A Modelica Sub- and Superset for Safety-Relevant Control Applications

Bernhard Thiele* Bernhard.Thiele@dlr.de
Stefan-Alexander Schneider† stefan-alexander.schneider@bmw.de
Pierre R. Mai‡ pmai@pmsf.de

* German Aerospace Center (DLR), Institute for Robotics and Mechatronics, Germany
† BMW AG, 80788 München, Germany
‡ PMSF IT Consulting, Marzling, Germany

Abstract

Fueled by the continuous, rapid progress within microelectronics, ever more intelligent and intricate functions are realized in mechatronic systems. To control the complexity associated with such designs, model-based control design methods are increasingly adapted in industry. Despite Modelica's obvious suitability to efficiently create appropriate high fidelity system models, the utilization of Modelica for developing discrete control functions is not yet wide spread. Adoption of Modelica for this task offers the potential for a seamless development methodology from the logical virtual model down to the technical system architecture, with corresponding traceability and maintainability benefits.

This contribution will specifically address this potential and propose a Modelica sub- and superset adequate for use within the development of safety-relevant control applications.

Keywords: embedded systems; functional safety; simulation; code generation; compiler; formal methods; validation; verification

1 Introduction

Model-based design has emerged as a standard development approach for the design of embedded systems. Its original promise to provide a more rapid and economic development process is confirmed in industrial practice [5].

More and more embedded software components are specified in models representing the so-called high-level application that is then automatically transformed (usually via embedded C-code) into binary code that is executable on the embedded target: Figure 1 shows a typical model-based development environment where the specification model is first designed using a next generation high-level, domain-oriented modeling tool. These specification models are typically enriched with implementation details and converted to so-called code generation models. A code generator converts the code generation model into C-code that a cross compiler translates to object code. The different object codes, including legacy and basic software code are then finally linked to a binary to be executed by an embedded target. This approach reduces the implementation effort and time, especially in iterative development workflows. Model-based development methods have a significant impact on the development process and the development environment with its tools.

With the increase of applications, and along with that of software size and complexity, model-based approaches have found their way into safety-relevant applications, especially in the aerospace and automotive domains. This evolution has thrust the safety impact of model-based development, especially with regard to high-level modeling and code generators, into the spotlight.

As described above, for practical purposes the process of the generation of the executable program from the model is mainly based on two development tools: the code generator and the cross compiler (including the cross linker). From an abstract point of view, this concatenation of these two compilers is again a (system) compiler, and can be treated by the same theory as a compiler that would translate directly from the code generation model to the executable code, see Figure 2. In the following all such translation tools will be denoted abstractly as *development tools*.

The generated C-code can be seen as intermediate representation of the model, because it is both output

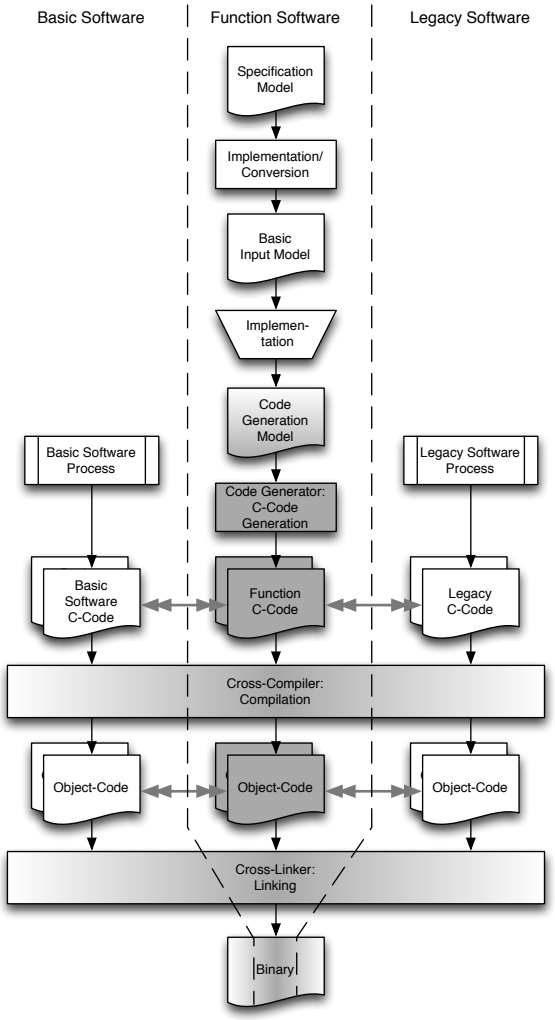

Figure 1: The generic build process for a model-based development toolchain with an automatic code generator (from [14]). The shaded parts indicate the tools and artifacts affected by what is later referred to as the *development tool* (code generator and cross compiler).

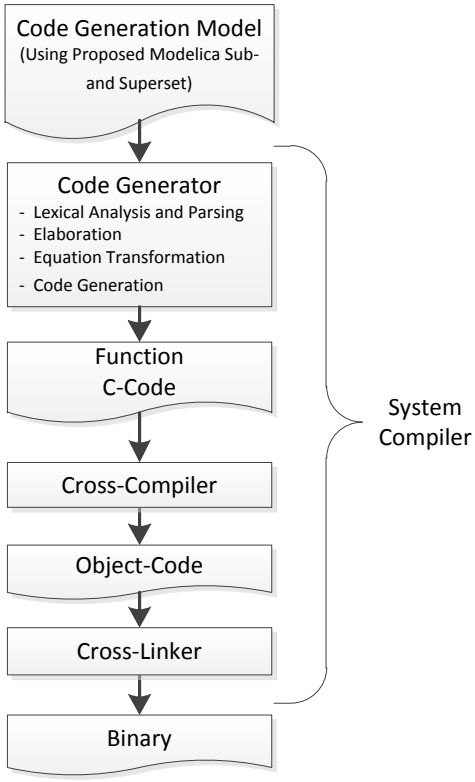

Figure 2: The concatenation of a code generator and a (cross) compiler can be treated as a (system) compiler that directly transforms from the code generation model to the executable (target) code. The Figure indicates that the system compiler is based on the proposed Modelica sub- and superset. Note that this system compilation process may be realized significantly different than a compilation process used in order to create *simulation* executables of models (see Section 4.2.5).

of the code generator as well as input to the cross compiler for the target. This perspective of the development tool is of central significance, because there is no need to perform any qualification activities on such internal representations as long as the C-code is only used as input to the cross compiler and is not further manipulated or used in any other activities that need to rely on the readability of the C-code. As a consequence no C-code reviews are needed in this case. This approach opens up the possibility to perform reviews on the model level. A main topic of this contribution is addressing the conditions that need to be established to allow such model level reviews.

Development tools may inject systematic faults into the executable program. Increasing the functional safety in this context means to minimize erroneous outputs of the development tools due to malfunctions and/or reliably detecting those erroneous outputs when they occur.

One method to gain confidence in a software development tool, is the validation of the software tool by a validation suite, which comprises a test suite specifically designed to exercise the development tool in ways that would provoke any systematic malfunctions.

In order to design a suitable modeling language, powerful development tools as well as an efficient and effective validation suite, it is important to understand precisely the role of translators and what the validation suite is intended to demonstrate. We therefore introduce some definitions in the context of a validation suite in Section 5, together with a discussion of the role of language structure and complexity.

For both the validation of the input language and

DOI 10.3384/ecp12076455

the transformation process, we have to cope with the curse of complexity. It is therefore of crucial importance to keep the language of the code generator models as simple and well-defined as possible, especially with regards to the number and complexity of basic constructs in the language, while also minimizing the number and complexity of performed transformation rules in the code generation process. This is especially true of transformation rules stemming from optimization rules. On the other hand the language so defined still has to be suitable for human consumption, so that the complexities of the code generation process are not just offloaded to the programmer.

Up to now the use of Modelica in embedded systems development is usually restricted as a modeling language for the physical plant dynamics. This can be attributed to:

1. A somewhat too limited expressiveness in modeling discrete controller functions.

2. The lack of a flexible, seamless development approach from the controller model comprising the *logical functions* to the *technical system architecture* (i.e., code running on the target platform).

3. And last but not least because safety-relevant software functions need means to achieve a high assurance level, which is not supported with current Modelica.

When discussing the use of Modelica in the context of control application, often advanced control concepts based on inverted plant dynamics are described [18], [19], [3]. Some Modelica tools are capable of automatically synthesising such controllers and generate code for them. This usually requires fairly sophisticated symbolic manipulation capabilities by the tool. For example, it may require to differentiate a subset of the equations, select appropriate states and solve the resulting system of differential and algebraic equations numerically.

However, the intrinsic complexity in the resulting control algorithms imposes an additional burden if the topmost design goal is in providing high assurance control systems. For the following discussion it is therefore assumed that the design trade-off is biased to prefer high assurance control over high performance control systems.

The aim of the paper is to study impacts of a safety-relevant development process (relying on validated tools) to high-level, domain-oriented modeling languages. In particular it proposes a sub- and superset of the modeling language Modelica suitable for such safety-relevant software development activities. To illustrate the development using the proposed language elements a showcase library (referred to as SAFEDISCRETECONTROL library) is presented and applied at an exemplary use case.

2 Development Roles

Model-based development is an established method in the development of safety-relevant products. As seen above, especially the two transformations code generation and cross compilation play an important role. To ensure that the benefits of this approach have full effect the working mode of the development tools needs to be well understood.

The intended software development process, and hence the development environment, has to provide a balance between controlled process steps and flexibility of user access: On the one hand, the user may not be restricted too much and must still have principal control over all development activities – too many restrictions reduce the acceptance and thus also the productivity and quality of the work. On the other hand, too few restrictions lead to error-prone development practices, and ultimately to preventable faults in the software.

Various stake-holders participate in the development in different roles with different requirements and expectations. A suitable Modelica sub- and superset will have to support at least the following roles in the development process:

Role 1 - Developer of the Embedded Control System. This role requires a sufficiently expressive modeling language with sound language elements with clear semantics to design and test the intended functionality.

Role 2 - Tool Developer. This role requires the precise definition of the input modeling language: There should be no unclear corner cases in the semantics. The language should be efficiently compilable to target code.

Role 3 - Reviewer for Functional Safety. This role requires a clear and unambiguous description of the functionality, including all semantically relevant modeling details in compact form for efficient reviews. It should be possible to determine coverage at the model level, and allow for tracing of requirements to the relevant model parts.

Role 4 - Tool Qualifier. This role requires a suffi-

ciently small number of modeling elements with clear semantics as well as clear, ideally highly localized composition rules, in order to establish a validation suite for the development tool. The boundaries of the development tools, i.e., input and output notations, have to be clearly defined. Automated processes should ideally be separately testable, to minimize complexity. For more details again see Section 5.

These different roles have partly coincident — semantic aspects — and partly contradictory — expressiveness of the language — requirements to the development process and with that to the development tools and their modeling languages.

3 Requirements for a Safety Oriented Modelica Sub- and Superset

In this section we study the requirements of the above introduced roles to modeling language, to the development enviroment and their tools, and a validation suite for safety-relevant developments.

We will see later in this contribution that these requirements can be met only by specific restrictions and extensions of the modeling domain language. This finally leads us to the introduction of a sub- and superset of Modelica based on these requirements, that is simple in order to facilitate high assurance designs, yet expressive enough to allow modeling of many control strategies of practical relevance.

The textual representation of the Modelica language defines the full semantics of a given model, therefore we will begin with the requirements to the textual representation.

3.1 Requirements to the Textual Representation

At a first glance, we argue that functional reviews should be done at the textual Modelica language level, since the language semantics are specified at the textual level and graphical representation may hide important details. However, of course the graphical level provides an abstraction that eases comprehension of the intended model semantics and is hence an extremely valuable supplementary to the textual review. We therefore describe in the following section additional requirements to the graphical representation designed to avoid any hidden important details in the graphical representation. It is then left to the reviewer and his preferences either to perform a textual or a graphical review.

We start with a coincident requirement for the roles:

Requirement 1 - Formally Sound Language Set. The language sub- and superset must be formally sound, so that validation and verification methods e.g. formal methods can be supported. Required by Roles 1, 2, 3, and 4

Requirement 2 - Minimum Expressiveness of the Language Set. The language sub- and superset must have enough expressiveness to allow the clear and concise specification of discrete open- and closed-loop control algorithms and their related support logic. Required by Roles 1 and 2.

Requirement 3 - Target Data Types and Operations. The language should provide a mechanism that allows to extend its data types and operations to support fundamental data types and operations available on the embedded target platform. Required by Roles 1 and 2.

Requirement 4 - Target Code Generation. The language should permit automatic generation of target platform C-code[1] that is: a) efficient, b) avoids unsafe constructs[2], c) is traceable[3], and d) integrates smoothly into embedded systems software architectures. Required by Roles 1, 2, and 1.

Requirement 5 - No Continuous-Time Dependencies. The language must not have dependencies to continuous-time system solver functionalities running in the background. Required mainly by Roles 1 and 2.

Requirement 6 - Compile Time Analysis. The language should allow compile time analysis of important properties in order to reject dubious programs (missing initial values, type checking, clock analysis, detection of cyclic definition that result in algebraic loops, etc.). Required mainly by Roles 1 and 3.

Requirement 7 - Modular Code Generation. The language must support modular code generation, i.e., within a model composed by connecting several blocks it must be possible to generate a transition function for each block definition and by composing them together produce the overall transition function. Required mainly by Roles 1 and 3.

[1] Automatic C-code generation is stipulated, since C-code is the most popular language for targeting embedded systems and (certifiable) compilers are available. However direct-to-binary code generators are not precluded by this, and similar though not identical concerns arise for those cases.

[2] For example by conforming to coding standards like MISRA AC AGC [17].

[3] Given a fragment of the automatically generated C-code it must be possible to trace it back to the model elements that caused its generation.

DOI 10.3384/ecp12076455

Requirement 8 - Modular Initialization. As a consequence of Requirement 7 also initialization of (state) variables must be supported in a modular manner, i.e., initial values of (state) variables in modular blocks deduced during compile time analysis must not depend on the environment enclosing the block. Furthermore, if initial values can not be *uniquely* determined from the given constraints and set start values code generation shall abort with an error message. Required mainly by Roles 1, 3 and 4.

Requirement 9 - Tangible Fixation of Automatically Deduced Properties. In order to ensure reproducibility of code generation and reviewability[4], it must be possible to fixate all properties of a model that influence code generation in a tangible, reviewable form. In particular it must be possible to fixate initial values that are automatically deduced by a tool, so that code generation will always use the fixated values instead of recalculating those values on the fly at the time of code generation. Required mainly by Role 3 and 4.

Requirement 10 - Manual Block Scheduling. Manual scheduling of block execution (as opposed to scheduling based on automatic causality analysis) must be possible on an optional basis. Required mainly by Role 1.

The following requirement is mainly motivated by the role of a tool qualifier and typically holds the most potential for discussion with the other roles, especially with the role developer:

Requirement 11 - Restricted Language Scope. To ease tool validation the language should be as simple and clear as possible. This shall be achieved by restricting the scope of the Modelica language to a (preferably small) *sub- and superset* relevant for the addressed problem domain, i.e, suitable for the implementation of the blocks in the SAFEDISCRETECONTROL library. Particularly, simplicity and clarity of the language sub- and superset is to be preferred over feature richness. Required mainly by Role 4.

As already mentioned above, in the following section we describe additional requirements to an optional graphical representation in order to perform a fully equivalent review on the graphical representation.

[4]Note that this requirement also enables separate validation of code generator and property-deduction code, since the fully fixated model provides the checkable interface between both processes.

3.2 Additional Requirements to the Graphical Representation

In computer science, semantics of a textual or graphical language refers to the meaning of programs written in it. Although the semantics of Modelica are described on a textual language level, Modelica provides standardized annotations for the graphical representation of models [10].

The main idea is that a *library* developer uses the textual Modelica language to code basic functionalities in components that are annotated with a graphical illustration, while an *application/model* developer (library user) works on a graphical level by just dragging, dropping and connecting the library components in order to compose the intended functionality.

How can we now avoid that not obvious or even hidden details in the graphical representation prevent the reviewers from performing an efficient and effective graphical review? In this section we therefore formulate *additional requirements to the graphical representation* that enable both developers (Role 1) and reviewers (Role 3) *to entirely work at a graphical level*.

Within Section 4.4 a conceptual library design (denoted SAFEDISCRETECONTROL library) is briefly presented that complies to the requirements stated in this section. In combination with adherence to the rules formulated in Section 4.3.2 the usage of such a library could then enable both, developers and reviewers, to entirely work at a graphical level.

We start with the graphical pendant to the textual requirement 1:

Requirement 12 - Intuitive Block Semantics. Blocks from SAFEDISCRETECONTROL and their compositions should not exhibit any behaviour which would be deemed surprising or non-obvious by a domain expert. Required by Roles 1, 2, 3, and 4.

The restriction on blocks reflects the textual requirement 11:

Requirement 13 - Restricted Set of Allowed Blocks. A high-level application model is only allowed to be composed from a set of thoroughly tested and validated basic blocks defined in the SAFEDISCRETECONTROL library. Required mainly by Roles 3 and 4.

Requirement 14 - Data Flow Semantic. Block diagrams with data flow semantic are used at the graphical level. Required by Roles 1, 2, 3, and 4.

Requirement 15 - Graphical Level Code Reviews. Code reviews of models should be feasible as far as possible at the graphical level. Consequently, any se-

mantics associated with the blocks from SAFEDISCRETECONTROL and their compositions should be completely evident by inspecting the graphical diagram layer, i.e., apart from clearly marked exceptional cases there should be no cases where the semantics of a model is not entirely and uniquely understandable from inspection of the block diagram. Required mainly by Role 3.

Requirement 16 - Block Testability. Any block within SAFEDISCRETECONTROL must be designed, so that extensive testing of the block is easily possible, i.e., simple, lean designs are preferred over sophisticated, complex designs. Note that this also implies that when balancing modeling comfort of blocks against simplicity, the bias is towards simplicity. Required mainly by Roles 3 and 4.

Requirement 17 - Composition Testability. Compositions of blocks from SAFEDISCRETECONTROL must again result in a block that is suitable for extensive testing, e.g., by restricting the number of inputs and outputs that are allowed for a block. Required mainly by Role 3 and 4.

Requirement 18 - Traceability. Compositions of blocks from SAFEDISCRETECONTROL must result in generated code that can be traced back to the blocks in the model, in order to easily perform, e.g., code coverage analysis on the target level but mirror back the results onto the model level. Required mainly by Role 3.

4 Proposal for a Safety Oriented Modelica Sub- and Superset

The aim of this section is to introduce a sub- and superset of Modelica that is simple in order to facilitate high assurance designs, yet expressive enough to allow modeling of many control strategies of practical relevance.

4.1 Terminology

The following list defines some key terms used subsequently.

Basic Blocks Blocks that have no inner instance of other blocks are subsequently referred to as *basic blocks*. These blocks may only contain parameters, connectors and (textual) equations.

Composite Blocks Blocks that are (graphically) composed from other blocks are subsequently referred to as *composite blocks*. These blocks may only be composed from other blocks connected by `connect(..,..)` equations. Therefore they do not contain any other textual equations.

Clocks Clocks provide an activation signal or *clock signal* used for synchronous scheduling of a set of equations activated by that clock signal. They recently entered the Modelica language standard.

Clock Blocks Special basic blocks containing clocks that provide a clock signal are subsequently referred to as *clock blocks*.

Atomic Blocks Blocks which are executed as a single unit (akin to a function call with input and output arguments) are referred to as *atomic blocks*.

4.2 Superset: Language Extension Proposal

To meet the requirements defined in Section 3 it is not sufficient to solely *restrict* the language elements to a *subset* of the current Modelica 3.3 standard specification. In addition it is necessary to *extend* the language elements, effectively forming a *superset* of the current language.

This section proposes several language extensions by

1. Explaining the perceived limitation of Modelica 3.3 that needs to be addressed.
2. Proposing a language extension that overcomes the limitation.

4.2.1 Data Types Extension

Modelica 3.3 [10, Section 12.9] specifies the following data type mapping to C:

Modelica data type	Default mapping to C
Real	double
Integer	int
Boolean	int
String	const char*
Enumeration	int

Embedded processors often need finer control about the used data type[5]. Again it is necessary to make a trade-off between feature completeness and validation costs. Validation effort will raise for every supported

[5] E.g., for increased memory efficiency or because the embedded system simply doesn't provide efficient support for that data type, e.g., an embedded system with a FPU (Floating Point Unit) that supports only single precision floating point arithmetic.

DOI 10.3384/ecp12076455

data type. In effect it needs to be checked whether the savings gained by supporting a particular data type (e.g., because a cheaper electronic control unit (ECU) can be used) outweighs the additional costs in (tool) validation.

The following section will propose a rather general mechanism to extend the standard Modelica data types with more low-level hardware encoding information. Note that although the SAFEDISCRETECONTROL library presented in Section 4.4 only supports a subset of the listed data types, the extension to additional data types is straight forward. However, the associated additional validation effort for any additional supported data type is considerable.

4.2.2 Proposal for Data Type Extension

The relevant part of the Modelica specification defining the basic data types is [10, Section 4.8]. The notation in the specification is adapted to extend the definition of the `Real`, `Integer` and `Boolean` data types[6]. The following predefined enumeration types are used for the definition.

```
type PlatformType = enumeration(
UInt8 "8-bit unsigned integer",
SInt8 "8-bit signed integer",
UInt16 "16-bit unsigned integer",
SInt16 "16-bit signed integer",
UInt32 "32-bit unsigned integer",
SInt32 "32-bit signed integer"
);

type PlatformRealType = enumeration(
Float "IEEE 754 single precision
       floating type",
Double "IEEE 754 double precision
        floating type",
);
```

Using the definitions above the predefined types of Modelica are extended with the additional attribute `platformType`. The rationale for not supporting an attribute is given in the corresponding footnote. Note that the types are defined with Modelica syntax although they are predefined, fundamental data types in Modelica.

```
type Real
  RealType value; /* Accessed
    without dot-notation */
  parameter StringType quantity;[6]
  parameter StringType unit;
  parameter StringType displayUnit;[6]
  parameter RealType min=-Inf, max=+Inf;
  parameter RealType start = NaN;[7]
  parameter BooleanType fixed;[8]
  parameter RealType nominal;[9]
  parameter StateSelect stateSelect;[10]
  parameter PlatformRealType platformType
    = PlatformRealType.Double;
end Real;


type Integer
  IntegerType value; /* Accessed
    without dot-notation */
  parameter StringType quantity;[6]
  parameter IntegerType min=-Inf, max=+Inf;
  parameter IntegerType start = +Inf;[7]
  parameter BooleanType fixed;[8]
  parameter PlatformType platformType
    = PlatformType.Sint32;
end Integer;

type Boolean
  BooleanType value; /* Accessed
    without dot-notation */
  parameter StringType quantity;[6]
  parameter BooleanType start = false;[7]
  parameter BooleanType fixed;[8]
  parameter PlatformType platformType
    = PlatformType.Sint32;
end Boolean;
```

Note that the values of the variables may not be directly manipulated in memory and consequently there are no access routines.

4.2.3 Activation of Discrete-time Equations in Modelica

Before the recently released Modelica 3.3 language standard the activation of discrete-time equations was either due to *time events* or *state events*.

[6] In this work the dedicated support of fixed-point arithmetic is not (yet) considered. Note that if fixed-point arithmetic is required it is possible (though not convenient) to use the proposed Modelica language extensions to implement and validate custom basic blocks that provide the required functionality.

[6] Omitted for the sake of language simplification (Requirement 11).

[7] If no start value is given, the start value is deduced (in compliance with Requirement 8) during compile time analysis.

[8] The Attribute "`fixed`" cannot be applied on clocked discrete-time variables. It is true for variables to which the `previous()` operator is applied, otherwise false [10, Section 16.9].

[9] Nominal values are only useful in the context of numerical solvers. They have no relevance in our targeted discrete applications.

[10] Only useful for solving (continuous) differential equation systems.

Time events are scheduled by the solver along a global simulation time line. Time is a (physical) real number (as opposed to the principle of *multi-form time*[12] adapted by synchronous languages) that steadily increases during execution (simulation) of a Modelica model. The global simulation time can be accessed anywhere in a Modelica model by the built-in variable `time`[13].

State events are detected by the solver if a variable (controlled by the solver) experiences a zero-crossing.

The event handling approach of Modelica works well for *simulating* a plethora of hybrid system models, but it has shortcomings if embedded systems code shall be generated from a Modelica model. The prerequisite that an "omniscient" solver "running in the background" detects and schedules events in order to activate the evaluation of a set of equations impedes straightforward integration into external environments.

In order to allow smooth integration of code generated from Modelica into embedded systems software projects, Modelica needs to allow *external* code to simply cause the evaluation of a set of (discrete-time) Modelica equations (without the internal participation of a hybrid systems solver that tries to detect whether the equations shall be evaluated or not). Nikoukhah and Furic [11] provide a notable discussion about the missing feature of external activation in context of using Modelica models within the Scicos[14] modeling environment which similarly applies to using Modelica models in embedded systems software projects.

To allow external activation of Modelica models Nikoukhah and Furic propose in [11] to add an `Event` type to the Modelica language and discuss the elements and semantics needed to integrate that new type in a general and backwards compatible way[15].

The latest Modelica 3.3 language standard added *synchronous language elements* particularly targeted at the implementation of control systems [10, Chapter 16, *Synchronous Language Elements*]. They add *clock activation* as a third way of activating discrete-time equations that largely solves the hitherto criticised deficiencies.

Another notable advantage of clock activation in comparison to activation through the traditional state and time events mechanism is the support of *clock inference*. It is no longer necessary to explicitly propagate an event to all (`block`) instances that contain equations that should be activated by that event. The property of a variable that is explicitly *associated with a clock* is propagated to other variables that are related with that variable through equation relations. The usage of variables associated with different clocks within the same expression requires special *clock conversion operators*, otherwise it is a model error. This increases the modeling comfort and protects against modeling errors related to unconscious combination of signals sampled at different points in time.

The following section will use a subset of the synchronous language elements as a base to realize a mechanism that, sloppily speaking, allows to call `block`s as functions. On the one hand the proposal will restrict the allowed set of synchronous language elements to a subset (for language simplification reasons), on the other hand it will introduce a slight extension in order to satisfy two use cases:

1. Allow smooth integration of generated code into external environments, e.g., AUTOSAR authoring environments.

2. Allow manual scheduling of block execution as depicted in Figure 3.

The requirement to allow manual scheduling of block activation might appear strange, since a program can figure out the "correct" activation sequence easily from the data flow. However interaction with external software components, as well as execution time and real-time requirements, can place additional restrictions on the activation sequence that can not be determined by data flow only. Therefore, manual scheduling can be necessary. Additionally, if discussing a safety-relevant design with authorities it can often be beneficial to document that a human being has thought of the correct activation sequence rather than a machine.

4.2.4 Proposal for Atomic and Priority Based Activation

Conceptually, the *atomic block* definition in Section 4.1 yields the semantic depicted in Figure 4.

[12] The multi-form time principle states that any sequence of events can be considered as a time scale for the reactive system that perceives these events.

[13] Note that at the beginning of Section 4.2 it is stated that the built-in variable `time` is not supported for the SAFEDISCRETECONTROL library.

[14] Scicos is a graphical dynamical system modeler and simulator with support for continuous and discrete time models (http://www.scicos.org/).

[15] An early draft version of this document actually proposed an activation mechanism inspired by the proposal of Nikoukhah and Furic.

DOI 10.3384/ecp12076455

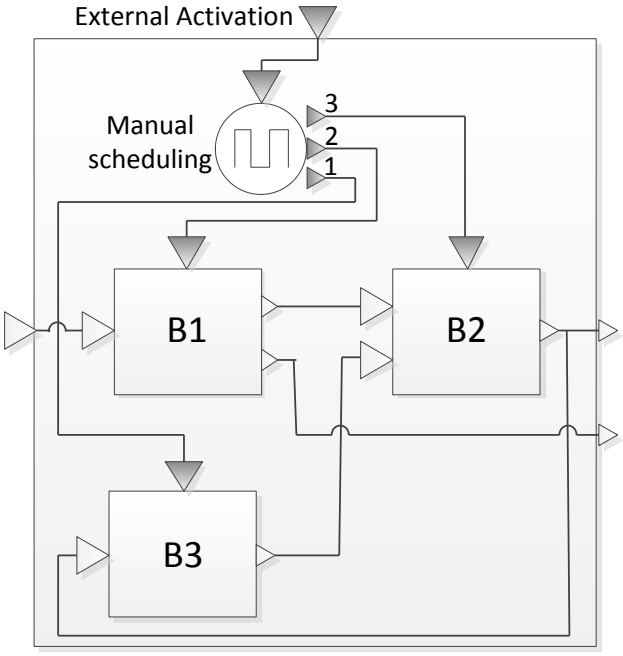

Figure 3: Manual scheduling of blocks.

Since an atomic block is executed as single unit it is required that all equations within the block must be activated from the same clock signal. However, it is possible that a block internally subsamples a clock signal and provides it as an output. Note that due to clock inference it is not necessary to provide an explicit clock input to every atomic block within a diagram as long as a unique clock can be inferred for it.

Atomic Blocks Currently there is no language support for treating a block as atomic according to our definition. To mitigate that deficiency the prefix **`atomic`** is proposed. The atomicity of a block is defined at the instance declaration.

```
atomic BlockModule a;
```

Akin to the execution of algorithms in Modelica models, an atomic block can be conceptually viewed as an atomic vector-equation (potentially with internal state) that maps its inputs to outputs, e.g.,

```
(out1,out2,...) = BlockModule(in1,in2,...);
```

Figure 5 further illustrates the difference between conventional and atomic block semantics.

Clock Priorities To allow manual scheduling of blocks it is proposed to extend the **`subSample`**`(u, factor)` operator with an (optional) additional integer argument denoting the priority of a clock:

```
subSample(u, factor, priority)
```

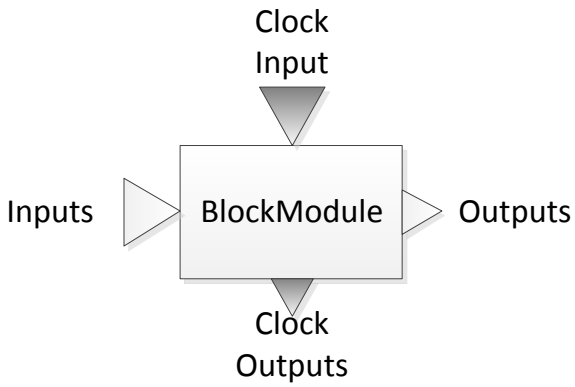

Figure 4: Conceptual atomic MIMO-block with data inputs and outputs, as well as one clock signal input (for activating the block) and (potential) several subsampled clock signal outputs.

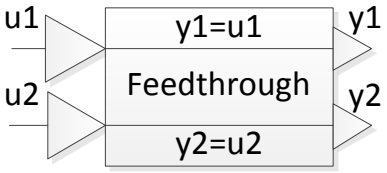

Figure 5: Trivial feedthrough block. *If declared atomic*, it is required that input signals `u1` and `u2` are active *at the same clock ticks* allowing to conceptually transform the block to a (periodically called) function `(y1,y2) = Feedthrough(u1,u2)` (the block hierarchy is maintained at execution level). If *not declared atomic*, `u1=y1` and `u2=y2` are allowed to be *active at completely unrelated points in time* (the block hierarchy is flattened, see Section 4.2.5)!

Values assigned to `priority` must be positive (including zero), lower values indicate a higher priority. If omitted, the argument defaults to "`priority = 0`". The priorities are always *relative* to the source clock signal.

An example implementation for the scheduler block in Figure 3 is given below.

```
block Scheduler
input Clock clk;
output Clock clk1;
output Clock clk2;
output Clock clk3;
equation
  clk1 = subSample(clk,1,1);
  clk2 = subSample(clk,1,2);
  clk3 = subSample(clk,1,3);
end Scheduler;
```

The semantics is that the equations activated by a

higher priority clock must be executed first[16]. Note that due to the relative nature of clock priorities stated above, it is not possible that a clock has a higher absolute priority than the input clock. Furthermore, in order to uniquely associate every variable with one clock a **subSample**(u)[17] operator needs to be present between the block connection shown in Figure 3. That has not been depicted to keep the diagram well-arranged.

Direct Block Activation The following language extension proposal is not essential to meet the requirements, however it supports more clearly arranged models.

The current language standard does not allow to directly assign a clock to a block with the semantics that all equations and variables in the scope of that block are marked to be part of the same base-clock partition. Therefore, the designated way to execute clocked equations within a controller block is by providing the clocking information at the inputs of that block and rely on clock inference. If the block needs several inputs that may result in a diagram like depicted in Figure 6.

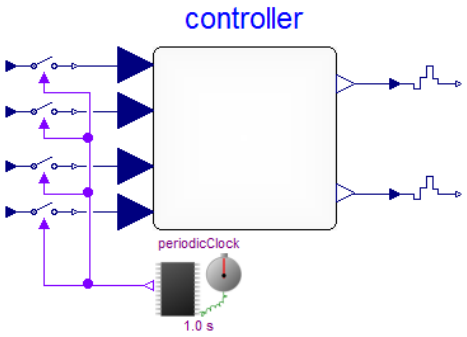

Figure 6: The designated way to execute clocked equations within a controller block using Modelica 3.3 is by providing the clocking information at the inputs of that block (by using the `sample(u, c)` operator and passing in the clock by the second argument) and rely on clock inference to forward the clocking information.

A tentative more explicit mechanism would be to introduce a built-in attribute "`clock`" for `block` classes that allows to assign a clock to a block. The semantics would be, that any variable and equation enclosed by the block would be associated with the assigned clock, e.g.,

```
block Controller
  input Real u;
  output Real y;
equation
  u = y;
end Controller;

model Environment
  Clock clk = Clock(0.5);
  Real s = sin(time);
  // associate c.u and c.y with clk
  Controller c(clock=clk);
equation
  c.u = sample(s);
end Environment;
```

However, in order to keep extensions to the official Modelica standard at a minimum this extension is not part of the proposed Modelica superset.

Clock Blocks and Interaction with the Physical Environment Clock blocks provide clock signals. They are *source* blocks, since they need no input to provide a clock signal as output. In order to convert between continuous-time (physical environment) and clocked discrete-time signals the operators **sample**(u) (continuous-time variable u converted to discrete-time) and **hold**(u) (zero-order hold conversion of discrete-time variable u to continuous-time) are needed.

Note that according to Section 4.3, Rule 9 clock blocks as well as the conversion operators may not be part of the high-level application intended for code generation. They are elements needed to *simulate* the execution of the high-level applications within the simulation tool (depicted in Figure 7). Or formulated differently, they are idealized models of the environment that will execute the high-level application running on the ECU, e.g., a periodic scheduler of an operating system that activates the high-level application task.

Since clock blocks, sample(u) and hold(u) reside outside the high-level application they are not part of the Modelica sub- and superset proposed for code generation!

The Modelica Specification [10, Section 16.3] defines several overloaded **Clock**(..) constructors. A simple clock source block is modeled below.

```
block PeriodicClock
parameter Real sampleRate = 0.1;
output Clock y = Clock(sampleRate);
end PeriodicClock;
```

[16]The execution order of equations which are activated by clocks with equal priority is determined by the (standard) causality analysis algorithms of the Modelica tool.

[17]If the arguments "`factor`" and "`priority`" are not provided or zero, they are inferred.

DOI 10.3384/ecp12076455

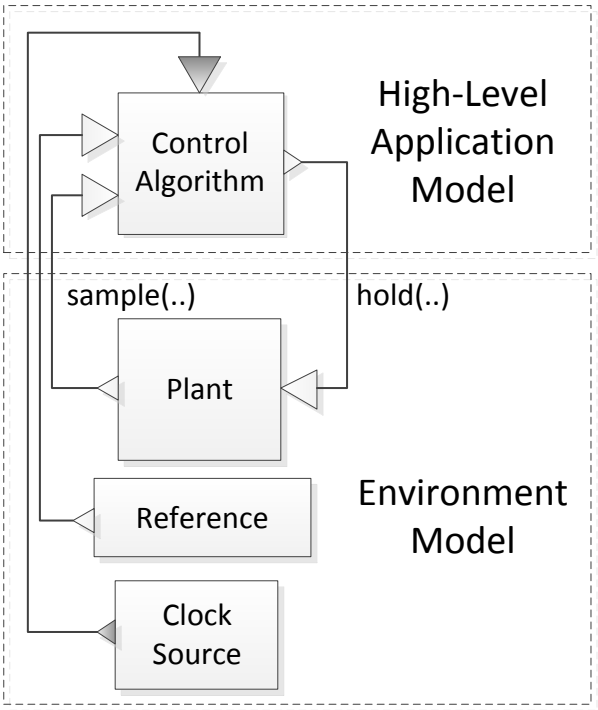

Figure 7: Simulation of a high-level application model using a clock block to model the execution of the application by its environment, e.g., by an operating system scheduler.

Calling a Block as a Function Combining the proposed priority based clock activation with modular code generation (Section 4.2.6) fulfils the requirement to "call blocks as functions" (stated in Section 4.2.3).

To exemplify, assume `B1`, `B2` and `B3` from Figure 3 have the annotation `Inline=false` and the manual scheduler is modeled by assigning appropriate priorities to the clock signals. A tool could generate following (conceptual) C-code.

```
double EnclosingBlock_B2_y = 0;

void EnclosingBlock(double u,
       double* y1, double* y2) {
  double B1_y1, B1_y2;
  double B2_y, B3_y;
  B3(EnclosingBlock_B2_y, &B3_y);
  B1(u, &B1_y1, &B1_y2);
  B2(B1_y1, B3_y1,
    &EnclosingBlock_B2_y);
  *y1 = EnclosingBlock_B2_y;
  *y2 = B1_y2;
}
```

4.2.5 Typical Modelica Code Generation

The typical Modelica code generation process differs significantly from the automatic target code generation intended for safety related applications. This section will give a short overview over the typical code generation process in order to better appreciate and understand the proposal for simplified and modular code generation presented in Section 4.2.6.

Compiling Modelica code usually involves substantial code transformation. Figure 8 gives an overview of the compilation and simulation process as described by Broman [4, p. 29].

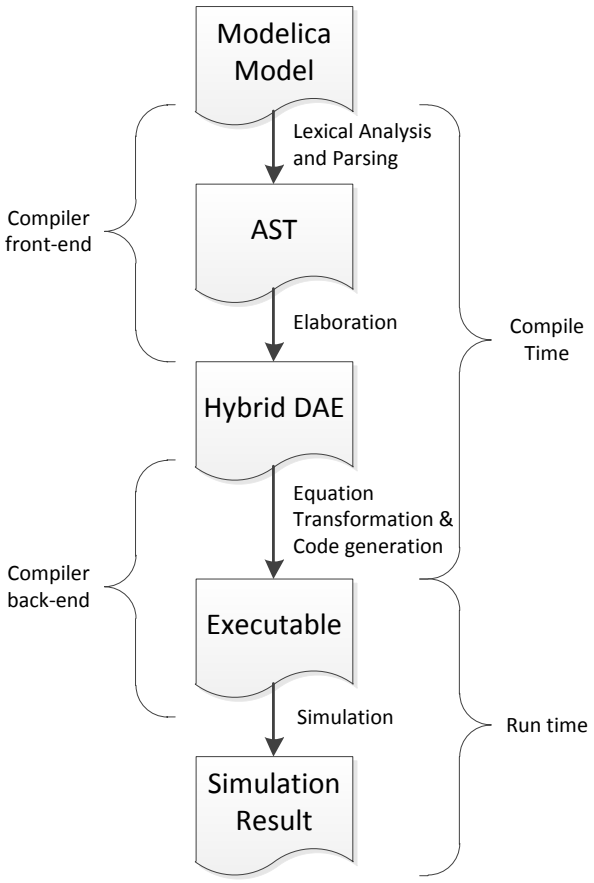

Figure 8: Outline of a typical compilation and simulation process for a Modelica language tool [4, p. 29].

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 DAE (consisting of variable declarations, equations from equations sections, algorithm sections, and `when`-clauses for triggering discrete-time behaviour).

Equation Transformation This step encompasses transforming and manipulating the equation sys-

tem 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. While 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 inhibiting modularisation and traceability at the C-code level. That conflicts with Requirements 7, 8 and 18.

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

Therefore, the compilation process for *simulation* may be significant different to the *target code compilation* process depicted in Figure 2. 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. In particular the target code generator depicted in Figure 2 is only required to understand the sub- and superset of the Modelica language intended for (discrete) software application models.

4.2.6 Proposal for Simplified and Modular Code Generation

In the context of safety related function development it is proposed to

1. Use simplified and transparent equation transformation algorithms for block diagrams.

2. Support modular code generation for blocks.

Simplified Transformation Algorithms The complexity in the compilation process of Modelica models is mainly due to acausal[18], physical modeling. Transforming typical physical models in a form that can be efficiently solved by a numerical solver requires advanced symbolic manipulation techniques.

However, the proposed Modelica subset allows for a hugely simplified compilation. Recall, that compared to full Modelica, the following restrictions apply to the block diagram subset of Modelica proposed in this paper:

- Modelica block diagrams allow only *causal* connections between blocks.

- Only difference equations are permitted by the proposed language subset (the `der(..)` operator is not available in the proposed language subset).

Transforming these equations to (causal) serial code is completely feasible by resorting to well known, published algorithms[19] without needing additional expert knowledge to perform challenging tasks like index reduction (which is needed for almost all physical Modelica models of practical relevancy). Therefore, to increase the transparency of the transformation process it is proposed to use plain and open transformation algorithms suitable for the targeted Modelica subset.

Modular Code Generation Modular, or separate, code generation for blocks improves *traceability* within a code generation process, a key requirement when developing safety related functions [2, 1, p. 75].

The aim of modular code generation is to produce a transition function for each block definition and compose them together to produce the main transition function. However, flattening each block and manipulating the corresponding equation system on the global

[18] The term *acausal* in Modelica is somehow similar to what is referred in computer science as *descriptive*.

[19] For example by employing basic "Modelica" algorithms for causalization of an equation system into a block lower triangular form [12, 15, 6].

DOI 10.3384/ecp12076455

level usually allows to generate a better optimized, more efficient code. Consequently, a *trade-off* between *efficiency* and *traceability* is required [1, p. 75].

Instead of collapsing the instance hierarchy, as typically done within the **Elaboration phase**, it is proposed to provide an option that preserves the modularity of an instantiated `block`.

The Modelica specification already knows of annotations that can influence code generation [10, Section 18.3]:

```
code_annotation:
  annotation"(" codeGenerationFlag "="
    { false | true } ")"

codeGenerationFlag:
  "Evaluate" | "HideResult" | "Inline" |
  "LateInline" | "GenerateEvents"
```

Within the specification the effect of the flag `"Inline"` is limited to function declarations. We propose to extend that scope in order to use the flag to annotate block instances and block declarations that have been declared "`atomic`". An annotation at the block instance takes precedence over an annotation at the block declaration. If the flag is not explicitly set it defaults to `Inline=false`.

Therefore, for the example from Figure 9 the declaration to enforce separate code generation for the block instance writes:

```
atomic BlockModule blockModule
  annotation(Inline=false);
```

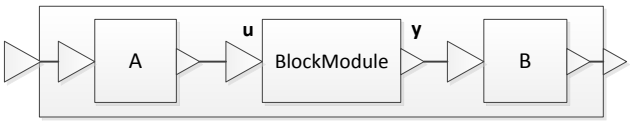

Figure 9: For every block it must be possible to *optionally* state whether the modularity of the block instance used in a composition is to be preserved at source code level.

For the `BlockModule` example from Figure 9 a C-function with suitable input and output data structures could be generated in way similar to

```
void BlockModule(inBlockModule_u *u,
                 outBlockModule_y *y);
```

The internal state variables of the block (if any) could be either part of the output data structure, or alternatively could be provided as a third argument to the function.

A common alternative approach is to use global variables with a suitable naming scheme to avoid variable clashes for input, output, and state variables which results in functions with a void signature, e.g.,

```
void BlockModule(void);
```

If a suitable communication mechanism exists the code may also instead of directly accessing the variables use the communication interfaces provided by the run-time environment, e.g.,

```
void BlockModule(void) {
  inBlockModule_u u =
    get_inBlockModule_u();
  outBlockModule_y y;
  /* .. */
  set_outBlockModule_y(y);
}
```

Clarity may improve if for initialization or reset of state variables an additional, dedicated function is generated.

Additional Remarks on Code Generation A detailed discussion about automatic target code generation by Modelica tools is out of scope of this article. With no doubt the user needs to have more influence on the code generation than the options given by the proposal above. Customized control over some aspects of the code generation might be provided within the model in form of Modelica annotations (standardized or tool specific) or also at completely different locations and in different forms.

4.3 Subset: Reducing Language Complexity

The practice of defining *modeling* or *coding standards* for safety-relevant software projects is well established[20]. This section proposes several rules for the development of safety-relevant control applications, akin to a modeling standard, in order to reduce the complexity of the language.

The rules *restrict* the allowed language elements, and hence define a *language subset*.

4.3.1 Textual Language Rules

To meet the requirements defined in Section 3 it is not sufficient to solely *restrict* the language elements to a *subset* of the current Modelica 3.3 standard specification. In addition it is necessary to *extend* the language

[20] An example is the functional safety standard IEC 61508-3, in which the use of coding standards is highly recommended for SIL 3 and above [9]. Several coding/modeling guides published by The Motor Industry Software Reliability Association (MISRA) provide standards specifically targeted (but not limited) at the automotive industry, the most famous one being MISRA C [16].

elements, effectively forming a *superset* of the current language.

A language extension capable of meeting the requirements is proposed in Section 4.2. The following rule definitions require that this extension is available.

Rule 1 - Clocked Variables Exclusivity. Occurring variables and equations must be part of (discrete-time) clocked partitions. Note that this restriction implies that all allowed high-level applications have a purely time-discrete nature.
Rationale: Clocked variables and equations were introduced in Modelica 3.3 to provide improved support for implementation of (discrete-time) control systems.
Trace: Requirement 4, 5.

Rule 2 - Reduced Set of Keywords. Table 1 reproduces the keywords from the Modelica specification [10, Section 2.3.3]. Keywords that are not allowed in the proposed Modelica subset are stroked through. The semantics of the remaining elements are maintained appropriately[21].
Rationale: Language simplification.
Trace: Requirement 11.

Rule 3 - Reduced and Restricted Set of Operators. The available Modelica operators are slightly reduced (no `.*` `./` `.+` `.-`) and the arithmetic operators are restricted to scalar types (see Table 2).
Rationale: Language simplification.
Trace: Requirement 11.

Rule 4 - Reduced Set of Built-in Functions and Operators with Function Syntax. Table 3 specifies the subset of supported built-in functions and operators with function syntax defined in [10, Section 3.7 and Chapter 10, 16 and 17].
Rationale: Language simplification.
Trace: Requirement 11.

Rule 5 - Supported Data Types. The scalar data types `Boolean`, `Real` and `Integer` are fully supported and extended to support more fine grain control about the underlying hardware encoding in Section 4.2.1. `Enumeration` is not supported, the support of `String` is limited to parameter values and constants. Array support is limited. Most (overloaded) operators and built-in functions related to arrays are not supported (see Table 2 and 3).
Rationale: Language simplification, as well as increase of language expressiveness to satisfy common data type requirements from the embedded systems domain. Functionality provided by the array operators and built-in functions, e.g., scalar product, can be programmed by using the scalar operators and loops.
Trace: Requirement 4, 11.

Rule 6 - No Support of Built-in Variable `time`. The built-in variable `time` (see [10, Section 3.6.7]) is not supported.
Rationale: Physical time is a quantity of continuous system simulation and therefore not supported in the time-discrete language subset. If an absolute wall-clock time is needed in the application logic, it has to be passed in from the external environment as a (Real) input signal.
Trace: Requirement 5.

Table 1: Reduced set of allowed Modelica keywords.

algorithm	~~discrete~~	false	loop	record
and	~~each~~	~~final~~	~~model~~	~~pure~~
annotation	else	~~flow~~	not	~~redeclare~~
~~assert~~	elseif	for	~~operator~~	~~replaceable~~
block	~~elsewhen~~	function	or	~~return~~
~~break~~	~~encapsulated~~	if	~~outer~~	~~stream~~
~~class~~	end	~~import~~	output	then
connect	~~enumeration~~	~~impure~~	package	true
connector	equation	in	parameter	~~type~~
constant	~~expandable~~	~~initial~~	~~partial~~	when
~~constrainedby~~	~~extends~~	~~inner~~	protected	~~while~~
~~der~~	external	input	public	~~within~~

Table 2: Reduced set of allowed operators

Operator Group	Operator Syntax
postfix array index operator:	`[]`
postfix access operator:	`.`
postfix function call:	`funcName(..)`
exponentiation:	`^`
multiplicative[a]*:*	`* / ~~.*~~ ~~./~~`
additive[a]*:*	`+ - +expr -expr ~~.+~~ ~~.-~~`
relational:	`< <= > >= == <>`
unary negation:	`not expr`
logical and:	`and`
logical or:	`or`
array range:	`expr : expr` `expr : expr : expr`
conditional:	`if expr then expr else expr`
named argument:	`ident = expr`

[a] Note that contrary to [10, Section 3.4] the arithmetic operators `^ * / + -` are limited to operate on scalar types only and the elementwise operators `.* ./ .+ .-` are not available.

4.3.2 Additional Graphical Representation Rules

The following rules establish a modeling standard for the graphical representation of Modelica models, targeting the requirements formulated in Section 3.2, to

[21] Note that the reason for excluding a keyword is not because it would be unsafe to allow it. The reasons for excluding keywords is to reduce the complexity of the language as much as possible down to a set of (indispensable) core elements.

DOI 10.3384/ecp12076455

Table 3: Reduced set of built-in functions and built-in operators with function syntax

Numeric Functions and Conversion Functions	abs(v) sqrt(v) ~~String(..)~~	sign(v) Integer(e)
Event Triggering Mathematical Functions[a]	div(x,y) rem(x,y) floor(x)	mod(x,y) ceil(x) integer(x)
Built-in Mathematical Functions and External Built-in Functions	sin(x) tan(x) acos(x) atan2(x,y) cosh(x) exp(x) log10(x)	cos(x) asin(x) atan(x) sinh(x) tanh(x) log(x)
Derivative and Special Purpose Operators with Function Syntax	~~der(expr)~~ ~~cardinality(c)~~ ~~semiLinear(..)~~ ~~actualStream(v)~~ ~~getInstanceName()~~	~~delay(..)~~ ~~homotopy(..)~~ ~~inStream(v)~~ ~~spatialDistribution(..)~~
Event-Related Operators with Function Syntax	~~initial()~~ ~~noEvent(expr)~~ ~~sample(s,i)~~ ~~edge(b)~~ ~~reinit(x, expr)~~	~~terminal()~~ ~~smooth(p, expr)~~ ~~pre(y)~~ ~~change(v)~~
Synchronous Language Elements	Clock() previous(u) ~~hold(u)~~[b] superSample(..) ~~backSample(..)~~ interval(u)	~~Clock(..)~~[b] ~~sample(u, c)~~[b] subSample(..) ~~shiftSample(..)~~ ~~noClock(u)~~
State Machines[c]	~~transition(..)~~ ~~activeState(state)~~ ~~timeInState()~~	~~initialState(state)~~ ~~ticksInState()~~
Array Dimension and Size Functions	~~ndims(A)~~ ~~size(A)~~	size(A,i)
Dimensionality Conversion Functions	~~scalar(A)~~ ~~matrix(A)~~	~~vector(A)~~
Specialized Array Constructor Functions	~~identity(n)~~ ~~zeros(..)~~ ~~fill(..)~~	~~diagonal(v)~~ ~~ones(..)~~ ~~linspace(x1,x2,n)~~
Reduction Functions and Operators	~~min(..)~~ ~~sum(..)~~	~~max(..)~~ ~~product(..)~~
Matrix and Vector Algebra Functions	~~transpose(A)~~ ~~symmetric(A)~~ ~~skew(x)~~	~~outerProduct(v1,v2)~~ ~~cross(x,y)~~
Array Constructor and Concatination	array(..)	~~cat(..)~~

[a] No events are triggered from these functions for the proposed language subset (see [10, Section 16.8.1]).

[b] Only the "*Inferred Clock*" operator variant `Clock()` is supported. The other `Clock(..)` constructors as well as the `sample(u,c)` and `hold(u)` operators are not part of the language subset proposed for embedded target code generation (see Section 4.2.4 "Clock Blocks and Interaction with the Physical Environment").

[c] Present proposal excludes state machines (see Rule 7).

enable development and reviews to be conducted entirely on the graphical level.

Rule 7 - Block Diagrams Only. The present proposal focuses on the support of *block diagrams* and excludes *state diagrams*.
Rationale: This is to limit the required effort and associated complexity. Introduction of expressive state diagrams that integrate naturally with block diagrams and allow generation of efficient and safe code is a huge effort in its own.
Trace: Requirement 14.

Rule 8 - Causal Connectors Exclusivity. Only causal `Connectors` are allowed.
Rationale: Corollary to Rule 7

Rule 9 - Atomic Blocks for High-level Application. The use of atomic blocks is suggested for the top-level hierarchies of a model that shall be automatically translated into embedded C-code.
Rationale: Enables a clean and clear execution model, as well as an obvious translation of a block to a function call (see Section 4.2.6).
Trace: Requirements 4, 7, 8, 12, 15, 16, 17, 18.

Rule 10 - Basic/Composite Block Exclusivity. Blocks need to be *either* basic blocks *or* composite blocks. Mixing of textual equations with (graphical) block instances is not allowed.
Rationale: Mixing textual equations with graphical block instances in one block can be very confusing since a modeler may expect that the semantics of a composite block can be entirely deduced from the graphical level.
Trace: Requirement 15.

Rule 11 - Semantical Unambiguousness at Graphical Layer. The semantics of a composite block must be completely understandable at the graphical layer.
Rationale: This is required since otherwise code reviews can not be done at the graphical level.
Trace: Requirement 12, 15.

Rule 12 - Scalar Signal Extraction via "(De)mux" Only. Direct scalar signal connections from and to array connectors are not allowed. Intermediate "(De)mux" blocks must be used when scalar signals shall be connected with array connectors.
Rationale: Improves clarity and understandability of models.
Trace: Requirement 12, 15.

4.4 The SAFEDISCRETECONTROL Library

The conceptual SAFEDISCRETECONTROL library provides a restricted set of modeling blocks compliant with the requirements formulated in Section 3. Figure 10 gives an overview about the structure of the library.

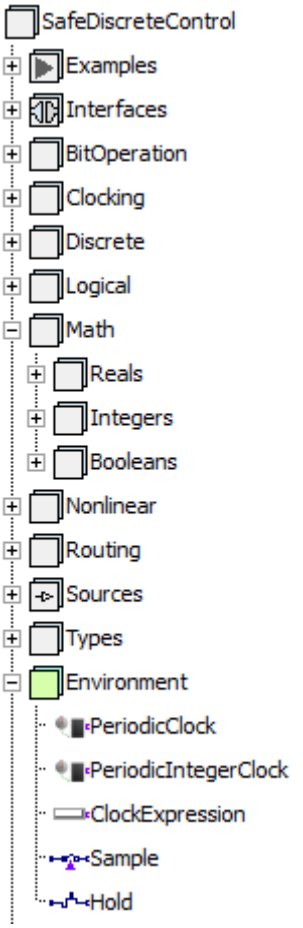

Figure 10: Structure of the SAFEDISCRETECONTROL library. Note that the Environment package contains blocks for modeling the environment in which the modeled high-level application is executed. However, these blocks may not be part of a high-level application (software) model.

As may be expected, the library has to duplicate many blocks found in the `Modelica.Blocks` standard library. However, it also needs to provide a user friendly access to the elements from the language superset, e.g., extended data types. Figure 11 shows a block for adding two integer signals that includes a choices menu to further specify the integer platform type to be used.

Figure 12 shows a traffic light controller modeled with the `SafeDiscreteControl` library. The controller is motivated by the example described in [13].

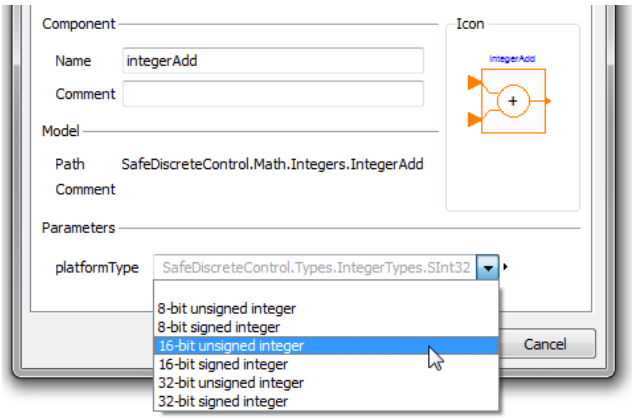

Figure 11: Integer addition block with extended data type support.

The controller's output are the interval lengths of the green phases, respectively for the north-south and the east-west direction. Note that an atom icon in the upper right corner provides the visual information that the block has been declared `atomic`.

The inside of the controller composite block is depicted in Figure 13. Semantically relevant information like data types or vector valued signals (e.g., between the multiplex and summation blocks) are clearly visible.

Adherence to the modeling rules described in Section 4.3.2 in combination with an adequate library allows to comply with the graphical level requirements formulated in Section 3.2. As a consequence, high-level application development (Role 1), as well as the model reviewing (Role 3), could be done entirely at the graphical level for the presented example.

Please note the presented SAFEDISCRETELIBRARY is a *conceptual* library that was created to check whether it is feasible to model typical discrete control algorithms and their related support logic solely with the use of the proposed sub- and superset of the Modelica language. It is therefore not a library that is available or usable for production purposes!

5 Validation Suites

In order to more clearly understand the role of validation suites in the qualification of development tools, the following section will provide formal definitions of relevant terms and a formalized description of the interplay between development tools and validation suites. Additionally this theoretical framework allows us to specify the relationship between specification

DOI 10.3384/ecp12076455

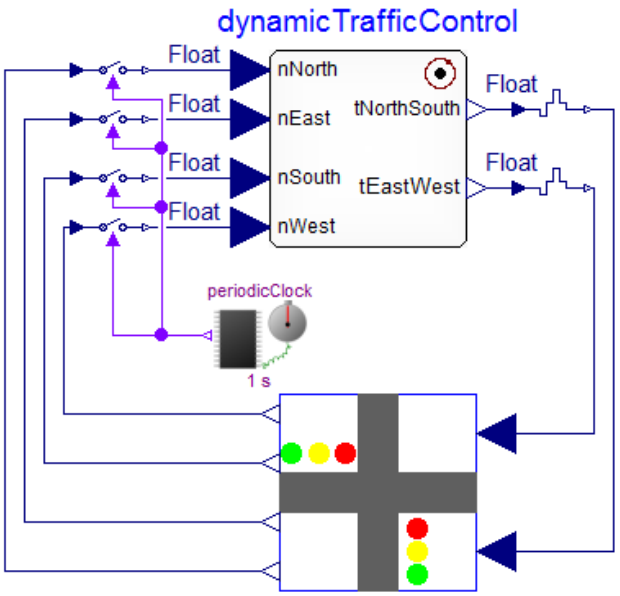

Figure 12: Model of a dynamic traffic light controller including the physical plant model of an intersection of two roads. The controller adjusts its timing and phasing to meet changing traffic conditions. Atomicity of the controller composite block is denoted by the atom symbol in the upper right corner. Data types are visible at the input and output connectors.

models in Modelica and code generation models in the proposed sub- and superset of Modelica.

5.1 Definitions

We define

- M as the set of all valid input models of the intended development tool chain suitable for code generation, i.e. in our case the set of models valid in the proposed Modelica sub- and superset.
- $\tilde{M} \subset M$ as the set of models given by a defined language subset for which the tool chain is to be validated[22],
- $\hat{M} \subset \tilde{M}$ as the set of all test models of a validation suite,
- S_m as the set of all valid stimuli for a given model $m \in M$, and
- $\hat{S}_m \subset S_m$ the set of all test stimuli of a validation suite for a given model $m \in \hat{M}$,

[22] Trivially $\tilde{M}$ can be M, though in practice the language is usually further subset to work around known defects in the code generator, elide unused language constructs or avoid language constructs not suitable for the intended application domain.

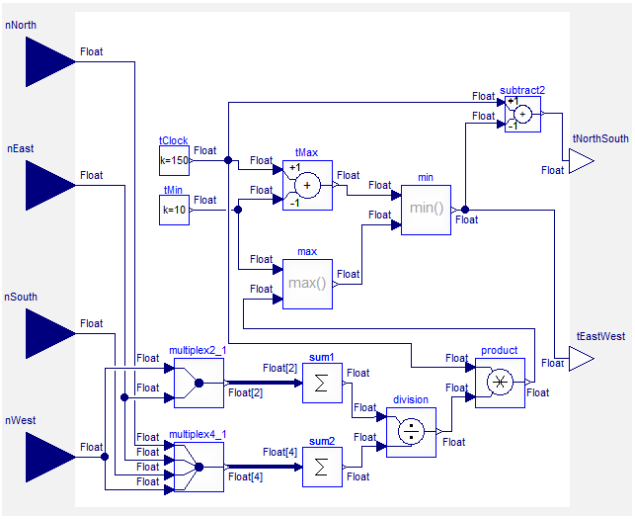

Figure 13: Inside the traffic light controller composite block. All semantically relevant information is visible at the graphical level.

and further the evaluation function

$$\text{eval}_{mil} : M \times S_m \to R_m$$

of a model by a theoretical simulator (Model-in-the-Loop), where R_m is the set of possible results of a given model $m \in M$.

We are interested in the proper functioning of a development toolchain (DT) consisting of an automatic code generator (ACG), compiler (C), assembler (A) and linker (L) for a target (t), so that

$$DT_t : M \to B_t = L_t \circ A_t \circ C_t \circ ACG_t$$

is the translation function of a model $m \in M$ into a binary B_t executable on target t.

Due to possible differences in the representations of stimuli and results between host and simulation (S_m, R_m) vs. target and executable (S'_m, R'_m) we also define mappings

$$g : S_m \to S'_m$$

$$h : R'_m \to R_m$$

converting between the different representations. Ideally $S'_m = S_m$ and $R'_m = R_m$ and thus $g = \text{id}_{S_m}$ and $h = \text{id}_{R_m}$.[23]

Finally

$$\text{eval}_{bil_t} : B_t \times S'_m \to R'_m$$

is the evaluation function of the binaries $b \in B_t$ on the target t by the processor (Binary-in-the-Loop).

[23] The mappings g and h are typically defined as component-wise mappings on the underlying algebraic data types.

5.2 The Task of a Validation Suite

We require from a correct development toolchain that

$$\forall m \in \tilde{M} : \forall s_m \in S_m : \\ d(\text{eval}_{mil}(m, s_m), h(\text{eval}_{bil_t}(DT_t(m), g(s_m)))) = 0$$

for a given metric $d : R_m \times R_m \to \mathbb{R}$.

Excluding the impact of d it is to be shown that the diagram in Figure 14 commutes.

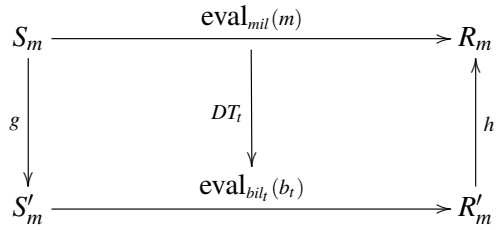

Figure 14: The diagrams shows the interaction of the defined sets and mappings.

Since it is generally not feasible to demonstrate this, the task of the validation suite is to gain confidence in the validity of this assertion by demonstrating the validitiy of the assertion only for the set of test models and test stimuli, i.e.

$$\forall m \in \hat{M} : \forall s_m \in \hat{S}_m : \\ d(\text{eval}_{mil}(m, s_m), h(\text{eval}_{bil_t}(DT_t(m), g(s_m)))) = 0$$

and ensuring through a suitable selection of the subsets $\hat{M} \subset \tilde{M}$ and $\hat{S}_m \subset S_m$, and additional measures of quality assurance, like e.g. fault-injection, that the generalisation to $m \in \tilde{M}$ and $s_m \in S_m$ is defensible.

5.3 Structure of the Sets M and B_t

The preceding analysis makes no reference to the structure of the sets of valid models M and the set of executable binaries B_t. We will analyse these sets in the following through the lens of the theory of algebraic specifications ([7, 8]).

In this context these sets are the sets of all terms with variables corresponding to their underlying signatures Σ_M or Σ_{B_t}. The sets of stimuli S_m or S'_m are then the sets of all possible values of the variables for given terms in M and B_t.

The evaluation functions eval_{mil} or eval_{bil_t} are correspondingly extended evaluation functions for a given Σ_M- or Σ_{B_t}-algebra, realized by the simulator or the target processor t.

The function DT_t is thus a transformation of terms from $M = T_{\Sigma_M}$ into terms of $B_t = T_{\Sigma_{B_t}}$. A closer examination of the resulting properties of the functions DT_t, g, h, eval_{mil} and eval_{bil_t}, especially with the means of category theory of the categories defined by Σ_M and Σ_{B_t} and their corresponding algebras can be helpful: The structure of the sets $M = T_{\Sigma_M}$ and $B_t = T_{\Sigma_{B_t}}$ has particular influence on the selection of the test sets $\hat{M}$ and $\hat{S_m}$, since the structure of these sets also affects the internal structure of the definition of the function DT_t to be tested.

In the test strategy of a validation suite, as described in [14], this observation among other considerations motivates the introduction of specific test areas dealing with the structure of the programming language, e.g. test areas *1 basic constructs of language*, *2 combined constructs of language*, and *4 inner equivalence*, as well as the structure of the transformation process, e.g. *6 internal structure of the code generator*.

Obviously the complexity of the structure, especially the number and kind of different language constructs as well as the number of different ways of combining them and any non-local effects, will determine to a large degree the size of the required test sets. Importantly this relationship due to combinatorial size explosion is at minimum quadratic or cubic, and possibly exponential in complexity. Therefore all effort should be expended to keep the language subset and the complexity of the language semantics as small and simple as possible.

5.4 The Relation between Specification Models and Code Generation Models

We assume in the following that the specification model of Figure 1 is a program in the language Modelica and the code generation model is a program of a possible subset of the proposed sub- and superset of the language Modelica for safety-relevant control applications. Then if we define

- $\underline{M}$ as the set of all valid Modelica models

there exists a mapping $r : M \to \underline{M}$ for every model $m \in M$ to a corresponding Modelica model $\underline{m} \in \underline{M}$. Thus

$$\underline{\tilde{M}} = r(\tilde{M})$$

is the subset of valid specification models corresponding to the set of code generation models expressible in the validated language subset.

Conversely, the left-unique, left- and right-total relation

$$p \subseteq \underline{\tilde{M}} \times \tilde{M} = \{(\underline{m}, m) \in \underline{\tilde{M}} \times \tilde{M} : r(m) = \underline{m}\}$$

DOI 10.3384/ecp12076455

represents the mapping between specification and code generation models.

The reason for p not being right-unique or functional is indicative of the freedom of choice of the developer in their implementation, for example in the choice of implementation data types, which are usually left open in the specification model.

If we restrict this freedom of choice by e.g. using a strict mapping of the data types[24] and providing other default implementation choices, we obtain a right-unique relation and with that a bijective mapping function

$$p' : \underline{\tilde{M}} \to \tilde{M}$$

with

$$\forall \underline{m} \in \underline{\tilde{M}} : \exists m \in \tilde{M} : (\underline{m}, m) \in p \wedge p'(\underline{m}) = m \wedge \underline{m} = r(p'(\underline{m}))$$

With corresponding definitions for the functions $\underline{g} : \underline{S_m} \to S_m$ and $\underline{h} : R_m \to \underline{R_m}$ we expand the diagram from Figure 14 for the evaluation function $\text{eval}_{spec} : \underline{M} \times \underline{S_m} \to \underline{R_m}$, see Figure 15.

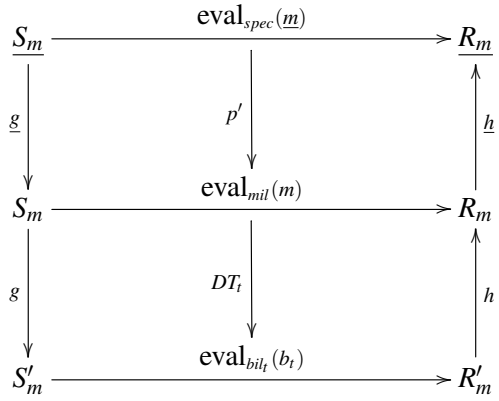

Figure 15: The diagram extends the diagram of Figure 14 with the interaction of the level of the specification model.

This diagram should commute, too, if needed taking into account a suitable (pseudo-)metric $\underline{d} : \underline{R_m} \times \underline{R_m} \to \mathbb{R}$. For other models $m' \in M$ with $(\underline{m}, m') \in p$ and $m' \neq m$ the diagram in Figure 15 may commute with suitable mappings $\underline{g}$, $\underline{h}$ and a (pseudo-)metric for suitable stimuli.

Taken together these characteristics make it possible to produce a set of validated tools for both the code generation from code generation models and the transformation from specification to code generation models, so that the probability of fault injection along those two (independent) transformations can be minimized.

[24] We map e.g. all `double` in the specification models to `double` in the code generation model, etc.

6 Conclusion

The article presented a general set of *requirements* that need to be imposed on a high-level, domain-oriented modeling language and its development tools in order to use it for the development of safety-relevant applications. Based on these requirements the suitability of using Modelica within a safety related development process was further analyzed and a *sub- and superset of the Modelica language* was proposed that seems capable of satisfying the formulated requirements.

As a preferred method of choice to gain confidence in software development tools the use of a *validation suite* was proposed and a formal description of the role of a validation suite within a tool qualification effort was given. The precise understanding of the task and effort needed to qualify a tool (based an the validation suite method) is necessary to appreciate the importance of minimizing number and complexity of the allowed language elements.

A prototypical development of a block diagram library (denoted SAFEDISCRETECONTROL) based on established data flow semantics was started in order to test the suitability of the proposed language set to satisfactorily model typical control applications. The first analysis is very encouraging. Currently, there are approximately 60 candidates of blocks and their parameters for the intended SAFEDISCRETECONTROL library. This is comparable to the number of blocks and parameters of already successful validation projects for comparable modeling languages and their development tools, so that the implementation of a validation suite for this language set seems eminently possible.

The next task will be to demonstrate the practical suitability of the proposed sub- and superset (which comprises less than half of the language elements of the Modelica Standard 3.3) for real applications. While it will likely become necessary to enhance or modify the language set based on the practical lessons learned, it will remain of crucial importance, to limit the number of the blocks in the SAFEDISCRETECONTROL library as much as possible:

Experience with validation suites for other modeling languages has shown, that for a language with around 90 basic building blocks, the test sets of the test areas dealing mainly with language structure (i.e. test areas *1 basic constructs of language*, *2 combined constructs of language*, and *4 inner equivalence*, see again [14]) already comprise around 223 000 test outputs. Asuming only quadratic or cubic growth, an increase of a mere 10 % in blocks and parameters will result in an increase of approximately 20–30 % in val-

idation effort. A strong release management process on the SAFEDISCRETECONTROL library with an ideal limit of about 50 basic building blocks therefore seems advisable.

It should be noted that tool qualification is most effective when performed in an early phase of the development process: Errors in development tools are usually hard to detect and analyse in normal development, and the effects of work-arounds and tool limitations can have a huge impact on the efficiency of the development process when introduced at a late stage of development. Tool qualification should therefore be concluded prior to the start of the development project. Ideally, tool qualification efforts start even before the language definition is finalized: When language constructs and definitions are viewed through the lens of tool qualification, error-prone or hard to test constructs and corner cases are highlighted, and improvements can still be adopted. By the concurrent creation of a test suite, interpretation of the language definition can be clarified and harmonized between possible implementations. For this reason we would welcome further work in this area even at this early stage of language set definition.

The authors hope that, on the one hand this contribution sheds some light on the requirements and intricacies that need to be faced when considering the usage of Modelica for safety related applications, and on the other hand hope to stimulate further discussion on the rationale and possible approaches to employing Modelica in this field.

Acknowledgments

The first author would like to thank Dominik Sommer for the valuable discussions about safety relevant software development in aerospace applications.

References

[1] Albert Benveniste, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. In *Proceedings of the IEEE*, volume 91 (1), pages 64–83, 2003.

[2] Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. Clock-directed modular code generation for synchronous data-flow languages. *SIGPLAN Not.*, 43(7):121–130, June 2008.

[3] Tilman Bünte, Akin Sahin, and Naim Bajcinca. Inversion of Vehicle Steering Dynamics with Modelica/Dymola. In Gerhard Schmitz, editor, 4th *Int. Modelica Conference*, March 2005.

[4] David Broman. *Meta-Languages and Semantics for Equation-Based Modeling and Simulation*. PhD thesis, Linköping University, PELAB - Programming Environment Laboratory, The Institute of Technology, 2010.

[5] Manfred Broy, Helmut Krcmar, Jens Zimmermann, and Sascha Kirstan. Einfluss des Software-Designs auf die Wirtschaftlichkeit von Software-Entwicklungen. *ATZelektronik*, 02:34–37, April 2011.

[6] I. S. Duff and J. K. Reid. An implementation of tarjan's algorithm for the block triangularization of a matrix. *ACM Trans. Math. Softw.*, 4(2):137–147, June 1978.

[7] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1985.

[8] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1990.

[9] IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, 1998.

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

[11] Ramine Nikoukhah and Sébastien Furic. Towards a full integration of modelica models in the scicos environment. In *7th Modelica Conference, Como, Italy*, September 2009.

[12] Constantinos C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231, 1988.

[13] Stefan-Alexander Schneider and Tobias Hofmann. Functional Development with Modelica: A Use-Case Analysis. In 9th *Int. Modelica Conference*, Munich, Germany, September 2012.

[14] Stefan-Alexander Schneider, Tomilav Lovric, and Pierre Mai. The validation suite approach to safety qualification of tools. *SAE Technical Paper 2009-01-0746*, 2009.

[15] Robert Tarjan. Depth-first search and linear graph algorithms. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pages 114 –121, oct. 1971.

[16] The Motor Industry Software Reliability Association. MISRA-C:2004 - Guidelines for the use of the C language in critical systems, 2004. http://www.misra.org.uk.

[17] The Motor Industry Software Reliability Association. MISRA AC AGC - Guidelines for the application of MISRA-C:2004 in the context of automatic code generation, 2007. http://www.misra.org.uk.

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

[19] Michael Thümmel, Martin Otter, and Johann Bals. Vibration control of elastic joint robots by inverse dynamics models. In H. Ulbrich and W. Günthner, editors, *IUTAM Symposium on Vibration Control of Nonlinear Mechanisms and Structures*, pages 343–353, München, 2005.

DOI 10.3384/ecp12076455