahn semantics (except for the semantics of the modular reset operator "**every**" which is formally defined in (Hamon and Pouzet, 2000)).

Table 1: Expressions in SDFK

v	*Values* are either immediate values ("i"), *e.g.*, integer values, or they are constructors ("C") belonging to a finite enumerated type (*e.g.*, the Boolean type is defined as "**bool** = **False** + **True**").
x	*Variables.*
$(a_1, \ldots, a_n)$	*Tuples.*
v **fby** a	*Initialized delays.* The first argument "v" (the initial value) is expected to be an immediate value, the second argument "a" is the stream that is delayed.
$op(a_1, \ldots, a_n)$	*Point-wise applications.* To simplify the presentation, "$op(a_1, \ldots, a_n)$" is a placeholder for any point-wise application of an external function op (*e.g.*, $+$, *not*) to its arguments. To improve the readability of examples, the application of classical arithmetic operations will be written in infix form.
$f(a_1, \ldots, a_n)$ **every** a	*Node instantiations* with possible reset condition "a". At any instant at which Boolean stream "a" equals "**True**" the internal state of the node instantiation is reset. To simplify the notation the reset condition "**every** a" may be omitted which is equal to writing "**every False**" as reset condition.
a **when** $C(x)$	*Sampling operations.* Sample a stream "a" at every instant where "x" equals "C".
merge x $(C \to a_1) \ldots (C \to a_n)$	*Combination operations* are symmetric to the sampling operation: They combine complementary streams in order to produce a faster stream. "x" is a stream producing values from a finite enumerated type "$bt = C_1 + \ldots + C_n$". "$a_1, \ldots, a_n$" are complementary streams, *i.e.*, at an instant where "x" produces a value at most one stream of "$a_1, \ldots, a_n$" is producing a value. At every instant where "x" equals "C_i" the value of the corresponding stream "a_i" is returned.

and brief presentation, only abstract types and enumerated types are considered in the discussion. A global node declaration "d" has the form "**node** $f(p) = p$ **with var** p **in** D". Within this node declaration "p" denotes a list of variables while "D" denotes a list of parallel equations. In an equation "$pat = a$" the pattern "pat" is either a variable or a tuple of patterns "$(pat, \ldots, pat)$" and "a" denotes an annotated expression "e" with its clock "ct". Table 1 briefly describes various elements that can be part of an expression. The expressions can be extended by the conditional "**if/then/else**" which relates to the original kernel by the equivalence relation:

$$\begin{aligned} &\textbf{if } x \textbf{ then } e_2 \textbf{ else } e_3 = \textbf{merge } x \\ &\qquad (\textbf{True} \to e_2 \textbf{ when True}(x)) \\ &\qquad (\textbf{False} \to e_3 \textbf{ when False}(x)) \end{aligned} \tag{1}$$

Note that the clock annotations "ct" have no impact on the data-flow semantics of the language. Clocks do not have to be explicitly given in the SDFK language, although they are part of the language semantics. For example, instead of writing "$((v \textbf{ fby } x^{ck})^{ck} + y^{ck})^{ck}$" it suffices to write "$((v \textbf{ fby } x) + y)$". Clock annotations in the SDFK language are determined automatically by a clock calculus which is defined as a type inference system. This clock calculus precedes the code generation step (see (Biernacki et al., 2008, Section 2.2) for more details).

The syntax of the (clock-annotated) SDFK language is defined by the following grammar:

$$\begin{aligned} td &::= \textbf{type } bt \mid \textbf{type } bt = C + \ldots + C \\ d &::= \textbf{node } f(p) = p \textbf{ with var } p \textbf{ in } D \\ p &::= x : bt; \ldots; x : bt \\ D &::= pat = a \mid D \textbf{ and } D \\ pat &::= x \mid (pat, \ldots, pat) \\ a &::= e^{ck} \\ e &::= v \mid x \mid (a, \ldots, a) \mid v \textbf{ fby } a \mid op(a, \ldots, a) \\ &\quad \mid f(a, \ldots, a) \textbf{ every } a \mid a \textbf{ when } C(x) \\ &\quad \mid \textbf{merge } x \ (C \to a) \ldots (C \to a) \\ v &::= C \mid i \\ ct &::= ck \mid ct \times \ldots \times ct \\ ck &::= \textbf{base} \mid ck \textbf{ on } C(x) \end{aligned}$$

For the translational semantics the expressions "e" are extended by the conditional "**if/then/else**" as defined in (1).

Some of the traditional operations supported by Lus-

tre are related to SDFK through the following equivalences:

Lustre		SDFK
e **when** x	=	e **when True**(x)
Lustre's *sampling operation*, where x is a Boolean stream.		
e_1 − > e_2	=	**if True fby False then** e_1 **else** e_2
Lustre's *initialization operator*.		
pre(e)	=	*nil* **fby** e
Lustre's *uninitialized delay operator*. The shortcut *nil* stands for any constant value e which has the type of e. It is the task of the initialization analysis to check that no computation result depends on the actual *nil* value.		

In order to illustrate the effect of these operators Table 2 shows example applications of these operators to streams of values.

Table 2: Examples for applying the SDFK operators

Stream/Expression	Stream values				
h	**True**	**False**	**True**	**False**	...
x	x_0	x_1	x_2	x_3	...
y	y_0	y_1	y_2	y_3	...
v **fby** x	v	x_0	x_1	x_2	...
$x + y$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	...
x − > y	x_0	y_1	y_2	y_3	...
pre(x)	*nil*	x_0	x_1	x_2	...
$z = x$ **when True**(h)	x_0		x_2		...
$t = y$ **when False**(h)		y_1		y_3	...
merge h (**True** → z) (**False** → t)	x_0	y_1	x_2	y_3	...

3.2. mACG-Modelica

To allow a clear and brief presentation of the translational semantics the Modelica language is reduced to a small subset of elements that is considered to be representative for data-flow based digital control functions. The resulting language kernel is denoted as mACG-Modelica.

A program is made of a list of global type ("td"), connector ("cd") and block ("bd") declarations. Only abstract types "t" are considered (nevertheless the provided examples will use concrete Modelica types, *e.g.*, replacing "t" by "`Real`"). Connectors "cd" have either input or output causality. A block declaration "d" has the form "**block** id p **equation** D **end**;", where "id" is the name of the block, "p" contains the local component declarations and "D" the equation declarations. A component declaration "p" can be modified by a modification "mo" (compared to Modelica modifications are more restricted, see Section 5).

Parameters can be declared with modification expression "**parameter** t x = me;", or without modification binding "**parameter** t x;"[7].

The use of component dot accesses for parameters is not supported in the presented translation in order to simplify the presentation. Allowing it would require to additionally introduce component dot access normalization (Figure 21) and "dummy" equation generation (Figure 22) in a slightly adapted form for parameters in the normalization step. For the actual translation step it would be necessary to translate all parameters not only to node input arguments, but also to node outputs arguments. This seems to make the translation harder to understand without adding much additional conceptual value[8].

Equations "D" are either connect equations "**connect**(cx, cx)" or equations of the form "cx = e", where "cx" is a single variable (the unknown of the equation, which is either accessed by a simple identifier "x", or by using a component dot access "$x.x$") and "e" is an expression (hence, equations are more restricted than in Modelica where the unknown of an equation may appear at an arbitrary place). Similar to SDFK an abstract n-ary operator "$op(e, \ldots, e)$" is provided to simplify the presentation (nevertheless the provided examples will be presented using concrete Modelica operators).

The syntax of mACG-Modelica is defined by the following grammar:

$$
\begin{aligned}
td &::= \textbf{type}\ t; \\
bd &::= \textbf{block}\ id\ p\ \textbf{equation}\ D\ \textbf{end}; \\
cd &::= \textbf{connector}\ id\ =\ c\ t; \\
c &::= \textbf{input} \mid \textbf{output} \\
p &::= p\ p \mid t\ x; \mid t\ x\ mo; \mid c\ t\ x; \mid c\ t\ x\ mo; \\
&\quad \mid \textbf{parameter}\ t\ x; \\
&\quad \mid \textbf{parameter}\ t\ x\ =\ me; \\
mo &::= (ar\ , \ldots ,\ ar) \\
ar &::= id\ =\ me \\
D &::= D\ D \mid eq;
\end{aligned}
$$

[7] Modelica semantics require that modification bindings for parameters have *parametric* or *constant variability* (Modelica Association, 2012, Section 3.8). This needs to be ensured by a statical check *before* the translation (for conciseness the description of that check is omitted).

[8] In addition to transforming parameters to input arguments of a (C-) function, it becomes necessary to make them available as output arguments. E.g., consider "`block A parameter Real p1 = 0.1; end A;`" which is instantiated in "`block B A a(p1=0.2); parameter Real p2 = a.p1; end B;`" and note that `a.p1` needs to return the value `0.2`. If `A` is translated to a function, there needs to be an input argument to set `p1` *and* an output argument to retrieve its value.

$$
\begin{aligned}
e &::= v \mid cx \mid op(e,\ldots,e) \mid \textbf{previous}(x) \\
&\quad \mid \textbf{if } e \textbf{ then } e \textbf{ else } e \\
me &::= v \mid x \mid op(me,\ldots,me) \\
&\quad \mid \textbf{if } me \textbf{ then } me \textbf{ else } me \\
eq &::= cx \;=\; e \mid \textbf{connect}(cx, cx) \\
cx &::= x \;.\; x \mid x \\
x &::= id \\
v &= \text{value} \\
id &= \text{identifier}
\end{aligned}
$$

Abstract types "t" encompass predefined primitive types and user-defined structured types (blocks and connectors). The set "B" is introduced to denote the set of all predefined primitive types, particularly Modelica's "**`Boolean`**", "**`Integer`**", and "**`Real`**" types.

From all the clocked synchronous language elements that are listed in (Modelica Association, 2012, Chapter 16) only "**previous**" appears in the grammar above. The clock conversion operators are omitted — for practical applications this is not as restrictive as it may appear at a first glance. Section 5 briefly comments on that.

Also the operator "**interval**(u)" is missing in the grammar. This operator is considered to be available as external function call or macro — the time span between a previous and a present tick is typically only known by the environment that triggers the execution of the synchronous data-flow program. Consequently, that value needs to be provided by the runtime environment. In the case of single-rate programs (*i.e.*, if no clock conversion operators are supported) the interval duration is simply the duration between two ticks of the base clock, in case of multi-rate programs it becomes more complicated and the value depends on the specific clock that is associated to the operators argument "u".

3.3. A Multilevel Translation Approach

The translation to the SDFK language is rather complex. In order to keep the translation manageable and understandable the translation is subdivided into several steps. The two major steps are:

1. *Normalization* of mACG-Modelica (formulated as source-to-source transformation). This step is again subdivided in:
   a) Generation of connection equations.
   b) Stripping of parameter modifications, normalizing instance modifications and extracting instance dot accesses appearing in expressions.
   c) Creating a fresh block that instantiates the top-level block as a component with normalized component modifications.
2. *Translation* to the SDFK language.

Normalization and translation are defined as a system of mutually recursive functions. The normalization is needed in order to transform mACG-Modelica into a (simplified) *normalized* form which is the basis for the translation to the SDFK language. The syntax for the *normalized* mACG-Modelica language can be found in Section 3.5.

Using a multilevel translation approach facilitates including further language elements, as long as a source-to-source transformation into a smaller language kernel can be given. The generation of connection equations is a good example for this: it eliminates the connector declarations and connect equations by replacing them with simple variable declarations (using the proper input/output causalities) and simple equations of the form "$x = e$". Hence, the multilevel translation approach provides a path to incrementally extend the mACG-Modelica subset to more comprehensive subsets for control function development (see also the discussion in Section 5).

3.4. The Normalization

The normalization is presented by using example code snippets that illustrate the required source-to-source transformation. The formal translation equations are rather heavy and are therefore provided as a supplementary part of the appendix, Section B.

The line numbers of the code snippets are consecutively incremented, so that descriptive text can refer to them. Once a class/type is declared, it may reappear in subsequent code snippets.

3.4.1. Generation of Connection Equations

Connector Declarations Connector declarations (line 1–2) are replaced by their corresponding short class definition (*e.g.*, `In` ↦ **`input Real`**, compare line 4 ↦ 10 and `Out` ↦ **`output Real`**, compare line 5 ↦ 11).

```
1 connector In  = input Real;
2 connector Out = output Real;
3 block A
4   In u;
5   Out y;
6 equation
7   y = u;
8 end A;
```

$\overset{source}{\downarrow}_{normal}$

```
9  block A
10   input Real u;
11   output Real y;
12 equation
13   y = u;
14 end A;
```

Connect Equations The connect equations are replaced by simple equations of the form "$x = e$". Note that the causality is not directly encoded in the connect equations, so it has to be inferred from the variable declarations.

```
15 block B
16   In u;
17   A a1;
18 equation
19   connect(u, a1.u);
20 end B;
```

$\overset{source}{\downarrow}_{normal}$

```
21 block B
22   input Real u;
23   A a1;
24 equation
25   a1.u = u;
26 end B;
```

3.4.2. Parameters, Instance Modifications, and Dot Accesses

Parameter Stripping Parameter modifications in a class are stripped away.

```
27 block PI
28   parameter Real kd = Td*2;
29   parameter Real Td = 0.1;
30 end PI;
```

$\overset{source}{\downarrow}_{normal}$

```
31 block PI
32   parameter Real kd;
33   parameter Real Td;
34 end PI;
```

Parameter Modifications at Instantiated Blocks All parameters of an instantiated block are extracted, merged with applicable instance modifications and introduced as fresh parameters with a modification expression.

Note that the parameter modification from line 28 is extracted and reintroduced in line 43, while the modification from line 29 is overridden by the instance modification in line 28 before being assigned to the fresh parameter `_pi_Td` in line 44.

```
35 block C
36   parameter Real k;
37   parameter Real Td=0.2;
38   PI pi(Td=Td);
39 end C;
```

$\overset{source}{\downarrow}_{normal}$

```
40 block C
41   parameter Real k;
42   parameter Real Td;
43   parameter Real _pi_kd=_pi_Td*2;
44   parameter Real _pi_Td=Td;
45   PI pi(kd=_pi_kd, Td=_pi_Td);
46 end C;
```

Extraction of dot accesses Instance dot accesses in RHS equations are extracted and replaced by fresh substitute variables.

```
47 block D
48   output Real y;
49   A a;
50 equation
51   a.u = 3;
52   y = a.y + 2;
53 end D;
```

$\overset{source}{\downarrow}_{normal}$

```
54 block D
55   output Real y;
56   A a;
57   Real _a_y;
58 equation
59   a.u = 3;
60   _a_y = a.y;
61   y = _a_y + 2;
62 end D;
```

3.4.3. Generation of Top-Level Instantiation Blocks

If a block is the top-level block for code generation, it needs a special treatment: parameter modifications in that block should not be lost by stripping them away. To achieve that without requiring a special case treatment in the preceding translation step, a fresh block that instantiates the top-level block with normalized instance modifications is inserted for every block. Hence, block `C` (line 35–39) would introduce the fresh block:

```
63 block _Inst_C
64   parameter Real _c_k;
65   parameter Real _c_Td=0.2;
66   C _c(k=_c_k, Td=_c_Td);
67 end _Inst_C;
```

Inputs and outputs of a block are simply propagated through, *e.g.*, block `D` (line 47–53) would introduce the fresh block:

```
68 block _Inst_D
69   output Real y;
70   D _d;
71 equation
72   y = _d.y;
73 end _Inst_D;
```

3.5. Normalized mACG-Modelica

After the normalization all component dot accesses are extracted from nested expressions. All "**connect**-equations" are resolved. At instance declarations all available parameters are set as modifications. In a block that instantiates another block, any output of the instantiated block is at least accessed once. The syntax of *normalized* mACG-Modelica is defined by the following grammar:

$$
\begin{aligned}
td &::= \textbf{type}\ t; \\
d &::= \textbf{block}\ id\ p\ \textbf{equation}\ D\ \textbf{end}\ id; \\
c &::= \textbf{input} \mid \textbf{output} \\
p &::= p\ p \mid t\ x; \mid t\ x\ mo; \mid c\ t\ x; \mid c\ t\ x\ mo; \\
&\quad \mid \textbf{parameter}\ t\ x \mid \textbf{parameter}\ t\ x\ =\ e; \\
mo &::= (ar\ , \ldots ,\ ar) \\
ar &::= id\ =\ e \\
D &::= D\ D \mid eq; \\
e &::= v \mid x \mid op(e, \ldots, e) \mid \textbf{previous}(x) \\
&\quad \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \\
eq &::= x\ =\ e \mid x\ .\ x\ =\ e \mid x\ =\ x\ .\ x \\
x &::= id \\
v &= \text{value} \\
id &= \text{identifier}
\end{aligned}
$$

3.6. The Translation

After normalization (Section 3.4), the model is available in the *normalized* mACG-Modelica language (Section 3.5). This form allows a more straightforward translation to the SDFK language (Section 3.1) which will be described in the following sections.

Any *normalized* mACG-block is directly mapped to an SDFK node — no context information from surrounding blocks is needed.

3.6.1. Intuitive Translation

Inputs, Outputs, Parameters without Modification, and Initialized Delays Inputs and parameters without modification bindings are mapped to node input arguments. Outputs are mapped to node return values. A lexicographic order relation on node input and output arguments ensures an unambiguous mapping.

Delays and the start values of their arguments are mapped to initialized delays (*e.g.*, "`previous(x)`" ↦ "`0 fby x`", if "`x(start=0)`", compare line 6,8 ↦ 16).

```
1  block PI
2    input Real u;
3    output Real y;
4    parameter Real kd;
5    parameter Real Td;
6    Real x(start=0);
7  equation
8    x = previous(x) + u/Td;
9    y = kd*(x + u)
10 end PI;
```

$\xrightarrow{normal}_{sdfk}$

```
11 node PI (Td:real,
12          kd:real,
13          u:real) =
14 y:real with
15 var x:real in
16     x = 0 fby x + u/Td
17 and y = kd*(x+u)
```

Parameters with Modification Parameters with modification bindings are mapped to local variables and equations.

```
18 block E
19   parameter Real k1 = 2*k2;
20   parameter Real k2 = 4;
21 end E;
```

$\xrightarrow{normal}_{sdfk}$

```
22 node PI () =  with
23 var k1:real,
24     k2:real in
25     k1 = 2*k2
26 and k2 = 4
```

Instance Modifications and Dot Access Instance modifications and instance dot accesses are mapped to function application like node instance calls.

```
27 block F
28   input u;
29   output y;
30   parameter Real _pi_Td = 0.1;
31   parameter Real _pi_kd = 2;
32   PI pi(kd=_pi_kd, Td=_pi_Td);
33   Real _pi_y;
34 equation
35   pi.u = 0.1*u;
```

```
36    _pi_y = pi.y;
37    y = _pi_y + 2;
38  end F;
```

$\overset{normal}{\downarrow_{sdfk}}$

```
39  node F (u:Real) =
40  y:real with
41  var _pi_Td:real,
42      _pi_kd:real,
43      _pi_y:real in
44      _pi_Td = 0.1
45  and _pi_kd = 2
46  and _pi_y = PI(_pi_Td, _Pi_kd, 0.1*u)
47  and y = _pi_y + 2
```

3.6.2. Formal Translation Semantics

This section defines the translational semantics for the *normalized* mACG-Modelica language. Hence, semantics of modeling constructs in *normalized* mACG-Modelica are expressed in terms of constructs from the SDFK language.

Notation The following notation shall be used:

- A sequence of elements $(e_1, \ldots, e_n)$ is frequently written as a *list* $[e_1;\ \ldots;\ e_n]$ for which an operator "+" is defined so that if $p_1 = [e_1;\ \ldots;\ e_n]$ and $p_2 = [e'_1;\ \ldots;\ e'_k]$ then $p_1 + p_2 = [e_1;\ \ldots;\ e_n;\ e'_1;\ \ldots; e'_k]$ provided $e_i \neq e'_j$ for all i, j such that $1 \leq i \leq n,\ 1 \leq j \leq k$. $[]$ denotes an empty list. Furthermore, $e_1 \in p_1$ shall denote that e_1 appears as an element in p_1.

- Frequently, elements e_i are tuples (*e.g.*, $e_i = (x_i, t_i)$). Especially if tuples encode variable names and their associated types an alternative notation is preferred in which "," is replaced by ":" and the parentheses are dropped (*e.g.*, $e_i = x_i : t_i$). The underscore "_" is used as a placeholder for entries which are irrelevant in the specific context.

- Mutually recursive functions "Function$_{(context)}(element)$" are used for defining a transformation of an element within a context. To keep the notation concise the definition of a function is often overloaded — its actual interpretation should be clear from the context.

- Additionally, the transformation relies on a suitable lexicographical order relation "$<_L$" to allow an unambiguous ordering of node input and output arguments (see Figure 13 on page 43).

Translation Equations The function $T(.)$ in Figure 3 defines the translation of a block from the *normalized* mACG-Modelica language to the SDFK language. Note that the translation is performed block-by-block, without requiring a context of global block declarations as it was required during normalization.

The function CEq creates an auxiliary representation structure d, i, o, l, s, j, q by traversing equations D and using an accumulator initialized by the function CId. The function CId is similar to CEq, but it traverses the block's instance declarations P and starts with all elements of the accumulator being set to the empty list $[]$. The auxiliary structure is then directly used in the translation T or further processed by translation functions TEqList and TCList and their supporting functions TE and TC (all described later).

$d = [x_1 : t_1;\ \ldots; x_n : t_n]$ stands for a list of declarations within the scope, $i = [x_1 : t_1;\ \ldots; x_n : t_n]$ stands for a list of input argument declarations, $o = [x_1 : t_1;\ \ldots; x_n : t_n]$ for a list of node output argument declarations, and $n = [x_1 : t_1;\ \ldots; x_n : t_n]$ for a list of local declarations (note that $d = i + o + l$). $s = [x_1 : v_1;\ \ldots; x_n : v_n]$ stands for an environment of variables with start values (*e.g.*, state variables).

$j = [(c_1 : t_1, ci_1, co_1); \ldots; (c_n : t_n, ci_n, co_n)]$ is an environment to collect information of non primitive class instances (*i.e.*, block components). c_i denotes the component name, t_i its type (*i.e.*, the block declaration's name). $ci_i = [u_1 \mapsto e_1; \ldots; u_n \mapsto e_n]$ contains the input variables of c_i. Each u_i denotes the name of an input variable and e_i is the expression bound to that variable. Similarly, $co_i = [y_1 \mapsto x_1;\ \ldots; y_n \mapsto x_n]$ contains the output variables of c_i. Each y_i denotes the name of an output variable and x_i is a local variable bound to that output variable.

$q = [x_1 = e_1;\ \ldots; x_n = e_n]$ stands for a list of equations, where x_i is a variable or parameter name and e_i is an expression.

Most of the function CId (defined in Figure 4) is rather straightforward. It seems worth mentioning that a parameter with a bound expression is added as equation to q while a parameter without a bound expression is added as input to i. The parameter modifications $(m_1 = e_1, \ldots, m_n = e_n)$ in component declarations are mapped to input variables with their respective bound expression in the block components environment j $([m_1 \mapsto e_1;\ \ldots; m_n \mapsto e_n])$.

Function CEq (defined in Figure 5) adds equations from a block to q, unless the equation includes a component dot access. In that case the block components environment j is modified and the equation is either mapped to the inputs *ci* or outputs *co* bindings of that component.

Once CEq returns the auxiliary representation struc-

Translate list of equations into SDFK equations:

$$\text{TEqList}_{(d,s)}(x_1 = e_1, \ldots, x_n = e_n) = x_1 = \text{TE}_{(d,s)}(e_1) \textbf{ and } \ldots \textbf{ and } x_n = \text{TE}_{(d,s)}(e_n)$$

Translate list of block instance contexts into SDFK equations with node instantiations:

$$\begin{aligned}
&\text{TCList}_{(d,s)}((c_1 : t_1, ci_1, co_1), \ldots, (c_n : t_n, ci_n, co_n)) \\
&\quad = \text{TC}_{(d,s)}(c_1 : t_1, ci_1, co_1) \textbf{ and } \ldots \textbf{ and } \text{TC}_{(d,s)}(c_n : t_n, ci_n, co_n) \\
&\text{T}(\textbf{block } id\ P \textbf{ equation } D \textbf{ end } id;) \\
&\quad = \text{let } d, i, o, l, s, j, q = \text{CEq}_{\text{CId}_{([],[],[],[],[],[],[])}(P)}(D) \text{ in} \\
&\qquad \textbf{node } id(is) = os \textbf{ with var } l \textbf{ in } qs \\
&\qquad \text{where } is = \text{SortList}(i), os = \text{SortList}(o), \\
&\qquad\quad qs = \text{TEqList}_{(d,s)}(q) \textbf{ and } \text{TCList}_{(d,s)}(j)
\end{aligned}$$

Figure 3: Translation from the *normalized* mACG-Modelica language into the SDFK language. The translation function T(.) utilizes functions CId (defined in Figure 4) and CEq (defined in Figure 5) to create auxiliary structures d, i, o, l, s, j, q which are further used in the translation functions TEqList and TCList and their supporting functions TE (defined in Figure 6) and TC (defined in Figure 7).

$$\begin{aligned}
&\text{CId}_{(d,i,o,l,s,j,q)}(t\ x) = (d + [x : t], i, o, l, s, j + [(x : t, [], [])], q) \text{ where } t \notin B \\
&\text{CId}_{(d,i,o,l,s,j,q)}(t\ x(m_1 = e_1, \ldots, m_n = e_n);) = (d + [x : t], i, o, s, j + [(x : t, [m_1 \mapsto e_1; \ \ldots; \ m_n \mapsto e_n], [])], q) \\
&\qquad \text{where } t \notin B \wedge m_i \neq \textbf{start} \\
&\text{CId}_{(d,i,o,l,s,j,q)}(t\ x) = (d + [x : t], i, o, l + [x : t], s, j, q) \text{ where } t \in B \\
&\text{CId}_{(d,i,o,l,s,j,q)}(t\ x(\textbf{start} = v)) = (d + [x : t], i, o, l + [x : t], s + [x : v], j, q) \text{ where } t \in B \\
&\text{CId}_{(d,i,o,l,s,j,q)}(\textbf{input } t\ x) = (d + [x : t], i + [x : t], o, l, s, j, q) \text{ where } t \in B \\
&\text{CId}_{(d,i,o,l,s,j,q)}(\textbf{input } t\ x(\textbf{start} = v)) = (d + [x : t], i + [x : t], o, l, s + [x : v], j, q) \text{ where } t \in B \\
&\text{CId}_{(d,i,o,l,s,j,q)}(\textbf{parameter } t\ x) = (d + [x : t], i + [x : t], o, l, s, j, q) \text{ where } t \in B \\
&\text{CId}_{(d,i,o,l,s,j,q)}(\textbf{parameter } t\ x = e) = (d + [x : t], i, o, l + [x : t], s, j, q + [x = e]) \text{ where } t \in B \\
&\text{CId}_{(d,i,o,l,s,j,q)}(\textbf{output } t\ x) = (d + [x : t], i, o + [x : t], s, j, q) \text{ where } t \in B \\
&\text{CId}_{(d,i,o,l,s,j,q)}(\textbf{output } t\ x(\textbf{start} = v)) = (d + [x : t], i, o + [x : t], l, s + [x : v], j, q) \text{ where } t \in B \\
&\text{CId}_{(d,i,o,l,s,j,q)}(P;) = \text{CId}_{(d,i,o,l,s,j,q)}(P) \\
&\text{CId}_{(d,i,o,l,s,j,q)}(P_1;\ P_2) = \text{CId}_{\text{CId}_{(d,i,o,l,s,j,q)}(P_1)}(P_2)
\end{aligned}$$

Figure 4: Function CId—Create auxiliary structure for instance declarations.

$$\begin{aligned}
&\text{CEq}_{(d+[x:t],i,o,l,s,j,q)}(x = e) = (d + [x : t], i, o, l, s, j, q + [x = e]) \\
&\text{CEq}_{(d+[c:t_1]+[x:t_2],i,o,l,s,j+[(c:t_1,ci,co)],q)}(x = c\ .\ y) = (d + [c : t_1] + [x : t_2], i, o, l, s, j + [(c : t_1, ci, co + [y \mapsto x])], q) \\
&\text{CEq}_{(d+[c:t],i,o,l,s,j+[(c:t,ci,co)],q)}(c\ .\ u = e) = (d + [c : t], i, o, l, s, j + [(c : t, ci + [u \mapsto e], co)], q) \\
&\text{CEq}_{(d,i,o,l,s,j,q)}(D;) = \text{CEq}_{(d,i,o,l,s,j,q)}(D) \\
&\text{CEq}_{(d,i,o,l,s,j,q)}(D_1;\ D_2) = \text{CEq}_{\text{CEq}_{(d,i,o,l,s,j,q)}(D_1)}(D_2)
\end{aligned}$$

Figure 5: Function CEq—Create auxiliary structure for equation declarations.

ture d, i, o, l, s, j, q, the elements i, o, and l translate (after applying SortList to i and o) directly to an SDFK node signature and its local variable declarations. Equations gathered in q are translated to SDFK equations by function TEqList, which in turn applies TE to each RHS expression in q. Function TE (defined in Figure 6) translates mACG-Modelica expressions into SDFK expressions.

$$
\begin{aligned}
&\mathrm{TE}_{(d,s)}(v) && = v\\
&\mathrm{TE}_{(d+[x:t],s)}(x) && = x\\
&\mathrm{TE}_{(d,s)}(\textbf{if } e_1 \textbf{ then } e_2 && = \textbf{if } \mathrm{TE}_{(d,s)}(e_1) \textbf{ then}\\
&\qquad \textbf{else } e_3) && \quad \mathrm{TE}_{(d,s)}(e_2) \textbf{ else}\\
& && \quad \mathrm{TE}_{(d,s)}(e_3)\\
&\mathrm{TE}_{(d,s+[x:v])}(\textbf{previous}(x)) && = v \textbf{ fby } x\\
&\mathrm{TE}_{(d,s)}(op(a_1,\ldots,a_n)) && = op(c_1,\ldots,c_n)\\
& && \quad \text{where } c_1,\ldots,c_n =\\
& && \quad \mathrm{TEList}_{(d,s)}(a_1,\ldots,a_n)\\
&\mathrm{TEList}_{(d,s)}(a_1,\ldots,a_n) && = (\mathrm{TE}_{(d,s)}(a_1),\ldots,\\
& && \quad \mathrm{TE}_{(d,s)}(a_n))
\end{aligned}
$$

Figure 6: Function TE—Translate mACG-Modelica expression into SDFK expression

Finally j, in which information of block component usage is collected, is translated to SDFK equations by function TCList, which in turn applies function TC to each element of j. Function TC (defined in Figure 7) translates collected component dot accesses to an SDFK equation with an RHS node instance. For example, a *normalized* mACG-Modelica code containing the dot accesses `c.u1=x1;`, `c.u2=x2; x3=c.y1; x4=c.y2;` for a component `c` would finally be translated into the SDFK equation `(x3,x4)=c(x1,x2)`.

3.7. Translator Implementation

A concrete translator was implemented using the Scala programming language Odersky et al. (2010) and the Kiama language processing library Sloane (2011).

The translator is structured in three main components:

Parser Lightweight parser implementation using Scala's parser combinator library (extended by additional functionality provided by the Kiama library).

Transformation Implementation of the presented multilevel translation approach. Taking advantage of the functional nature of Scala allows a rather direct and lean implementation.

$$
\begin{aligned}
&\mathrm{TC}_{(d,s)} && = os = t(is) \text{ where}\\
&(c:t, ci, co) && \quad is = \mathrm{TE}_{(d,s)}(e_1),\ldots,\mathrm{TE}_{(d,s)}(e_n)\\
& && \qquad \text{where}\\
& && \qquad \{(e_i)_{i=1}^{n=|ci|}\} = \{(e_i)_{i=1}^{n=|ci|} \mid\\
& && \qquad\quad ci + [u_i \mapsto e_i] \ \wedge\ u_i <_L u_{i+1}\}\\
& && \quad \text{and } os = (x_1,\ldots,x_n) \text{ where}\\
& && \qquad \{(x_i)_{i=1}^{n=|co|}\} = \{(x_i)_{i=1}^{n=|co|} \mid\\
& && \qquad\quad co + [y_i \mapsto x_i] \ \wedge\ y_i <_L y_{i+1}\}
\end{aligned}
$$

Figure 7: Function TC—Translate component dot accesses to an SDFK equation with an RHS SDFK node instance.

Emitter Emitters to the SDFK language defined in Section 3.1 and to Lustre code. The emitter component utilizes the functional pretty printing combinators provided by the Kiama library.

The SDFK output is not executable and needs to be checked statically (currently by manual inspection). The Lustre output allows taking advantage of the software infrastructure that is available around Lustre. Particularly, using Lustre as intermediate presentation, allows to generate executable C-code utilizing Verimag's Lustre V4 Toolbox[9]. This C-code can be used for dynamic testing. The translator supports that by allowing to generate appropriate Modelica code adapters ("*C code wrapper blocks*") that provide an interface from Modelica to the generated C code (using Modelica's external function interface). These wrappers can be directly loaded into Modelica simulation environments, enabling convenient back-to-back testing.

Note that the presented translation differs significantly from typical Modelica code generation as described in Section 2.3. Hence, a completely new translator implementation was needed to allow experimenting with the new approach.

4. Example

This section aims to present a short, yet still illustrative, example of how ACG-Modelica can be used for modeling practical relevant control functions. The example is presented in the context of a control design

[9] The Lustre V4 Toolbox is available from the VERIMAG research center (2014), http://www-verimag.imag.fr/.

for an electric drive system. However, it is focused on one particular aspect: The *digital realization* of a practical proportional-integral-derivative controller (PID controller).

4.1. PID Controller Realization

PID controllers constitute the most widespread control loop feedback mechanism used within industrial practice. Hence, they can be considered the "bread and butter" of control engineering. The "textbook" equation for a PID controller is

$$y(t) = k\left(e(t) + \frac{1}{T_i}\int^t e(s)\,\mathrm{d}s + T_d\frac{\mathrm{d}e(t)}{\mathrm{d}t}\right) \qquad (2)$$

where $y(t)$ is the controller output signal (≡ actuator input signal), $e(t) = u_s(t) - u_m(t)$ is the error between set-point signal $u_s(t)$ and measurement signal $u_m(t)$, k is the proportional gain of the controller, and T_i and T_d are the integral- and the derivative time constants. However, practical implementations are more elaborate than this.

For example, the Modelica Standard Library contains a continuous-time model of a PID controller (library path of the model: *Modelica.Blocks.Continuous.LimPID*) which incorporates several aspects of practical PID controller design which are described in (Åström and Hägglund, 1995, Chapter 3), namely *limited controller output*, *anti-windup compensation* and *set-point weighting*.

The following PID controller equation incorporates set-point weighting and accounts for the fact that practical implementations use a modified derivative ("D") term which is more robust against high-frequency content in the controller input signal (a typical source of high-frequency content is measurement noise). The controller equation is given in its Laplace transformed form using s as Laplace variable:

$$y(s) = k\Big(w_p u_s(s) - u_m(s) + \frac{1}{sT_i}\big(u_s(s) - u_m(s)\big) + \frac{sT_d}{1 + sT_d/N_d}\big(w_d u_s(s) - u_m(s)\big)\Big). \qquad (3)$$

The "D" part of the controller is approximated by $sT_d \approx \frac{sT_d}{1+sT_d/N_d}$, where N_d limits the gain at high frequencies (typically: $3 \le N_d \le 20$). Set-point weighting is provided by parameters w_p and w_d and allows to weight the set-point in the proportional and derivative part independently from the measurement.

A digital implementation requires a discrete-time representation. Following the example in (Åström and Wittenmark, 1997, Listing 8.1) the discretization is performed by using *forward differences*[10] for the integral term and *backward differences*[11] for the derivative term. A valid Z-domain representation for the integral term is therefore

$$y_I(z) = y_I(z)z^{-1} + \frac{h}{T_i}\big(u_s(z) - u_m(z)\big)z^{-1} \qquad (4)$$

where z is the Z-transform variable and h is the *sampling period*. The Z-domain representation is commonly used in the area of digital control systems or digital signal processing. Such a representation can be conveniently mapped to ACG-Modelica code by the substitution rule $xz^{-1} \rightarrow \textbf{previous}(x)$. Hence, a corresponding Modelica equation for Equation (4) is:

```
y_I = previous(y_I)
 + h/T_i*(previous(u_s) - previous(u_m));
```

The implementation of the derivative part follows in an analogous way.

Figure 8 shows the complete digital PID-controller model. Its structure (Figure 8a) is similar to the continuous-time version provided in the Modelica Standard Library, the parameters (except for sampling period "`h`") are the same as in the continuous-time version so that parameters found for the continuous-time PID-controller can be directly reused in the digital version (Figure 8b).

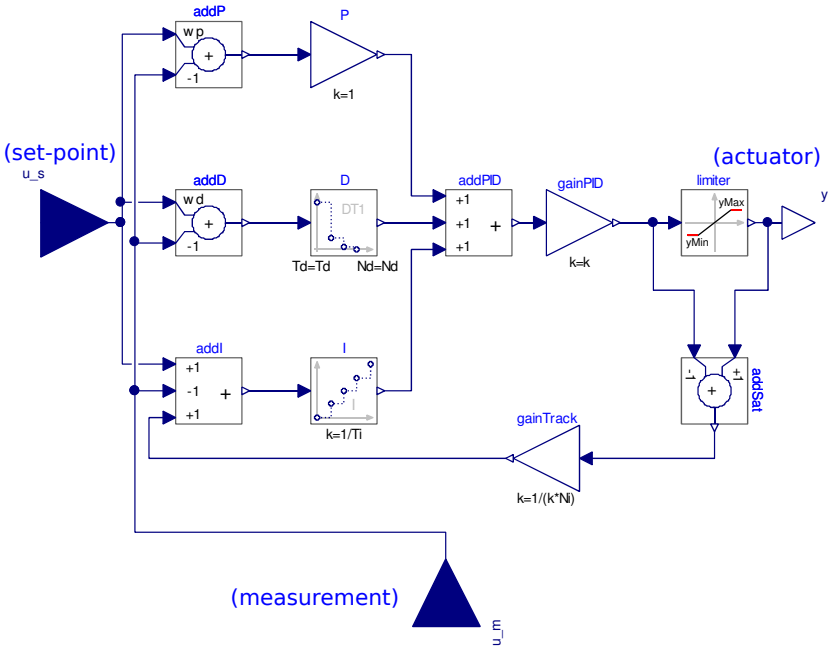
Anti-windup compensation is provided by an internal feedback loop which uses an error signal formed from the difference between the output of an actuator model (here the actor is modeled by a simple output limiter "`limiter`") and the output of the controller (i.e., the output of gain "`gainPID`") in order to drive the integrator to a value which makes the error signal equal to zero. This mechanism is identical to the one implemented in the continuous-time MSL version. Note that setting parameter `Td = 0` will give a controller with *PI controller characteristics*.

4.2. Code Generation and SIL Validation

Figure 9 shows a Software-in-the-Loop (SIL) configuration in which the PID controller from Figure 8 was translated to C code (using the tool chain described in Section 3.7) and imported back into Modelica by using Modelica's external C function interface. The generated C code is encapsulated in the "wrapper" block "`sILPI`". The digital PID block from Figure 8 is shown on the right side. It is fed with the same inputs as the "`sILPI`" block and is included in the model to allow a

[10] Forward differences are also known as Euler's method. This corresponds to the substitution rule $s \rightarrow \frac{z-1}{h}$ for the transformation from the Laplace-domain to the Z-domain.

[11] This corresponds to the substitution rule $s \rightarrow \frac{z-1}{zh}$ for the transformation from the Laplace-domain to the Z-domain.



(a) Component view of the digital PID-controller.

`h`	Sampling period
`k`	Gain of controller
`Ti`	Time constant of Integrator block
`Td`	Time constant of Derivative block
`yMax`	Upper limit of output
`yMin`	Lower limit of output
`wp`	Set-point weight for Proportional block (0..1)
`wd`	Set-point weight for Derivative block (0..1)
`Ni`	`Ni*Ti` is the time constant of the anti-windup compensation
`Nd`	The higher `Nd`, the more ideal the Derivative block (typical values between 3 to 20)

(b) Parameters of the the digital PID controller.

Figure 8: Digital PID controller with limited output, anti-windup compensation and set-point weighting.

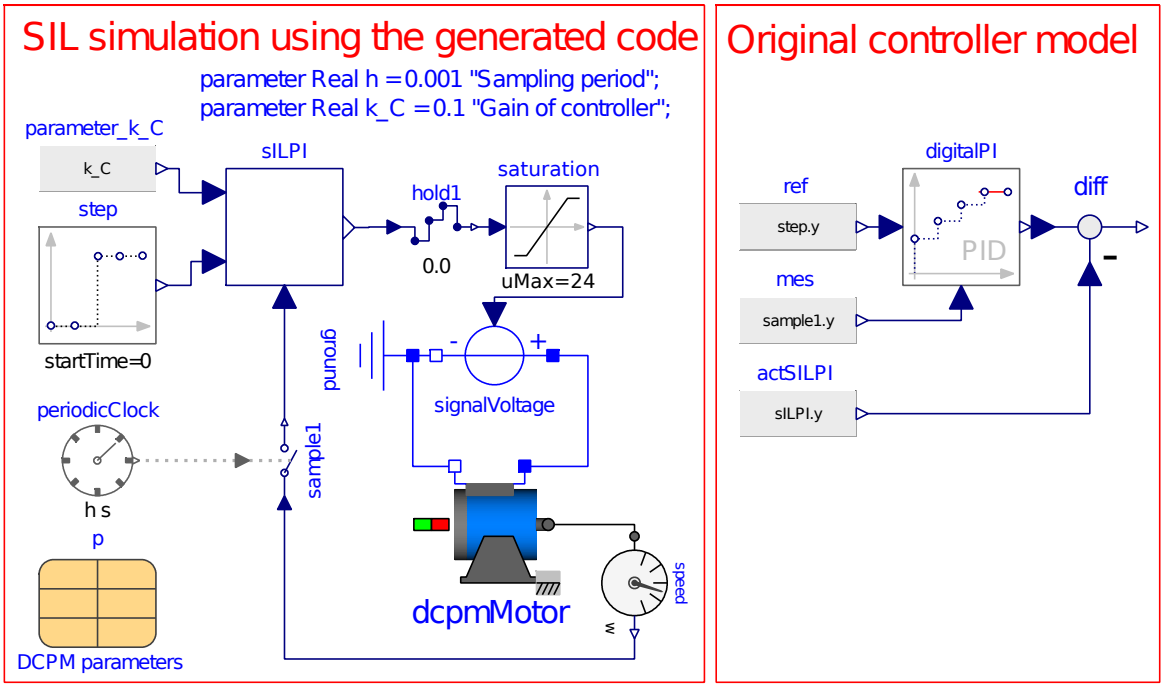


Figure 9: PI(D) controller and plant in a SIL configuration. The generated code is interfaced to the model by using a "wrapper" block ("`sILPI`"). For direct comparison, the original digital PID model (see Figure 8) is included as component "`digitalPI`".

direct comparison of "model" vs "software". The PID block regulates the speed of the DC permanent magnet machine "`dcpmMotor`". The block is configured as a PI controller (`Td = 0`), and the time constant of the integral part is set to a value that compensates the largest time constant of the motor model.

The controller gain "`k` = k_C" is provided as an input to the "software wrapper" block (the translation in Section 3.1 maps Modelica parameters to SDFK node inputs). Besides the top-level parameters "`k_C`" (controller gain) and "`h`" (sampling period for "`periodicClock`") the model has a parameter record component "`p`" which contains parameters for the motor model.The block "`saturation`" limits the actuator output to ±24 V. The sample element "`sample1`"has an additional input for a *clock* signal. Its origin is block "`periodicClock`" that defines a sampling period of $h = 0.001$ seconds. The hold element "`hold1`" implements a zero-order hold element. Hence, the digital control parts of the model will be executed with a sampling period of $h = 0.001$ seconds, while the physical parts are simulated as continuous-time systems. The blocks stem from the *Modelica_Synchronous* library[12] which is described in (Otter et al., 2012).

Figure 9 compares the step response of the SIL configuration for two gain values k_C. The plot shows that

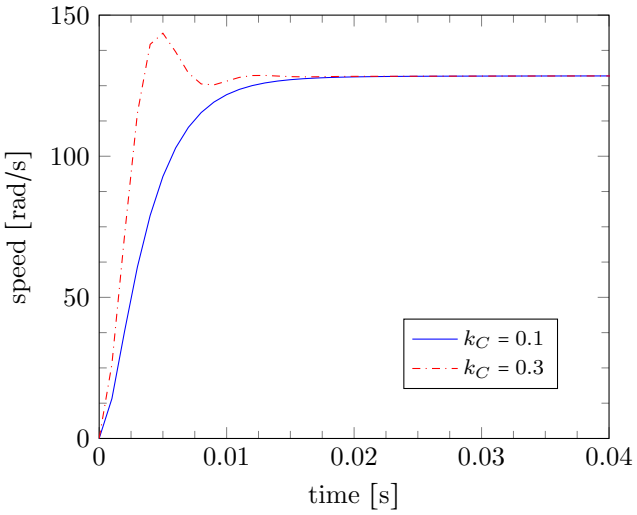


Figure 10: Step response of SIL model (with DCPM engine model and actuator saturation) for different controller gain values k_C.

the control performance cannot be improved arbitrarily by increasing the controller gain k_C.

Figure 11 shows a plot of the relative error between the output of the "controller software" encapsulated in the "wrapper" block "`sILPI.y`" and the output of the digital PID model "`digitalPI.y`". The difference

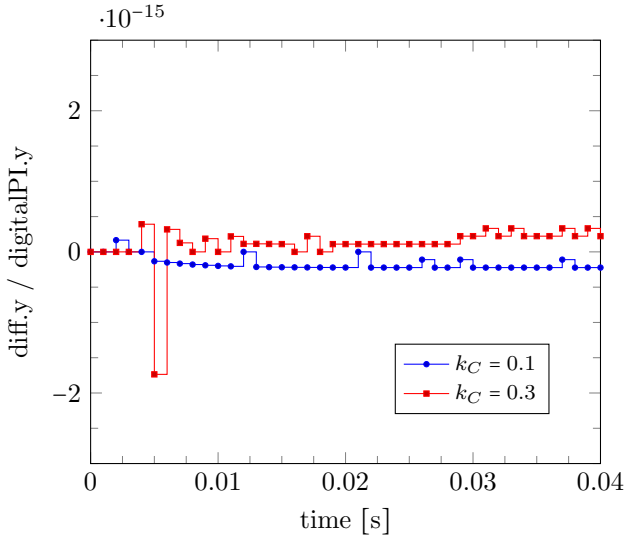


Figure 11: Relative error between the output of the "controller software" (i.e., code using the tool chain described in Section 3.7) and the "controller model".

between the "software" and the "model" stays within reasonable bounds. For $k_C = 0.3$ the actuator output is at its limits at the start and the anti-windup compensation with its associated equations is active. At $t = 0.005$ s the actuator output is close to zero and changes its sign for one tick, hence the sign change in the relative error.

4.3. Summary

The example demonstrates that the ACG-Modelica language subset is expressive enough in order to model a practically relevant digital control function. It furthermore indicates how Modelica allows to *integrate such discrete-time control models seamlessly with physical models* from the continuous-time domain. Modelica tools' support for hybrid models is based on solid mathematical foundations as well as long-term practical experiences in hybrid systems simulation. Hence, a model-based development process leveraging Modelica technology extended with high-integrity production code generation can be an attractive and powerful approach for the development of cyber-physical systems.

5. Discussion

The foregoing discussion is based on a rather restricted subset of the Modelica language. This allows to keep

[12] The library is available from the Modelica Association (2014), https://github.com/modelica/Modelica_Synchronous.

the scope of the translation within manageable bounds and keeps the language simple and well-defined. The considered subset is neither arbitrarily chosen nor without alternatives — this section comments design decisions and possible extensions.

5.1. Design Decisions

Languages restrictions and extensions for safety-related control applications in Modelica have been discussed in (Thiele et al., 2012). The mACG-Modelica subset is guided by the same considerations, but at the same time, the translation effort has also given additional insight. This section will briefly highlight important points that motivated the subset.

5.1.1. Clocked Variables

In the mACG-Modelica subset all variables are considered to be *clocked variables*. Clocked variables were introduced in Modelica 3.3 to improve the support for sampled-data systems. They provide improved modeling safety since the clock calculus ensures that only variables that are on the same clock (are active at the same time instant) can be combined in expressions (otherwise explicit clock conversion operators are needed). Previous Modelica support for sampled-data systems had an automatic sample and hold semantics for discrete-time variables so that this *temporal correctness* property could not be checked automatically by a compiler.

5.1.2. Causal Data-Flow without Algebraic Loops

Modelica is based on (acausal) equations. Hence, "`a*b = c`" is a valid gather and symbolic processing of the equation system (see Section 2.3) will transform the equation as needed. If symbolic processing identifies "`b`" as the *unknown* variable the equation will be transformed into the assignment "`b := c/a`". This is very convenient and powerful when modeling the *physical* part of a system, since physical "textbook" equations can be mapped naturally and directly to Modelica code. However, it can be problematic for (safety-relevant) digital control function modeling since symbolic transformations interfere with the ability of the developer to tightly control the evaluation of expressions. The transformation above is potentially harmful if "`a = 0`" cannot be excluded. It is conceivable to safeguard performed symbolic transformations, however, a simple solution is to only allow *causal* data-flow equations for the digital control part. This is the design decision used for the mACG-Modelica subset in which the left-hand side of an equation must be the unknown variable of that equation. Additionally, these data-flow equations may not contain algebraic loops. Solving algebraic loops needs (potentially unbounded) numerical iteration, which is not compatible with the hard real-time constraints of control applications.

5.1.3. Modular Code Generation

Modular code generation is understood identical to Biernacki et al. (2008) that define it as "*producing a transition function for each block definition and composing them together to produce the main transition function*". Support of modular code generation has considerable advantages:

- it facilitates to generate code that can be treated just like handcrafted code (*e.g.*, in order to pass generated code into a standard assurance process if the level of tool qualification is not sufficient for the development project at hand),
- it helps to establish a good traceability between model and generated code,
- it may decrease the size of the generated code, since a transition function that corresponds to a block can be reused for all instances of the block,
- it allows separate compilation of blocks which, on one hand has positive effects on the scalability of the development process and on the other hand allows to distribute modules without source code in order to protect intellectual property.

Modular code generation is known to impose stronger causality constraints when sorting the equations (see (Biernacki et al., 2008)). Hence, *design trade-offs* become necessary, *e.g.*, one might want to use modular code generation for high-level, highly-cohesive blocks, but not for basic arithmetic blocks. There are also approaches to modular code generation which impose less (or none) additional causality constraints. Lublinerman et al. (2009) provide a good discussion of the involved trade-offs and also present an alternative approach to modular code generation.

It is important to understand that the translation to the synchronous data-flow kernel language *enables* a straightforward path to modular code generation for Modelica, but it does not enforce a particular code generation approach. It is highly likely that an industry-relevant code generator would support modular code generation as a "per block" *option* and delegate necessary trade-off decisions to the developer.

5.1.4. Simplified Modifications

Modifications handling in Modelica is rather complex (Åkesson et al. (2010) provide a good exposition of the

challenges). Modifications allow to change aspects of a block/class (*e.g.*, parameter values) at instance declarations of that block/class. The difficulty lies in the possibility to nest modifications over several instance levels, *e.g.*, "`a(b(c=2))`". In mACG-Modelica modifications must not be nested, *e.g.*, "`b(c=2)`" is allowed, but not "`a(b(c=2))`".

This reduces the language complexity and is also advantageous with respect to modular code generation. The presented translation maps parameters to SDFK node inputs which means that a node that instantiates another node has to be aware of all the applicable parameters. Using this approach, supporting several levels of modifications would require to collect all applicable parameters of instantiated nodes and propagate them through by introducing them as additional inputs to the enclosing node. As a result node signatures (and consequently also function signatures of generated modular code) would grow significantly. Having simplified modifiers in the language keeps the size of node signatures within acceptable bounds.

Furthermore, restrictions on nesting modifications can be considered as beneficial in terms of code readability on the Modelica level — hence, restricting it (arguable) enforces a better modeling style.

5.2. Applicability to Extended Language Subsets

The multilevel translation approach presented in Section 3.3 allows adding new language elements in a way that keeps the lower level language representation untouched. This is thus an attractive way to further extend the considered language scope as long as a source-to-source transformation into a smaller kernel language can be given.

Essentially, a design trade-off needs to be made. On one hand the language should be as simple and well defined as possible to keep qualification efforts under control, but on the other hand control function developers need a sufficiently expressive modeling language in order to work efficiently.

5.2.1. Data typing and Hierarchical Scoping

Data typing and hierarchical scoping using **package** declarations has been left out of the discussion. Data typing is rather similar to typical general purpose languages and its discussion is considered to be out of scope of this work. The extension of the language with packages should be feasible by introducing an additional step to the aforementioned multilevel translation approach.

5.2.2. Multirate Control Systems

A more distinctive omission is the absence of clock conversion operators — hence, the considered language subset does not allow to model multirate systems. This is not as restrictive in practice as it may seem at first sight. Typically, generated code from behavioural models is integrated into an existing software architecture that handles *non-functional* aspects like communication and scheduling (see Figure 1). Linking the timely start of generated code sequences to activation events (*e.g.*, hardware timer interrupts) becomes therefore a responsibility of the software integration phase. In effect, this allows assigning different activation events to different code generation models during software integration and thus enables multirate systems.

Figure 12 illustrates this aspect on the basis of a multirate cascade control system for a very simple drive system. The objective is that the load inertia "`load`" follows the reference angle given by block "`reference`". The discrete-time controllers are connected to the

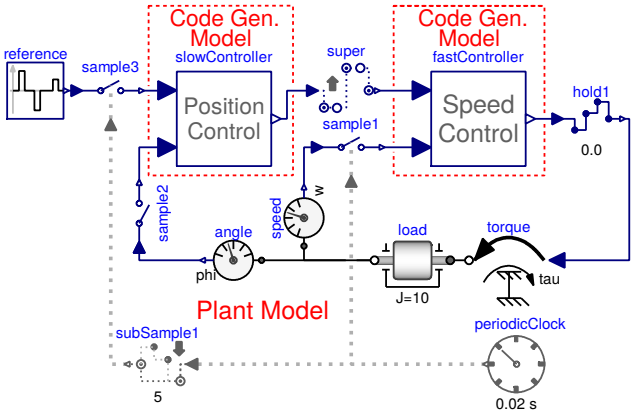


Figure 12: Multirate cascaded control loop with *Code Generation Models*. Note, that the rate-conversion elements ("`subSample1`", "`super`") and the clock are considered to be part of the *Plant Model*. Hence, it is not essential that the *qualified* code generator supports that elements.

continuous-time model parts by sample and hold elements. Sampling elements "`sample1`" and "`sample2`" have an additional input for a *clock* signal. The clock signal defines the activation instants of the controller blocks. Its origin is block "`periodicClock`" that defines a sampling period of 0.02 seconds. The rate transition element "`subSample1`" converts that sampling rate to 0.1 seconds. Hence, the inner speed control loop is faster than the outer position control loop. Element "`super`" provides slow-to-fast rate transition between

the two controller blocks. The utilized rate transition and clock blocks stem from the *Modelica_Synchronous* library. Otter et al. (2012) describe that library in detail.

While the complete multirate system can be modeled and simulated in Modelica, a qualified code generator would not need to support the rate-conversion elements: block "`slowController`" and "`fastController`" can be translated *separately* into sequential code and activation and rate-conversion can be offloaded to the software integration phase. Also note that clock "`periodicClock`" is not part of the code generation model, it is also considered to be part of the plant model. Hence, the omission of clock conversion operators in the behavioral model for code generation does not prohibit multirate models on the global system level.

5.2.3. Inheritance

Inheritance is not supported in the language subset. A full support of Modelica's multiple inheritance (including its related modification and redeclaration) features would increase the language complexity considerably and therefore complicate tool qualification efforts. However, extending the subset with a restricted support of inheritance seems quite feasible.

5.2.4. State Machines

Control application often consists of both: data-flow parts that are naturally described with block diagrams and system logic parts which are described more naturally using state machine formalisms. Modelica 3.3 introduced state machines with a comparable modeling power as Statecharts (Harel, 1987) as built-in language elements (Modelica Association, 2012, Chapter 17). Extending the presented translation with support for state machines is desirable, but is a challenge on its own.

6. Conclusions

Despite its suitability for integrated modeling and simulation of multi-domain physical systems and sampled-data systems, the use of Modelica for embedded systems development was so far limited. A model-driven development process for embedded systems is so far impeded by the lack of tools for qualifiable automatic code generation from Modelica.

To mitigate that deficiency this article presented a translational semantics of a Modelica subset for control applications to a synchronous data-flow kernel language. The synchronous data-flow kernel allows to resort to published and well-established compilation techniques which are accepted by certification authorities.

In addition to the formal translation a concrete prototype translator was implemented. The translator additionally supports emitting Lustre code for which the software infrastructure around Lustre can be leveraged. This demonstrates that the translation equations can also serve as a base to create a gateway from Modelica to established tools based on synchronous data-flow.

Enabling the proposed approach requires suitable design decisions regarding the supported Modelica subset. The necessary trade-offs that resulted in the considered data-flow based language subset for control application have been exposed and possible extensions have been discussed. Particularly, extending the subset with a state machine formalism seems a desirable task for the future.

The presented translation to a synchronous data-flow kernel language opens up new paths towards a qualifiable automatic code generator for Modelica that can directly utilize well-accepted methodology and technology for high-integrity software development.

Acknowledgements

The first author would like to thank Martin Otter and Dirk Zimmer from the German Aerospace Center (DLR) who supervised and supported him during and after his time at DLR where this work was initially started.

References

Benveniste, A., Edwards, S. A., Halbwachs, N., Le Guernic, P., and de Simone, R. The synchronous languages 12 years later. In *Proceedings of the IEEE*, volume 91 (1). pages 64–83, 2003. doi:10.1109/JPROC.2002.805826.

Biernacki, D., Colaço, J.-L., Hamon, G., and Pouzet, M. Clock-directed modular code generation for synchronous data-flow languages. *SIGPLAN Not.*, 2008. 43(7):121–130. doi:10.1145/1379023.1375674.

Boussinot, F. and De Simone, R. The ESTEREL language. *Proceedings of the IEEE*, 1991. 79(9):1293–1304. doi:10.1109/5.97299.

Broman, D. *Meta-Languages and Semantics for Equation-Based Modeling and Simulation*. Ph.D. thesis, Linköping University, PELAB - Programming Environment Laboratory, The Institute of Technology, 2010.

Broy, M., Krcmar, H., Zimmermann, J., and Kirstan, S. Einfluss des Software-Designs auf die Wirtschaftlichkeit von Software-Entwicklungen. *ATZelektronik*, 2011. 02:34–37.

Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S., and Niebert, P. From Simulink to SCADE/Lustre to

TTA: a layered approach for distributed embedded applications. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, LCTES '03. ACM, New York, NY, USA, pages 153–162, 2003. doi:10.1145/780732.780754.

Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. A. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87. ACM, New York, NY, USA, pages 178–188, 1987. doi:10.1145/41625.41641.

Colaço, J.-L., Pagano, B., and Pouzet, M. A conservative extension of synchronous data-flow with state machines. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05. ACM, New York, NY, USA, pages 173–182, 2005. doi:10.1145/1086228.1086261.

Colaço, J.-L. and Pouzet, M. Clocks as First Class Abstract Types. In R. Alur and I. Lee, editors, *Embedded Software*, volume 2855 of *Lecture Notes in Computer Science*, pages 134–155. Springer Berlin Heidelberg, 2003. doi:10.1007/978-3-540-45212-6_10.

Ebert, C. and Jones, C. Embedded software: Facts, figures, and future. *Computer*, 2009. 42:42–52. doi:10.1109/MC.2009.118.

Elmqvist, H., Otter, M., and Mattsson, S. E. Fundamentals of Synchronous Control in Modelica. In M. Otter and D. Zimmer, editors, 9$^{\text{th}}$ *Int. Modelica Conference*. Munich, Germany, 2012. doi:10.3384/ecp1207615.

Frank, S., Grabmüller, M., Hofstedt, P., Kleeblatt, D., Pepper, P., Mai, P. R., and Schneider, S.-A. Safety of Compilers and Translation Techniques – Status quo of Technology and Science. In *Automotive – Safety & Security*. 2008.

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

Fritzson, P., Pop, A., Broman, D., and Aronsson, P. Formal Semantics Based Translator Generation and Tool Development in Practice. In *Software Engineering Conference, 2009. ASWEC '09. Australian.* pages 256–266, 2009. doi:10.1109/ASWEC.2009.46.

Hamon, G. and Pouzet, M. Modular Resetting of Synchronous Data-flow Programs. In *ACM International conference on Principles of Declarative Programming (PPDP'00)*. Montreal, Canada, 2000. doi:10.1145/351268.351300.

Harel, D. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 1987. 8(3):231–274. doi:10.1016/0167-6423(87)90035-9.

Hatcliff, J., Wassyng, A., Kelly, T., Comar, C., and Jones, P. Certifiably Safe Software-Dependent Systems: Challenges and Directions. In *Proceedings of the on Future of Software Engineering*, FOSE 2014. ACM, Hyderabad, India, 2014. doi:10.1145/2593882.2593895.

ISO 26262-8:2011. Road vehicles – Functional safety – Part 8: Supporting processes. International Organization for Standardization, 2011.

Åkesson, J., Ekman, T., and Hedin, G. Implementation of a Modelica compiler using JastAdd attribute grammars. *Science of Computer Programming*, 2010. 75(1–2):21–38. doi:10.1016/j.scico.2009.07.003. Special Issue on ETAPS 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA '06 and '07).

Kågedal, D. and Fritzson, P. Generating a Modelica compiler from natural semantics specifications. In *Proceedings of the 1998 Summer Computer Simulation Conference (SCSC'98)*. 1998.

LeGuernic, P., Gautier, T., Le Borgne, M., and Le Maire, C. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 1991. 79(9):1321–1336. doi:10.1109/5.97301.

Lublinerman, R., Szegedy, C., and Tripakis, S. Modular code generation from synchronous block diagrams: Modularity vs. code size. In *ACM SIGPLAN Notices*, volume 44. ACM, pages 78–89, 2009. doi:10.1145/1594834.1480893.

Modelica Association. Modelica—A Unified Object-Oriented Language for Systems Modeling v3.3. Standard Specification, 2012. Available at http://www.modelica.org/.

Odersky, M., Spoon, L., and Venners, B. *Programming in Scala*. Artima Press, second edition, 2010.

Otter, M., Thiele, B., and Elmqvist, H. A Library for Synchronous Control Systems in Modelica. In M. Otter and D. Zimmer, editors, 9$^{\text{th}}$ *Int. Modelica Conference*. Munich, Germany, 2012. doi:10.3384/ecp1207627.

Sauvage, S. and Bouali, A. Development Approaches in Software Development. In *Embedded Real Time Software (ERTS)*. Toulouse, France, 2006.

Schneider, S.-A., Lovric, T., and Mai, P. R. The Validation Suite Approach to Safety Qualification of Tools. In *SAE World Congress*. SAE International, Detroit, MI, USA, 2009. doi:10.4271/2009-01-0746.

Schäuffele, J. and Zurawka, T. *Automotive Software Engineering*. Vieweg + Teubner, Wiesbaden, 4 edition, 2010.

Sloane, A. M. Lightweight Language Processing in Kiama. In J. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 408–425. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-18023-1_12.

Åström, K. J. and Hägglund, T. *PID Controllers: Theory, Design, and Tuning*. Instrument Society of America, 1995.

Åström, K. J. and Wittenmark, B. *Computer-Controlled Systems: Theory and Design*. Prentice-Hall, Inc., 1997.

Stürmer, I., Conrad, M., Fey, I., and Dörr, H. Experiences with Model and Autocode Reviews in Model-based Software Development. In *Proceedings of the 2006 international workshop on Software engineering for automotive systems*, SEAS '06. ACM, New York, NY, USA, pages 45–52, 2006. doi:10.1145/1138474.1138483.

Thiele, B., Schneider, S.-A., and Mai, P. R. A Modelica Sub-and Superset for Safety-Relevant Control Applications. In M. Otter and D. Zimmer, editors, 9$^{\text{th}}$ *Int. Modelica Conference*. Munich, Germany, 2012. doi:10.3384/ecp12076455.

A. Auxiliary Functions

$$
\begin{aligned}
&\_ <_L \_ && = \text{Lexicographical order relation} \\
&\text{SortList}(xs) && = [x_1 : t_1; \ldots; x_n : t_n] \text{ where} \\
& && \quad \{(x_i : t_i)_{i=1}^{n=|xs|}\} = \{(x_i : t_i)_{i=1}^{n=|xs|} \mid xs + [x_i : t_i] \ \wedge \ x_i <_L x_{i+1}\}
\end{aligned}
$$

Figure 13: Lexicographical order relation and function SortList—Auxiliary relation and function for arranging node input or output arguments in alphabetical order. The elements of the list xs have the structure $name : type$. They are sorted by a suitable lexicographical order relation on $name$.

B. Normalization

B.1. Generation of connection equations

The generation of connection equations is defined as a source-to-source transformation through a set of mutually recursive functions. $\mathrm{G}(D)$ at the bottom of Figure 14 defines the translation of Modelica code that includes **`connect`**-equations into a Modelica code in which these **`connect`**-equations are replaced by simple connection equations of the form "$x = e$".

$\mathrm{G}(D)$ relies on GCenv(..) to create an environment (I, O, Bs) which is utilized in the translation function $\mathrm{GTcd}_{(I,O,Bs)}(D)$.

GTcd traverses the class declarations. It removes the connector declarations and utilizes function GTd (see Figure 15) to replace **`connect`**-equations by simple equations of the form "$x = e$".

The environment (I, O, Bs) is a tuple of (globally) declared input connectors I, output connectors O and block declarations Bs. $I = [i_1 : t_1; \ \ldots; \ i_n : t_n]$ is a list of input connector short class definitions where i denotes the class name and t denotes the respective (primitive) data type associated to that input connector. $O = [o_1 : t_1; \ \ldots; \ o_n : t_n]$ is a list of output connector short class definitions where o denotes the class name and t denotes the respective (primitive) data type associated to that output connector. $Bs = [Bd_1; \ \ldots; \ Bd_n]$ is the list of block declarations

Each element $Bd = (b, d)$ is a tuple there b denotes the block class name and d stands for a list of component declarations in b. $d = [x_1 : t_1; \ \ldots; \ x_n : t_n]$ is composed from the component names x and their respective type t.

B.2. Modification and Dot Access Normalization

This normalization step includes the stripping of parameter modifications, the normalizing of component modifications and the extraction of component dot accesses appearing in expressions. As previously, it is defined as a source-to-source transformation through a set of mutually recursive functions. The translation makes use of a couple of auxiliary functions defined in Figure 16. A notable complication of the name decoration function $\mathrm{dn}_{(D,p)}(x)$ in Figure 16 is that it may be used at places there it is required that the decorated name introduces a fresh variable — *i.e.*, a variable of the same name must not already exist in its scope. To ensure that a fresh name is introduced one may provide a set of names D which are forbidden as a result of name decoration. The second equation in Figure 16 handles cases in which the resulting name would be in D by prefixing an additional "_" to the name until the resulting name is not any longer in D.

Function $\mathrm{N}(D)$ at the bottom of Figure 17 defines the translation from the mACG-Modelica language into the *normalized* mACG-Modelica language. This is achieved in three steps:

1. Creation of an auxiliary representation structure $C = \mathrm{NCenv}_{[]}(D)$ (defined in Figure 18) which encodes the mACG-Modelica block declarations "D" in a structure that facilitates the normalization transformation in the subsequent step,

$$
\begin{aligned}
&\mathrm{GTcd}_{(I,O,Bs+[(id,d)])}\\
&\quad(\textbf{block } id\ D \textbf{ equation } E \textbf{ end } id) && = \textbf{block } id\\
&\qquad \mathrm{GTd}_{\left((I,O,Bs+[(id,d)]),(id,d)\right)}(D)\\
&\qquad \textbf{equation}\\
&\qquad \mathrm{GTd}_{\left((I,O,Bd+[(id,d)]),(id,d)\right)}(E)\\
&\qquad \textbf{end } id;\\
&\mathrm{GTcd}_{(I+[id:t],O,Bs)}(\textbf{connector } id = \textbf{input } t) && = \quad \text{, remove declaration}\\
&\mathrm{GTcd}_{(I,O+[id:t],Bs)}(\textbf{connector } id = \textbf{output } t) && = \quad \text{, remove declaration}\\
&\mathrm{GTcd}_{(I,O,Bd)}(D;) && = \mathrm{GTcd}_{(I,O,Bd)}(D)\\
&\mathrm{GTcd}_{(I,O,Bd)}(D_1; D_2) && = \mathrm{GTcd}_{\mathrm{GTcd}_{(I,O,Bd)}(D_1)}(D_2)\\
&\mathrm{Gd}_{(d)}(t\ x) && = d + [x:t]\\
&\mathrm{Gd}_{(d)}(t\ x(m_1 = e_1, \ldots, m_n = e_n)) && = d + [x:t]\\
&\mathrm{Gd}_{(d)}(a) && = d, \text{ for the remaining forms of } a\\
&\mathrm{Gd}_{(d)}(D_1;\ \ldots;\ D_n) && = \text{let } d_1 = \mathrm{Gd}_{(d)}(D_1) \text{ in}\\
&\qquad \ldots \text{ let } d_n = \mathrm{Gd}_{(d_{n-1})}(D_n) \text{ in}\\
&\qquad d_n\\
&\mathrm{GCenv}_{(I,O,Bs)}(&& = \text{let } d = \mathrm{Gd}_{([])}(D) \text{ in}\\
&\quad \textbf{block } id\ D \textbf{ equation } E \textbf{ end } id) && \quad (I, O, Bs + [(id, d)])\\
&\mathrm{GCenv}_{(I,O,Bs)}(\textbf{connector } id = \textbf{input } t) && = (I + [id:t], O, Bs)\\
&\mathrm{GCenv}_{(I,O,Bs)}(\textbf{connector } id = \textbf{output } t) && = (I, O + [id:t], Bs)\\
&\mathrm{GCenv}_{(I,O,Bs)}(D;) && = \mathrm{GCenv}_{(I,O,Bs)}(D)\\
&\mathrm{GCenv}_{(I,O,Bs)}(D_1; D_2) && = \mathrm{GCenv}_{\mathrm{GCenv}_{(I,O,Bs)}(D_1)}(D_2)\\
&\mathrm{G}(D) && = \text{let } (I, O, Bs) = \mathrm{GCenv}_{([],[],[])}(D) \text{ in}\\
&\qquad \mathrm{GTcd}_{(I,O,Bs)}(D)
\end{aligned}
$$

Figure 14: Generation of connection equations.

$\mathrm{GTd}_{\left((I+[t{:}t_I],O,Bs),Bd\right)}(t\ x)$	$=$ **input** $t_I\ x$
$\mathrm{GTd}_{\left((I+[t{:}t_I],O,Bs),Bd\right)}(t\ x(\mathbf{start}=v))$	$=$ **input** $t_I\ x(\mathbf{start}=v)$
$\mathrm{GTd}_{\left(I,O+[t{:}t_O],Bs),Bd\right)}(t\ x)$	$=$ **output** $t_O\ x$
$\mathrm{GTd}_{\left(I,O+[t{:}t_O],Bs),Bd\right)}(t\ x(\mathbf{start}=v))$	$=$ **output** $t_O\ x(\mathbf{start}=v)$
x – internal input, y – internal output:	
$\mathrm{GTd}_{\left((I+[t_1{:}\_],O+[t_2{:}\_],Bs),\ (b,d_b+[x{:}t_1]+[y{:}t_2])\right)}(\mathbf{connect}(x,y))$	$= y = x$
x – internal output, y – internal input:	
$\mathrm{GTd}_{\left((I+[t_2{:}\_],O+[t_1{:}\_],Bs),\ (b,d_b+[x{:}t_1]+[y{:}t_2])\right)}(\mathbf{connect}(x,y))$	$= x = y$
$c.x$ – external input, y – internal input:	
$\mathrm{GTd}_{\left((I+[t_1{:}\_]+[t_2{:}\_],O,Bs+[(t_c,d_c+[x{:}t_1])]),\ (b,t_b+[c{:}t_c]+[y{:}t_2]])\right)}(\mathbf{connect}(c.x,y))$	$= c.x = y$
$c.x$ – external output, y – internal output:	
$\mathrm{GTd}_{\left((I,O+[t_2{:}\_]+[t_1{:}\_],Bs+[(t_c,d_c+[x{:}t_1])]),\ (b,d_b+[c{:}t_c]+[y{:}t_2])\right)}(\mathbf{connect}(c.x,y))$	$= y = c.x$
$\mathrm{GTd}_{(Cd,Bd)}(\mathbf{connect}(y,c.x))$	$= \mathrm{GTd}_{(Cd,Bd)}(\mathbf{connect}(c.x,y))$
$c.x$ – external input, $c.y$ – external output:	
$\mathrm{GTd}_{\left((I+[t_1{:}\_],o+[t_2{:}\_],Bs+[(t_c,d_c+[x{:}t_1])]+[(t_m,d_m+[y{:}t_2])]),\ (b,d_b+[c{:}t_c]+[m{:}t_m])\right)}(\mathbf{connect}(c.x,m.y))$	$= c.x = m.y$
$c.x$ – external output, $c.y$ – external input:	
$\mathrm{GTd}_{\left((I+[t_2{:}\_],O+[t_1{:}\_],Bs+[(t_c,d_c+[x{:}t_1])]+[(t_m,d_m+[y{:}t_2])]),\ (b,d_b+[c{:}t_c]+[m{:}t_m])\right)}(\mathbf{connect}(c.x,m.y))$	$= m.y = c.x$
$\mathrm{GTd}_{(Cd,Bd)}(a)$	$= a;$, for the remaining forms of a
$\mathrm{GTd}_{(Cd,Bd)}(D_1;\ \ldots;\ D_n;)$	$= \mathrm{GTd}_{(Cd,Bd)}(D_1);\ \ldots;\ \mathrm{GTd}_{(Cd,Bd)}(D_n);$

Figure 15: Function GTd—Replace **connect**-equations by equations of the form "$x = e$".

Name decoration, combining p, x to _p_x:

$\mathrm{dn}_{(D,p)}(x) = \_p\_x, \quad \text{if } x \text{ is an identifier and } \_p\_x \notin D$

If a variable name $\_p\_x$ already exists in D, try $\_\_p\_x$ instead:

$\mathrm{dn}_{(D,p)}(x) = \mathrm{dn}_{(D,\_p)}(x), \quad \text{if } x \text{ is an identifier and } \_p\_x \in D$

$\mathrm{dn}_{(D,p)}(v) = v, \quad \text{if } v \text{ is a value or } \bot$

$\mathrm{dn}_{(D,p)}(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) = \text{let } (a_1, a_2, a_3) = \mathrm{dnList}_{(D,p)}(e_1, e_2, e_3) \text{ in } \textbf{if } a_1 \textbf{ then } a_2 \textbf{ else } a_3$

$\mathrm{dn}_{(D,p)}(op(a_1, \ldots, a_n)) = \text{let } c_1, \ldots, c_n = \mathrm{dnList}_{(D,p)}(a_1, \ldots, a_n) \text{ in } op(c_1, \ldots, c_n)$

$\mathrm{dnList}_{(D,p)}(a_1, \ldots, a_n) = (\mathrm{dn}_{(D,p)}(a_1), \ldots, \mathrm{dn}_{(D,p)}(a_n))$

Extract names from lists of parameters, variables or (non-primitive) instance declarations:

$\mathrm{Vars}([x_1 : t_1; \ \ldots; x_n : t_n]) = \{x_1, \ldots, x_n\}$

$\mathrm{Vars}([x_1 : t_1 : e_1; \ \ldots; x_n : t_n : e_n) = \{x_1, \ldots, x_n\}$

$\mathrm{Vars}([(x_1 : t_1, cm_1); \ \ldots; (x_n : t_n, cm_n)]) = \{x_1, \ldots, x_n\}$

$\mathrm{Vars}((i, o, l)) = \mathrm{Vars}(i) \cup \mathrm{Vars}(o) \cup \mathrm{Vars}(l)$

Join a list of declarations using ";" as separator:

$\mathrm{JoinDecl}(d) = d;$

$\mathrm{JoinDecl}(d, r) = d; \ \mathrm{JoinDecl}(r)$

Figure 16: Auxiliary functions for the normalization.

2. application of the normalization transformation to the auxiliary structure, $C_N = \mathrm{NormC}(C)$ (defined in Figure 19 and discussed in more detail later), and finally

3. output of the auxiliary structure as *normalized* mACG-Modelica using function $\mathrm{NdeclList}(C_N)$.

The auxiliary representation structure $C = [(b_1, p_1, m_1, r_1, q_1); \ \ldots; \ (b_n, p_n, m_n, r_n, q_n)]$ is a list of block class declarations within the (global) environment. b_i is a class declaration name. $p_i = [x_1 : t_1 : e_1; \ \ldots; \ x_n : t_n : e_n]$ is the list of parameter declarations of block b_i with their respective type t_i and an optionally bound expression e_i. The case that no expressions is bound to a parameter e_i is denoted by the symbol $\bot$ (*e.g.*, $x : t : \bot$).

$m_i = [(c_1 : t_1, cm_1); \ \ldots; \ (c_n : t_n, cm_n)]$ is the list of (non-primitive) class instances appearing in block b_i, where c_i denotes the instance name, t_i denotes the respective type, and $cm_i = [x_1 = e_1; \ \ldots; \ x_n = e_n]$ is the list of respective class modifications.

r_i is partitioned into the tuple $r_i = (i_i, o_i, l_i)$ where $i_i = [i_1 : t_1; \ \ldots; \ i_n : t_n]$ is the list of inputs to block b_i with their respective primitive types, $o = [o_1 : t_1; \ \ldots; o_n : t_n]$ is the list of outputs from block b_i with their respective primitive types, and $l = [l_1; \ \ldots; l_n])$ is the list of *all* local class declarations of primitive type in block b_i, *except for* parameter declarations (*i.e.*, the list includes input, output and local variable declarations).

$q_i = [x_1 = e_1; \ \ldots; \ x_n = e_n]$ is the list of equations declared in block b_i.

Function NormC in Figure 19 performs the actual normalization transformation on the auxiliary block representation structure C constructed in function NCenv (see Figure 18). It applies functions NB and NbInst to each block. The translation differs depending on whether a block is used as instance in another block, or whether a block is used as the top-level instance. Although the final program will have only one top-level instance, function NormC always creates two versions of a block: One version that is used if the block is used as instance in another block (using function NB) and one version that is used if the block is used as top-level instance (using function NbInst which is described in more detail in Section B.3).

Function NB first strips all parameter modifications from a block using the auxiliary function NBstrip. This is necessary, because the parameters of a block will later be transformed to node inputs in the SDFK language

$$
\begin{array}{ll}
\text{Ncomponent}(c, ct, cm) & = \text{let } [m_1 = e_1;\ \ldots;\ m_n = e_n] = cm \text{ in} \\
 & \quad ct\ c(m_1 = e_1, \ldots, m_n = e_n) \\
\text{Nblock}(b, & = \text{let } pas = (\textbf{parameter } pat_i\ pa_i = pae_i)_{i \in \{1,\ldots,j\}} \text{ in} \\
\quad [pa_1 : pat_1 : pae_1; & \quad \text{let } pbs = (\textbf{parameter } pbt_i\ pb_i)_{i \in \{1,\ldots,k\}} \text{ in} \\
\quad\quad \ldots;\ pa_j : pat_j : pae_j] & \quad \text{let } ms = (\text{Ncomponent}(c_i, ct_i, cm_i))_{i \in \{1,\ldots,n\}} \text{ in} \\
\quad + [pb_1 : pbt_1 : \bot; & \quad \textbf{block } b \\
\quad\quad \ldots; pb_k : pbt_k : \bot], & \quad\quad \text{JoinDecl}(l) \\
\quad [(c_1 : ct_1, cm_1); & \quad\quad \text{JoinDecl}(pas) \\
\quad\quad \ldots;\ (c_n : ct_n, cm_n)], & \quad\quad \text{JoinDecl}(pbs) \\
\quad (i, o, l), q) & \quad\quad \text{JoinDecl}(ms) \\
 & \quad \textbf{equation} \\
 & \quad\quad \text{JoinDecl}(q) \\
 & \quad \textbf{end } b; \\
\text{NdeclList}(b) & = \text{Nblock}(b); \\
\text{NdeclList}(b, r) & = \text{Nblock}(b); \\
 & \quad \text{NdeclList}(r) \\
\text{N}(D) & = \text{let } C = \text{NCenv}_{([])}(D) \text{ in} \\
 & \quad \text{let } C_N = \text{NormC}(C) \text{ in} \\
 & \quad \text{NdeclList}(C_N)
\end{array}
$$

Figure 17: Translation from the mACG-Modelica language into the *normalized* mACD-Modelica language. The translation function N(D) utilizes functions NCenv (Figure 18) and NormC (Figure 19) for the transformation and the remaining functions for transforming the auxiliary representation structure C_N to *normalized* mACG-Modelica.

$$
\begin{aligned}
&\mathrm{Na}_{(b,p,m,r,q)}(x = e) && = (b, p, m, r, q + [x = e])\\
&\mathrm{Na}_{(b,p,m,r,q)} && = (b, p, m + [(x : t, [cm_1 = e_1;\ \ldots;\ cm_n = e_n])], r, q)\\
&\quad (t\ x(cm_1 = e_1, \ldots, cm_n = e_n)) && \quad \text{where } t \notin B\\
&\mathrm{Na}_{(b,p,m,r,q)}(t\ x) && = (b, p, m + [(x : t, [])], r, q) \text{ where } t \notin B\\
&\mathrm{Na}_{(b,p,m,r,q)}(\textbf{parameter}\ t\ x = e) && = (b, p + [x : t : e], m, r, q) \text{ where } t \in B\\
&\mathrm{Na}_{(b,p,m,r,q)}(\textbf{parameter}\ t\ x) && = (b, p + [x : t : \bot], m, r, q) \text{ where } t \in B\\
&\mathrm{Na}_{(b,p,m,(i,o,l),q)}(\textbf{input}\ t\ x) && = (b, p, m, (i + [x : t], o, l + [\textbf{input}\ t\ x]), q)\\
& && \quad \text{where } t \in B\\
&\mathrm{Na}_{(b,p,m,(i,o,l),q)} && = (b, p, m, (i + [x : t], o, l + [\textbf{input}\ t\ x(\textbf{start} = v)]), q)\\
&\quad (\textbf{input}\ t\ x(\textbf{start} = v)) && \quad \text{where } t \in B\\
&\mathrm{Na}_{(b,p,m,(i,o,l),q)}(\textbf{output}\ t\ x) && = (b, p, m, (i, o + [x : t], l + [\textbf{output}\ t\ x]), q)\\
& && \quad \text{where } t \in B\\
&\mathrm{Na}_{(b,p,m,(i,o,l),q)} && = (b, p, m, (i, o + [x : t], l + [\textbf{output}\ t\ x(\textbf{start} = v)]), q)\\
&\quad (\textbf{output}\ t\ x(\textbf{start} = v)) && \quad \text{where } t \in B\\
&\mathrm{Na}_{(b,p,m,(i,o,l),q)}(t\ x) && = (b, p, m, (i, o, l + [t\ x]), q) \text{ where } t \in B\\
&\mathrm{Na}_{(b,p,m,(i,o,l),q)}(t\ x(\textbf{start} = v)) && = (b, p, m, (i, o, l + [t\ x(\textbf{start} = v)]), q) \text{ where } t \in B\\
&\mathrm{NCdec}_{(b,p,m,r,q)}(d;) && = \mathrm{Na}_{(b,p,m,r,q)}(d)\\
&\mathrm{NCdec}_{(b,p,m,r,q)}(d_1;\ d_2) && = \mathrm{NCdec}_{\mathrm{Na}_{(b,p,m,r,q)}(d_1)}(d_2)\\
&\mathrm{NCenv}_{(C)}(\textbf{block}\ id\ D && = C + [\mathrm{NCdec}_{\mathrm{NCdec}_{(id,[],[],[],[])}(D)}(E)]\\
&\quad \textbf{equation}\ E\ \textbf{end}\ id) &&\\
&\mathrm{NCenv}_{(C)}(D;) && = \mathrm{NCenv}_{(C)}(D)\\
&\mathrm{NCenv}_{(C)}(D_1;\ D_2) && = \mathrm{NCenv}_{\mathrm{NCenv}_{(C)}(D_1)}(D_2)
\end{aligned}
$$

Figure 18: Function NCenv—Create an auxiliary representation structure for mACG-Modelica block declarations.

$$
\begin{array}{ll}
\mathrm{NBstrip}(b, [p_1 : t_1 : e_1; \\
\quad \ldots;\ p_n : t_n : e_n], m, r, q) & = (b, [p_1 : t_1;\ \ldots;\ p_n : t_n], m, r, q) \\
\mathrm{NB}_{(C)}(b, p, [m_1;\ \ldots;\ m_k], & = \text{let } B_{N_0} = \mathrm{NBstrip}(b, p, [m_1;\ \ldots;\ m_k], \\
\quad r, [q_1;\ \ldots;\ q_l]) & \qquad r, [q_1;\ \ldots;\ q_l]) \text{ in} \\
& \quad \text{let } B_{N_1} = \mathrm{NBm}_{(C, B_{N_0})}(m_1) \text{ in} \\
& \qquad \ldots\ B_{N_k} = \mathrm{NBm}_{(C, B_{N_{k-1}})}(m_k) \text{ in} \\
& \quad \text{let } B_{N_{k+1}} = \mathrm{NBq}_{(C, B_{N_k})}(q_1) \text{ in} \\
& \qquad \ldots\ B_{N_{k+l}} = \mathrm{NBq}_{(C, B_{N_{k+l-1}})}(q_l) \text{ in} \\
& \quad \text{let } (c_1 : t_{c_1}, \_), \ldots, (c_n : t_{c_n}, \_) = m_1, \ldots, m_k \text{ in} \\
& \quad \text{let } B_{N_{k+l+1}} = \mathrm{NBo}_{(C, B_{N_{k+l}})}(c_1 : t_{c_1}) \text{ in} \\
& \qquad \ldots\ B_{N_{k+l+n}} = \mathrm{NBo}_{(C, B_{N_{k+l+n-1}})}(c_n : t_{c_n}) \text{ in} \\
& \quad B_{N_{k+l+n}} \\
\mathrm{NormC}(C) & = \text{let } [(b_1, p_1, m_1, r_1, q_1);\ \ldots;\ (b_n, p_n, m_n, r_n, q_n)] = C \text{ in} \\
& \quad (\mathrm{NB}_{(C)}(b_1, p_1, m_1, r_1, q_1), \mathrm{NbInst}_{(C)}(b_1, p_1, m_1, r_1, q_1), \\
& \quad \ldots, \mathrm{NB}_{(C)}(b_n, p_n, m_n, r_n, q_n), \mathrm{NbInst}_{(C)}(b_n, p_n, m_n, r_n, q_n))
\end{array}
$$

Figure 19: Function NormC—Normalization transformation on the auxiliary structure C. Applies NB and NbInst (see Section B.3) to each block. Function NB uses NBstrip to remove parameter modifications, NBm (defined in Figure 20) to normalize component modifications, NBq (defined in Figure 21) to extract and normalize dot accesses appearing in RHS expressions and NBo (defined in Figure 22) to add "dummy" equations for component outputs that are not accessed.

which resemble function arguments. Therefore, the parameters of a block need to be set at the place there the block/node is instantiated and not within the block itself.

After that, function NB uses function NBm (defined in Figure 20) to normalize component modifications for every component instantiated in the current block. NBm first extracts all parameters from the component's block definition and introduces them as new (decorated) parameters in the block enclosing the component declaration. During that process it needs to be ensured that component modifications applied in the enclosing block replace parameter modifications in the component's block definition.

$$
\begin{aligned}
&\mathrm{NBm}_{\left(C+[(t_c,[p_{t_1}:t_{t_1}:e_{t_1};\ \ldots;\ p_{t_k}:t_{t_k}:e_{t_k}],\_,\_,\_)],\ (b_B,p_B,m_B+[(c:t_c,\_)],r_B,q_B)\right)}\\
&\quad\left(c:t_c,[p_{c_1}=e_{c_1};\ \ldots;\ p_{c_n}=e_{c_n}]\right)\\
&= \text{let } P_{t_p} = \{p_{t_1},\ldots,p_{t_k}\} \text{ in}\\
&\quad \text{let } P_{t_{pte}} = \{(p_{t_1},t_{t_1},e_{t_1}),\ldots,(p_{t_k},t_{t_k},e_{t_k})\} \text{ in}\\
&\quad \text{let } P_{c_p} = \{p_{c_1},\ldots,p_{c_j}\} \text{ in}\\
&\quad \text{let } P_{c_{pe}} = \{(p_{c_1},e_{c_1}),\ldots,(p_{c_n},e_{c_n})\} \text{ in}\\
&\quad \text{let } P_{c_{pte}} = \{(p_{c_1},t_{c_1},e_{c_1}),\ldots,(p_{c_n},t_{c_n},e_{c_n})\} =\\
&\qquad \{(p,t,e) \mid (p,e) \in P_{c_{pe}} \wedge (p,t,\_) \in P_{t_{pte}}\} \text{ in}\\
&\quad \text{let } P_{r_p} = P_{t_p} \setminus P_{c_p} \text{ in}\\
&\quad \text{let } P_{r_{pte}} = \{(p_{r_1},t_{r_1},e_{r_1}),\ldots,(p_{r_l},t_{r_l},e_{r_l})\} =\\
&\qquad \{(p,t,e) \mid p \in P_r \wedge (p,t,e) \in P_{t_{pte}}\} \text{ in}\\
&\quad \text{let } D = \mathrm{Vars}(p_B) \cup \mathrm{Vars}(m_B) \cup \mathrm{Vars}(r_B) \text{ in}\\
&\quad \text{let } p_{new} = [(\mathrm{dn}_{(D,c)}(p_{r_1}) : t_{r_1} : \mathrm{dn}_{(D,c)}(e_{r_1}));\\
&\qquad \ldots;\ (\mathrm{dn}_{(D,c)}(p_{r_l}) : t_{r_l} : \mathrm{dn}_{(D,c)}(e_{r_l}))]+\\
&\qquad [(\mathrm{dn}_{(D,c)}(p_{c_1}) : t_{c_1} : e_{c_1});\\
&\qquad \ldots;\ (\mathrm{dn}_{(D,c)}(p_{c_n}) : t_{c_n} : e_{c_n})] \text{ in}\\
&\quad \text{let } m_{new} = \big[(c:t_c,[p_{r_1} = \mathrm{dn}_{(D,c)}(p_{r_1});\\
&\qquad \ldots;\ p_{r_l} = \mathrm{dn}_{(D,c)}(p_{r_l})]+\\
&\qquad [p_{c_1} = \mathrm{dn}_{(D,c)}(p_{c_1});\\
&\qquad \ldots;\ p_{c_n} = \mathrm{dn}_{(D,c)}(p_{c_n})])\big] \text{ in}\\
&\quad (b_B, p_B + p_{new}, m_B + m_{new}, r_B, q_B)
\end{aligned}
$$

Figure 20: Function NBm—Normalize component modifications. All parameters from the modified component are extracted from the component's block definition and are introduced as fresh (decorated) parameters in the block enclosing the component declaration. During that introduction it is ensured that the components modification replace the parameter modifications from the component's block definition.

As a next step function NB applies function NBq (defined in Figure 21) to extract and normalize nested component dot accesses appearing in RHS expressions of equations. It traverses a RHS expressions e and searches for an occurrences of a component dot accesses $c.a$ in e. If it finds one, the component dot access is extracted and a *fresh* (decorated) variable $x_{ca} = \mathrm{dn}_{(D,c)}(a)$ is introduced and bound to $c.a$ in a new equation $x_{ca} = c.a$. The component dot access $c.a$ in e is then replaced by x_{ca}. In order to ensure that any component dot access $c.a$ is uniquely associated to *exactly one* equation $x_{ca} = c.a$ it is checked whether an equation association $x_{ca} = c.a$ already exists — if so, the variable x_{ca} is reused and no fresh variable is introduced.

And finally function NB applies function NBo (defined in Figure 22) to add "dummy" equations for component outputs that are not accessed in in the instantiating block. This is necessary, since after normalization it is required that *all* outputs of a component actually have a binding equation in a block that instantiates the component. Hence, function NBo first extracts all RHS component dot access variables that appear in the block equations for the respective component. Since component dot accesses have been previously normalized to $x = c.a$ by function NBq, the set Y_{B_o} of all RHS component dot access variables of component c is obtained by $Y_{B_o} = \{o \mid [\_ = c.o] \in q_B\}$. The remaining relations check whether there are outputs in c which have no binding in the current block context and create "dummy" variables and equations if such outputs exist.

$$
\begin{aligned}
&\mathrm{NBe}_{\left(C+[(t_c,\_,\_,(i,o+[a{:}t],l),\_)],\ \left(b_B,p_B,m_B+[c{:}t_c,m_c],(i_B,o_B,l_B),q_B\right)\right)}\left(c.a\right) \\
&\quad = \Big(x_{ca},\big(b_B,p_B,m_B+[c:t_c,m_c],\ (i_B,o_B,l_B+[t\ x_{ca}]),q_B+[x_{ca}=c.a]\big)\Big) \\
&\qquad \text{where } x_{ca}=\mathrm{dn}_{(D,c)}(a) \\
&\qquad \text{and } D=\mathrm{Vars}(p_B)\cup\mathrm{Vars}(m_B)\cup \mathrm{Vars}((i_B,o_B,l_B))\cup\{c\} \\
&\mathrm{NBe}_{\left(C+[(t_c,\_,\_,(i,o+[a{:}t],l),\_)],\ \left(b_B,p_B,m_B+[c{:}t_c,m_c],\ (i_B,o_B,l_B+[t\ x_{ca}]),q_B+[x_{ca}=c.a]\right)\right)}\left(c.a\right) \\
&\quad = \Big(x_{ca},\big(b_B,p_B,m_B+[c:t_c,m_c],\ (i_B,o_B,l_B+[t\ x_{ca}]),q_B+[x_{ca}=c.a]\big)\Big) \\
&\qquad \text{where } x_{ca}=\mathrm{dn}_{(D,c)}(a) \\
&\qquad \text{and } D=\mathrm{Vars}(p_B)\cup\mathrm{Vars}(m_B)\cup \mathrm{Vars}((i_B,o_B,l_B))\cup\{c\} \\
&\mathrm{NBe}_{(C,B)}(op(e_1,\ldots,e_n)) \\
&\quad = \mathrm{let}((a_1,\ldots,a_n),B_a)=\mathrm{NBeList}_{(C,B)}(e_1,\ldots,e_n)\ \text{in}\ (op(a_1,\ldots,a_n),B_a) \\
&\mathrm{NBe}_{(C,B)}(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) \\
&\quad = \mathrm{let}((a_1,a_2,a_3),B_a)=\mathrm{NBeList}_{(C,B)}(e_1,e_2,e_3)\ \text{in}\ (\textbf{if } a_1 \textbf{ then } a_2 \textbf{ else } a_3,B_a) \\
&\mathrm{NBe}_{(C,B)}(\textbf{previous}(e)) \\
&\quad = \mathrm{let}(a,B_a)=\mathrm{NBe}_{(C,B)}(e)\ \text{in}\ (\textbf{previous}(a),B_a) \\
&\mathrm{NBe}_{(C,B)}(e) \\
&\quad = (e,B) \text{ for the remaining forms of } e \\
&\mathrm{NBeList}_{(C,B)}(e_1,\ldots,e_n) \\
&\quad = \mathrm{let}\ (a_1,B_{a_1})=\mathrm{NBe}_{(C,B)}(e_1)\ \text{in} \\
&\qquad \ldots\mathrm{let}(a_n,B_{a_n})=\mathrm{NBe}_{(C,B_{a_{n-1}})}(e_n)\ \text{in} \\
&\qquad ((a_1,\ldots,a_n),B_{a_n}) \\
&\mathrm{NBq}_{(C,(b_B,p_B,m_B,r_B,q_B+[x=e]))}(x=e) \\
&\quad = \mathrm{let}\ (a,B_a)=\mathrm{NBe}_{(C,(b_B,p_B,m_B,r_B,q_B))}(e)\ \text{in} \\
&\qquad \mathrm{let}\ (\_,\_,\_,r_a,q_a)=B_a\ \text{in} \\
&\qquad (b_B,p_B,m_B,r_a,q_a+[x=a])
\end{aligned}
$$

Figure 21: Function NBq—Normalize RHS component dot access in equations. The RHS expression e is traversed by function NBe. An occurrence of a component dot accesses $c.a$ in e is extracted and a *fresh* (decorated) variable $x_{ca}=\mathrm{dn}_{(D,c)}(a)$ is introduced and bound to $c.a$ in a new equation $x_{ca}=c.a$. The component dot access $c.a$ in e is then replaced by x_{ca}. It is also checked whether $c.a$ already *is associated* to an equation $x_{ca}=c.a$ — in that case *no* fresh variable is introduced and x_{ca} is reused. This ensures that any component dot access $c.a$ is uniquely associated to exactly one equation $x_{ca}=c.a$.

$$
\begin{aligned}
&\mathrm{NBo}_{\left(C+[(t_c,\_,\_,(\_,[o_1:t_1;\ \ldots;o_k:t_k],\_),\_)],\ (b_B,p_B,m_B,(i_B,o_B,l_B),q_B)\right)}\left(c:t_c\right) \\
&\quad = \text{let } Y_{B_o} = \{o \mid [\_ = c.o] \in q_B\} \text{ in} \\
&\qquad \text{let } Y_{c_o} = \{o_1,\ldots,o_k\} \text{ in} \\
&\qquad \text{let } Y_{c_{ot}} = \{(o_1,t_1),\ldots,(o_k,t_k)\} \text{ in} \\
&\qquad \text{let } Y_{r_{ot}} = \{(o_{r_1},t_{r_1}),\ldots,(o_{r_n},t_{r_n})\} = \\
&\qquad\quad \{(o,t) \mid o \in Y_{c_o} \setminus Y_{B_o} \ \wedge\ (o,t) \in Y_{c_{ot}}\} \text{ in} \\
&\qquad \text{let } q_{new} = [\mathrm{dn}_c(o_{r_1}) = c.o_{r_1};\ \ldots;\mathrm{dn}_c(o_{r_n}) = c.o_{r_n}] \text{ in} \\
&\qquad \text{let } l_{new} = [t_{r_1}\ \mathrm{dn}_c(o_{r_1});\ \ldots;t_{r_n}\ \mathrm{dn}_c(o_{r_n})] \text{ in} \\
&\qquad (b_B,p_B,m_B,(i_B,o_B,l_B+l_{new}),q_B+q_{new})
\end{aligned}
$$

Figure 22: Function NBo—Add "dummy" equations for component outputs that are not accessed in the instantiating block. Normalization requires that *all* outputs of a component have bindings to equations in the block that instantiates the component.

B.3. Top-Level Block Instantiation

For any block declaration with name m in mACG-Modelica, function NbInst, defined in Figure 23, creates a fresh block with its name decorated by the string "*Inst*". The fresh block duplicates inputs and outputs of the instantiated block m and adds equations to connect its inputs and outputs to the inputs and outputs of m.

As a last step it calls function NBm (see Figure 20) on the freshly constructed block. That call ensures that parameter modifications in the block definition of m are applied at the created instance of m and that parameters in m that have no binding modification are introduced as parameters (likewise with no binding modifications) in the fresh block. Note that parameters that have no binding modification will be turned to additional node inputs in the subsequent translation to the SDFK language (see Section 3.6).

$$
\begin{aligned}
&\mathrm{NbInst}_{([(b_1,\_,\_,\_,\_);\ \ldots;\ (b_l,\_,\_,\_,\_)])}\left(b,p,m,([i_1:it_1;\ \ldots;\ i_k:it_k],\right. \\
&\qquad \left.[o_1:ot_1;\ \ldots;\ o_n:ot_n],l),q\right) \\
&\quad = \text{let } m_{Inst} = [(x_{new}:b,[])] \text{ in} \\
&\qquad \text{let } q_{Inst} = [x_{new}.i_1 = i_1;\ \ldots;\ x_{new}.i_k = i_k]+ \\
&\qquad\quad [o_1 = x_{new}.o_1;\ \ldots;\ o_l = x_{new}.o_n] \\
&\qquad \text{let } B = (\mathrm{dn}_{(\{b_1,\ldots,b_l\},Inst)}(b),[],m_{Inst}, \\
&\qquad\quad ([i_1:it_1;\ \ldots;\ i_k:it_k], \\
&\qquad\quad [o_1:ot_1;\ \ldots;\ o_n:ot_n],[it_1\ i_1;\ \ldots;\ it_k\ i_k]+ \\
&\qquad\qquad [ot_1\ o_1;\ \ldots;\ ot_n\ o_n]),q_{Inst}) \text{ in} \\
&\qquad \text{let } B_{Inst} = \mathrm{NBm}_{(C,B)}(m_{Inst}) \\
&\qquad B_{Inst} \\
&\qquad \text{where } x_{new} \notin \{i_1,\ldots,i_k\} \cup \{o_1,\ldots,o_n\}
\end{aligned}
$$

Figure 23: Function NbInst—Create block instance wrapper block.