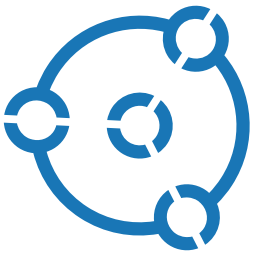
D3.2	Translation validation and traceability concept from acausal hybrid models to generated code
Access[1]:	**PU**
Type[2]:	**Report**
Version:	**1.0**
Due Dates[3]:	**M12**



openCPS

Open Cyber-Physical System Model-Driven Certified Development

Executive summary[4]:

This deliverable reports about concepts for translation validation and traceability for continuous-time models based on (acausal) equations to C/C++ code. It proposes a testing-based translation validation of generated code which is subdivided in two steps: (1) back-to-back testing to demonstrate numerical equivalence between model and code, and (2) traceability analysis to demonstrate structural equivalence.

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

Deliverable Contributors:

	Name	Organisation	Primary role in project	Main Author(s)[5]
Deliverable Leader[6]	Bernhard Thiele	LIU	WP3 leader	X
Contributing Author(s)[7]	Per Sahlin	EQUA	T3.3 leader	
Internal Reviewer(s)[8]	Martin Sjölund	LIU	WP4 leader	
	Magnus Eek	Saab AB	Project Coordinator	

Document History:

Version	Date	Reason for change	Status[9]
0.1	20/10/2016	First Draft Version	Draft
0.2	09/11/2016	Integrated comments by reviewers	In Review
1.0	14/11/2016	Release version	Released

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

Contents

Acronyms 4

Symbols 4

1 Introduction 6

2 Code Generation 7
2.1 Safeguarding Generated Code . . . 7
2.2 Testing-Based Translation Validation of Generated Code . . . 9
2.3 Modelica Translation in the Context of Production Code Generation . . . 9
2.4 Translation Complications for Continuous-Time Acausal Models . . . 10
2.5 Traceability of Generated Code . . . 10

3 Translation Validation and Traceability Concept 11
3.1 Back-to-back Testing . . . 12
3.1.1 Test Execution . . . 12
3.1.2 Equivalence Comparison . . . 12
3.1.3 Implementation Concept for Modelica . . . 13
3.2 Traceability Analysis . . . 13
3.2.1 Tracing Symbolic Manipulations . . . 13
3.2.2 Implementation Concept for Modelica . . . 14
3.3 Language Restrictions . . . 15

4 Conclusions 16

A Foundations of Modelica 17
A.1 Objectives of the Modelica Language . . . 17
A.2 Semantic Foundation . . . 17
A.3 Translation Approaches . . . 19
A.3.1 Compiler Generation from Operational Semantics Specifications . . . 19
A.3.2 Compiler Generation from Reference Attribute Grammars . . . 20
A.3.3 Translational Semantics . . . 20
A.3.4 Summary . . . 21
A.4 Modelica Hybrid DAE Representation . . . 21
A.4.1 Equations . . . 22
A.4.2 Simulation . . . 23
A.4.3 Event Iteration . . . 23
A.4.4 Remarks on DAEs . . . 24
A.5 Super-Dense Time . . . 25

B Requirements for Model-Based Function Development 27
B.1 Development Roles . . . 28
B.2 Modeling Language Requirements . . . 28
B.3 Graphical Representation Requirements . . . 32

References 38

Acronyms

AE	Algebraic Equation.
DAE	Differential Algebraic Equation.
EBNF	Extended Backus–Naur Form.
EOO	Equation-based Object-Oriented.
FMI	Functional Mock-up Interface.
FMU	Functional Mock-up Unit.
HIL	Hardware-in-the-Loop.
IC	Initial Condition.
IVP	Initial Value Problem.
JSON	JavaScript Object Notation.
LHS	Left-Hand Side.
MBD	Model-Based Development.
MIL	Model-in-the-Loop.
MLS	Modelica Language Specification.
ODE	Ordinary Differential Equation.
PDE	Partial Differential Equation.
PELAB	Programming Environments Laboratory.
PIL	Processor-in-the-Loop.
RHS	Right-Hand Side.
RML	Relational Meta Language.
UTP	Unifying Theories of Programming.
V&V	Validation and Verification.

Symbols

$c(t_e)$	vector containing all Boolean condition expressions, *e.g.*, if-expressions.
$m(t_e)$	vector of discrete-time variables of type **discrete Real**, **Boolean**, **Integer**, **String**. Change only at event instants t_e.

$m^{\mathbf{B}}(t_e)$	vector of discrete-time variables of type **Boolean**, $m^{\mathbf{B}}(t_e) \subseteq m(t_e)$. Change only at event instants t_e.
$m_{\text{pre}}(t_e)$	values of m immediately before the current event at event instant t_e.
$m^{\mathbf{B}}_{\text{pre}}(t_e)$	values of $m^{\mathbf{B}}$ immediately before the current event at event instant t_e, $m^{\mathbf{B}}_{\text{pre}}(t_e) \subseteq m_{\text{pre}}(t_e)$.
$p^{\mathbf{B}}$	parameters and constants of type **Boolean**, $p^{\mathbf{B}} \subseteq p$.
p	parameters and constants.
t	time.
$v(t)$	vector containing all elements in the vectors $x(t)$, $\dot{x}(t)$, $y(t)$, $[t]$, $m(t_e)$, $m_{\text{pre}}(t_e)$, p.
$x(t)$	vector of dynamic variables of type **Real**, *i.e.*, variables that appear differentiated at some place.
$\dot{x}(t)$	differentiated vector of dynamic variables.
$y(t)$	vector of other variables of type **Real** which do not fall into any other category (= algebraic variables).

1 Introduction

This deliverable reports about concepts for translation validation and traceability for continuous-time models based on (acausal) equations to C/C++ code. It describes a conceptual base for the translation of Modelica models to production code which is suitable for being used in safety-related embedded systems. The report provides essential input for the subsequent prototype developments in the same work package (T3.3, T3.4) and to the development of the testing infrastructure in T4.4.

Modern control technology often uses a model of the physical plant as part of advanced, digitally implemented control strategies. Modelica's excellent capabilities for physical modeling can be leveraged to synthesize such advanced controllers [AB06, TSGT08, TKOB05]. However, up to now there exists no qualified Modelica-based code generator that allows to directly use the generated code in safety-related (production) systems. The presented concepts in this report aim at enabling a code generation process which overcomes this limitation.

Current qualified translation tools for model-based control development are based on discrete-time, directed (causal) data-flow. An important aim in this research project is to extend the state of the art by allowing to directly validate the translation of continuous-time models based on (acausal) equations. This is a major enhancement over the State-of-the-Art (SotA) that allows eliminating several manual steps in the development process and thus increases development efficiency substantially as depicted in Figure 1.

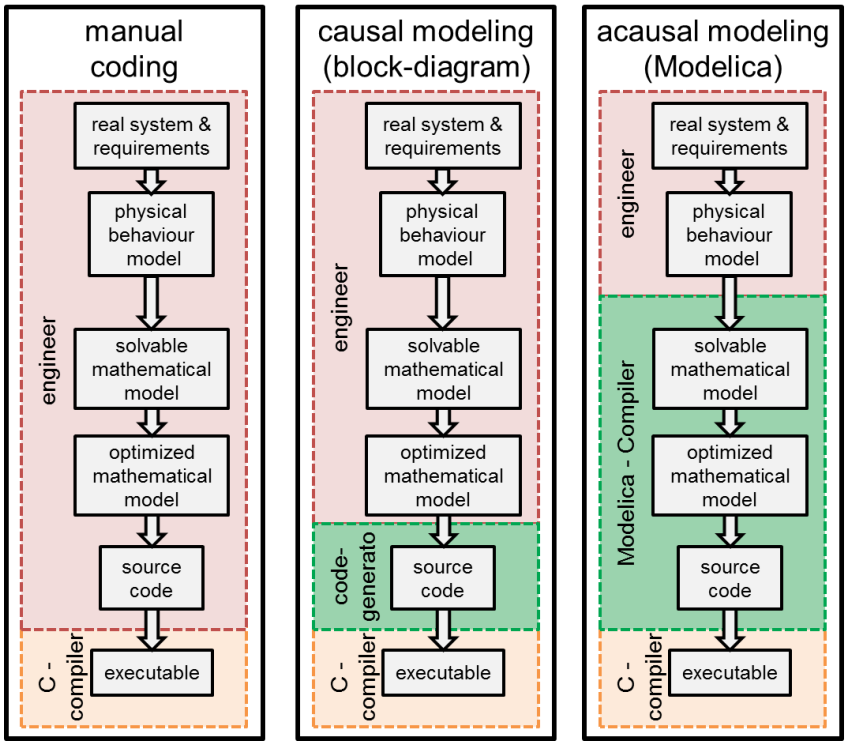


Figure 1: Comparison of code generation processes. Automation potential from manual coding over causal block modeling with code generation to the envisioned process based on acausal modeling.

The safety-related concepts discussed in this report are essentially orthogonal to the code-generation concepts discussed in the D3.3 report "Concept for customizing code generation

including flexible adaptation to different target systems" [TB16], *i.e.*, both concepts solve different problems but they can be (and will be) used together. Background information which is related to both reports is not repeated twice, instead references are used for referring to respective document parts.

2 Code Generation

To optimize the benefits gained by a model-based development process it is crucial that (high-level) models can be automatically transformed into executable binary code for the respective embedded platform, thus eliminating error prone and expensive manual recoding of the application into a general-purpose programming language.

This is typically achieved by generating embedded C-code which a cross-compiler transforms into object code as depicted in Figure 2. The model of the application software acts as executable specification which can be simulated together with the physical process (plant model). This kind of simulation is termed Model-in-the-Loop (MIL) simulation. The high-level specification model is further enriched and refined with implementation details and finally serves as the basis of the implementation (code generation model), *i.e.*, it is the input to the code generator. Notice that generated code that is executed on an embedded system (in a Cyber-Physical System (CPS) context) typically needs to integrate into an existing software architecture (compare [TB16, Section 2.1]). The figure shows, that the code generated from the behavioural model (the "Function Software", providing the actual function, *e.g.*, a control algorithm defined in Modelica), needs to be integrated with other software components, which were not produced by the tool chain, *e.g.*, (hand-written) legacy software components and the base software[10].

A prerequisite for understanding the concepts which are introduced in the following sections is an adequate knowledge about the foundations of Modelica. For that purpose Appendix A describes typical translation approaches for Modelica and discusses static and dynamic language semantics. In case that these concepts are already known by a reader, this material can be skipped.

In previous work one of the authors compiled a catalogue of requirements for a modelling language and associated tools (including the code generator) within a model-based control function development process [Thi15, Chapter 3]. Because of its relevance for the following discussion a copy of this chapter is made available in this report as Appendix B.

2.1 Safeguarding Generated Code

In the case of safety-related software it is required that sufficient confidence can be placed in its correctness. If the software is generated from a high-level language/specification the translation must be acceptable to the customer, regulatory agencies, and any legal interests. A general discussion about requirements for high-integrity code generation is provided by Whalen and Heimdahl [WH99] and in the more recent paper by Hatcliff et al. [HWK+14].

[10] Software components which offer services needed to run functional parts of an upper software layer.

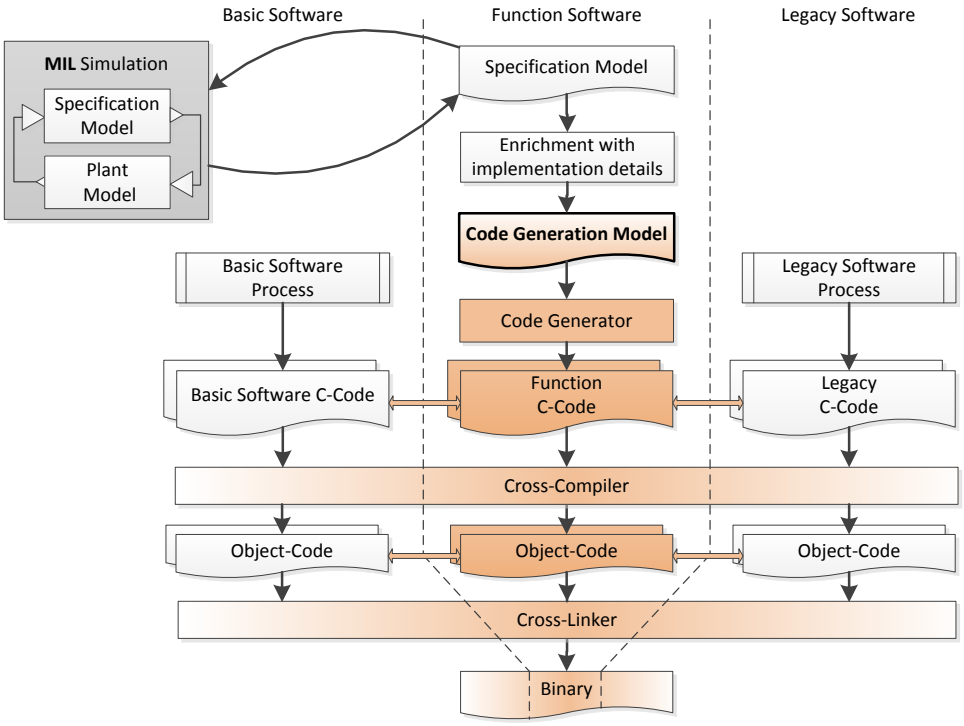


Figure 2: The generic build process for a Model-Based Development (MBD) tool chain with an automatic code generator (adapted from [SLM09]).

While there exists mature C or Ada compilers which have been proven reliable in use, code generators for high-level languages are typically not as mature and their output must be checked with almost the same expensive effort as is needed for manually written code [SWC05]. Stürmer et al. [SWC05] give an overview over state-of-the-art techniques for safeguarding generated code in a model-based development process.

An option to reduce the safeguarding effort is to rely on (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*. Frank et al. [FGH+08] provide an overview over compiler verification methods and testing technology that can be used in this context. Schneider et al. [SLM09] describe the practical qualification of a complete model-based tool chain for production code generation within the automotive industry (for models based on a discrete-time, causal data-flow formalism).

Apart from verification or systematic testing of compilers, another option to increase the confidence in the correctness of generated code is by using *translation validation* techniques. The term was introduced by Pnueli et al. [PSS98] to refer to verification approaches where each individual translation (*i.e.* a run of the compiler) is followed by a validation phase which verifies that the target code produced on this run correctly implements the submitted source program.

While [PSS98] implies formal verification techniques, the concept of translation validation has been transferred to approaches that use systematic testing instead of formal verification for carrying out verification and validation of generated code in an engineering context [Con09].

2.2 Testing-Based Translation Validation of Generated Code

This work adapts the approach of testing-based translation validation of generated code proposed by Conrad [Con09] in the context of the IEC 61508 safety standard [IEC00]. Similarly to [Con09], the proposed translation validation approach does not imply automatic formal verification but rather strives to provide support for the developer to carry out equivalence checks between the model and the generated code.

Conrad suggests an application-specific verification and validation workflow which is subdivided into the following two steps:

1. Demonstrating that the model is well-formed and meets all requirements.
2. Showing that the generated code is equivalent to the model.

The first step, termed *design verification*, combines suitable Validation and Verification (V&V) techniques at the model level. The second step, termed *code verification*, uses equivalence testing (also known as back-to-back testing or comparative testing) and other techniques to demonstrate equivalence between the model and the generated code compiled into an executable.

This report concentrates on the second step. Starting point is the *code generation model*[11] from Figure 2.

2.3 Modelica Translation in the Context of Production Code Generation

Looking at the typical Modelica translation stages described in Section A one can identify (non-exhaustively) following problems [TKF15, Thi15]:

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.

In order to solve these problems one of the authors of this report proposed, in previous work, a strongly restricted clocked discrete-time subset of Modelica and presented a transformation from this Modelica subset to a synchronous data-flow kernel language which is mainly based on local transformation rules [TKF15, Thi15]. This synchronous data-flow kernel language allows, in principle, to resort to established compilation techniques for data-flow languages

[11] There is no common agreement about how to term this model. Other terms besides *code generation model* include *implementation model*, *golden model*, or *model used for production code generation*.

which are understood enough to be accepted by certification authorities (however, up to now no qualifiable industrial-grade code-generator was developed on this basis). Notice, that the restricted language subset considered in this previous work has no direct support for continuous-time acausal models, hence, the associated code generation process is placed in the middle of Figure 1.

2.4 Translation Complications for Continuous-Time Acausal Models

Current qualified translation tools for model-based control development are based on discrete-time, causal (directed) data-flow. Direct use of continuous-time, acausal models has the potential of increasing development efficiency substantially as depicted in Figure 1. However, it also introduces several (non-exhaustive) complications:

1. The substantial symbolic manipulation during the equation transformation phase interferes with the ability of the developer to tightly control the evaluation of expressions in order to control finite precision issues in calculations or to detect and safeguard potential divisions through zero or similar problems.
2. Solution traces of an execution are dependent on the utilized discretization method. Utilizing a different method or different method parameters may drastically affect results including the numeric stability of the system.
3. Systems which have algebraic loops ("strong components") may result into nonlinear systems of equations which need to employ numeric root finding methods. In general, no guarantees can be given on the upper bound of required iterations of a root finding method (like the famous Newton-Raphson method) and whether the method will converge to a solution.
4. If the system has both, continuous-time and discrete-time behaviour, the execution needs to be controlled by a discrete-event system for which event triggering relations need to be monitored. At an event instant a mixed system of continuous-time and discrete-time equations needs to be solved, which may involve Real, Integer, and Boolean equations. For more general cases such systems are handled using iterative heuristics for which convergence to a solution (if it exists) cannot be guaranteed.

2.5 Traceability of Generated Code

Traceability refers to the property that 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. Traceable code facilitates validation activities, *e.g.*, manual inspections, automated analysis of the generated code[12], and testing [WH99]. Traceability is one of the requirements for a code generator which is formulated in [Thi15, Chapter 3, Requirement 8][13].

The extensive symbolic manipulation during the translation of Equation-based Object-Oriented (EOO) languages like Modelica makes it very hard to trace back fragments of the generated

[12] One example is to perform code coverage analysis on the target level but mirror back the results onto the model level.

[13] Reproduced in Section B.2

C/C++-code to the model elements that caused its generation. This is also a problem for debugging EOO languages. Run-time errors often result in error messages which are of little help to a user, since it is not clear which model elements are responsible for the observed error.

3 Translation Validation and Traceability Concept

This section is concerned with elaborating on the second step of the testing-based translation validation approach described in Section 2.2 — *Showing that the generated code is equivalent to the model*.

The basic idea for the envisioned translation validation of generated code is depicted in Figure 3. In order to show that the generated code is equivalent to the model the process is subdivided in two steps:

1. *Back-to-back testing* (aka equivalence testing): Demonstrate *numerical equivalence* between the model and the executable derived from the generated code.
2. *Traceability analysis*: Demonstrate *structural equivalence* between the model and the generated code, *i.e.*, demonstrate that the generated code does not perform any unintended function

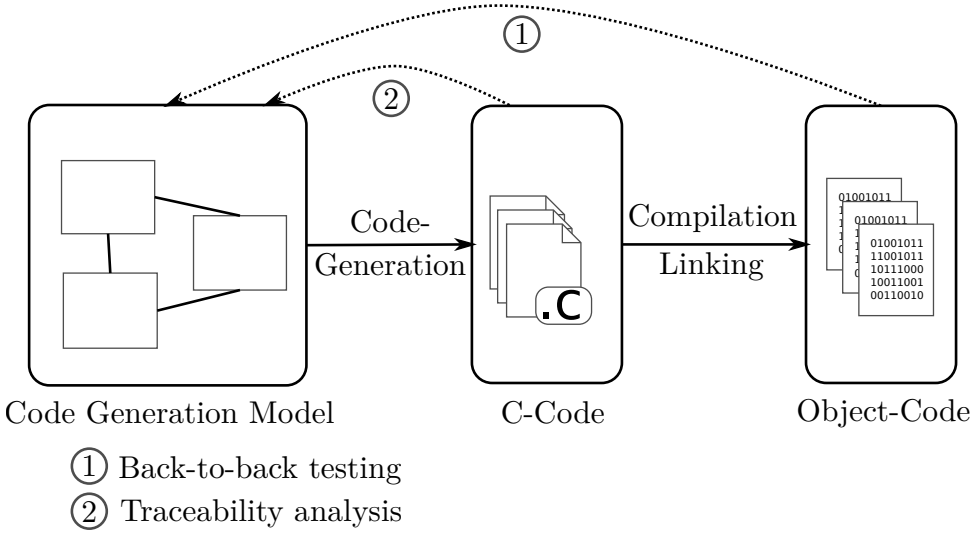


Figure 3: Translation validation process for generated code (compare with [Con09, Fig. 4]).

In addition, or alternatively to a traceability analysis, *model versus code coverage comparison* is proposed in [Con09]. This requires tool support to gather coverage information on the model level, which is currently not available (but in principle feasible) in OpenModelica. If discrepancies between model and code coverage are detected they need to be assessed. Again, traceability from model to code can be beneficial in this case, since it may allow to mirror back coverage analysis results for the generated code onto the model level.

3.1 Back-to-back Testing

The aim of back-to-back testing is to demonstrate the numerical equivalence between result vectors obtained by model simulation and result vectors obtained by execution of the generated code on the target platform[14].

Figure 4 outlines the approach. The simulation as well as the execution of the object code is subjected to the same set of stimuli (test vectors) and the respective system reactions (result vectors) are compared.

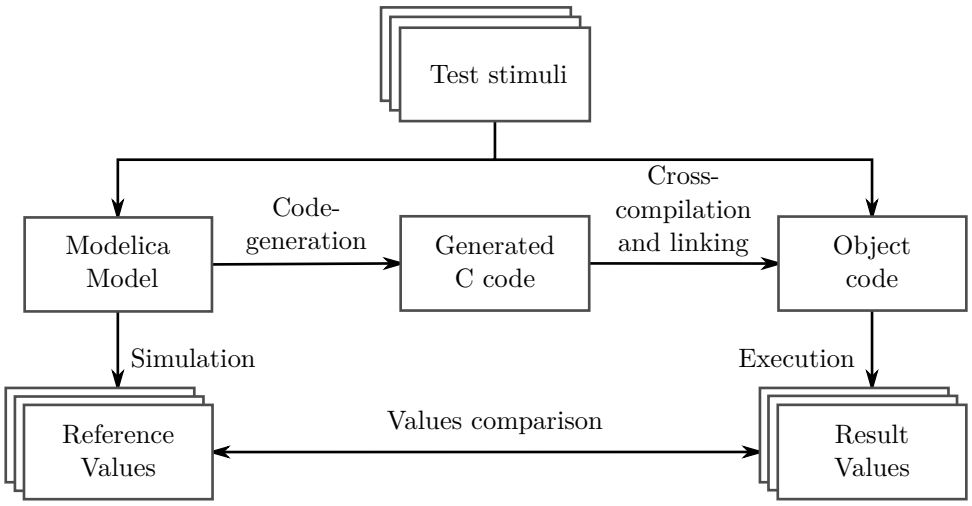


Figure 4: Back-to-back testing for numerical equivalence (compare with [Con09, Fig. 5]).

3.1.1 Test Execution

The resulting object code should be executed on a platform which is as close to the product as possible, *e.g.*, on the target processor or on a target-like processor (Processor-in-the-Loop (PIL) testing). Alternatively, the use of instruction set simulators for the target processor may be feasible, too. It should be noted that this approach to testing is not intended to verify real-time aspects of the system (this aspect can be checked by Hardware-in-the-Loop (HIL) simulation or by testing within an actual hardware prototype).

3.1.2 Equivalence Comparison

Even for a correct translation bitwise identical behavior cannot be expected when comparing the simulation results with the results of the object-code running on a target-like processor. Reasons for diverging results include limited precision of floating point numbers and differences between compilers, for example due to different compiler optimizations. Hence, suitable concepts and algorithms for testing on sufficient equality between the result vectors are required. Besides elementary comparison algorithms based on absolute or relative differencing there exist also more elaborate approaches (see [CSW05]). It depends on the application at

[14] Hence, after code generation and (cross-)compilation and linking for the target platform.

hand and the characteristics of the result signal whether a particular comparison algorithm can be considered as suitable.

3.1.3 Implementation Concept for Modelica

The OpenModelica test suite already provides basic comparison algorithms which can be used as a starting point to enable equivalence comparison testing. In order to inject stimuli and monitor behaviour and output of the target processor a suitable test harness (automated test framework) is required. The test harness needs to be adapted to the specific target hardware in order to allow the required stimuli injection and (output) behaviour observation.

The OpenModelica scripting based test suite provides a starting point which can be extended to allow back-to-back testing. OPENCPS task T4.4 “Test-based verification” aims at developing such an extension.

3.2 Traceability Analysis

The goal of the traceability analysis is to demonstrate that the generated code does not perform any *unintendend function*. It serves as technique to demonstrate *structural equivalence* between model and code. Traceability refers to the property that given a fragment of the generated code, it should be possible to trace it back to the model elements that caused its generation.

Traceability analysis means that the generated code is subjected to a limited review that exclusively focuses on traceability aspects. The traceability relations between model to source code elements and vice versa are analysed. *Non-traceable code* shall be flagged and assessed [Con09].

3.2.1 Tracing Symbolic Manipulations

The problem of tracing symbolic manipulations in EOO languages like Modelica was stated in Section 2.5. This is not only an issue for allowing a feasible traceability analysis, but also a hindrance for debugging EOO languages. When run-time errors occur in EOO languages the resulting error message are often of limited help to a user, since it is not clear which model elements are responsible for the observed error.

The traceability problem in Modelica is targeted by Sjölund et al. [Sjö15, PSA+14] for allowing equation based debugging of Modelica models in OpenModelica. Sjölund’s approach is to record model transformations as meta data during the translation process and provide means of using that information for tracing errors occurring in the binary executable back to the responsible model equations. Figure 5 outlines the approach. The symbolic manipulations during a translation are recorded and saved into a file using a JavaScript Object Notation (JSON) based encoding format. The JSON format is well known and easy to parse. This facilitates creating custom tools which utilize the recorded meta-data.

The OpenModelica *transformation browser* uses the model transformations recorded in the JSON file and allows to browse through equation transformations that have been performed in

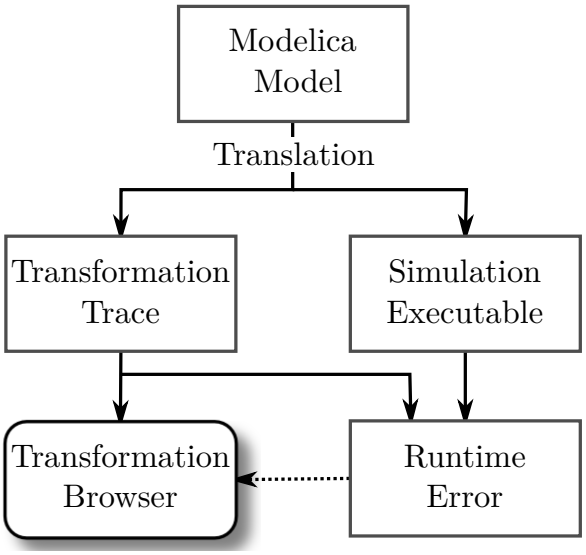


Figure 5: Using trace information from translation for subsequent debugging (adapted from [Sjö15, p. 54]).

a debugging session. Figure 6 shows a screen shot of the browser as well as an example for different equation transformations that may be carried out during a model translation.

3.2.2 Implementation Concept for Modelica

For enabling a traceability analysis of Modelica model translation the existing method of transformation recording is the starting point. Instead of providing the information in the context of debugging the information needs to be processed in a way that facilitates a traceability review in a translation validation process as indicated in Figure 3.

This can be achieved by generating readable code with appropriate comments which indicate the source model elements that caused the generation of a particular C/C++-code fragment. The generation of code review documentation as requested in the OPENCPS task T3.4 “Report generation for supporting V&V activities” can further improve the traceability review by using hyperlinks as a means to link generated code artifacts to respective elements from the source model. At the same time, such a document can be enriched with information which reveal the performed symbolic transformations.

In addition one may aim for keeping performed equation transformation as simple and localized as possible to facilitate traceability analysis. One complication is that the typical Modelica translation process impedes modularization as described in Section 2.3. In general, the composition of Modelica classes/models cannot be described by highly localized rules. However, this would be desirable to facilitate a traceability review. For example, it would be desirable to conduct a traceability review on a Modelica model/class which is instantiated several times within the complete model only *once*, and after that, only review that composition rules, *e.g.*, connections to other components, have been translated correctly. However, equations are manipulated and transformed at the global model level which impedes straight-forward reuse of already validated components. Also, due to this translation properties, reducing validation complexity by appropriate decomposition of the global model may not be very effective.

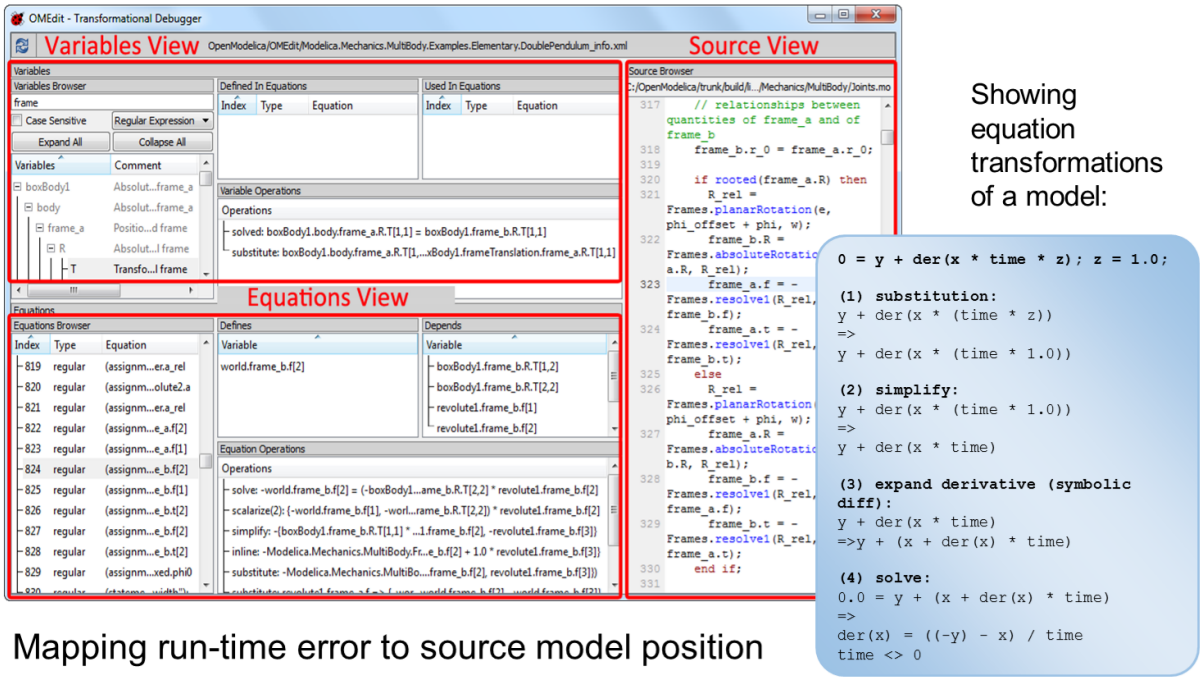


Figure 6: OpenModelica's *transformation browser* for equation based models.

A mitigation possibility for this problem is to work with components with an Functional Mock-up Interface (FMI) like interface. The composition of such components is more restricted than for general Modelica components (*e.g.*, no support of acausal connectors), but their composition can rely on more local rules so that reuse of an already validated component is possible with less effort. Another advantage of using an FMI like interface is that a sound mathematical description of the dynamic semantics in terms of a super-dense time conceptual framework is already provided in the standard (see Section A.5).

It is then left to the developer (modeller) to make skilled trade-off decisions between using component composition with a high flexibility, but potentially more complex translation validation *vs.* using less flexible component compositions, but possibly more manageable, more modular translation validation.

3.3 Language Restrictions

Imposing language restrictions is a common approach to increase compiler safety. The MISRA-C guidelines [The04] are a well-known example for restrictions and rules for the use of the C language in safety-relevant systems. There exists also a MISRA standard for using the C language in the context of automatic code generation [The07].

Previous work of the author proposed a rather restricted clocked discrete-time subset of the Modelica language [TSM12, TKF15, Thi15] to allow qualifiable code generation from Modelica. In [TKF15, Thi15] it is shown how such a subset can be translated formally to a well-understood synchronous data-flow kernel language. This kernel language then allows to resort to established compilation techniques for data-flow languages which are understood enough to be accepted by certification authorities.

However, the aim of this work is to develop a translation approach that allows for a less restricted language subset. It is expected that there are necessary or sensible restrictions and guidelines which need to be established in practice, but it is considered part of the upcoming work within this project to explore these restrictions.

4 Conclusions

A concept for translation validation and traceability has been presented which proposes to subdivide the translation validation process into two steps (see Figure 3):

1. *back-to-back testing* for demonstrating *numerical equivalence* between the model and the executable derived from the generated code, and
2. *traceability analysis* to demonstrate *structural equivalence* between the model and the generated code (verify that that the generated code does not perform any unintended function).

Development of the back-to-back testing can extend on the existing OpenModelica test suite and is connected to the OPENCPS task T4.4 “Test-based verification”.

Traceability analysis can extend on work which has been conducted in the area of equation-based debugging in OpenModelica. Debugging of equation-based models requires to record the symbolic manipulations performed during the translation process and provide means for tracing errors occurring in the binary executable back to responsible model equations. The development of the traceability analysis capabilities can build on results from the existing OpenModelica support for equation-based debugging. However, it is needed to adapt and transfer respective technology so that requirements on an efficient and effective traceability review process can be met.

The code generation needs to be adapted so that tracing back generated code fragments to their source model elements is facilitated (task T3.2 and T3.3). Furthermore, the generation of core review reports is needed which support traceability analysis, *e.g.*, by providing hyperlinks as a means to link generated code artifacts to respective elements from the source model (task T3.4). Additionally, such a document can be enriched with information which reveal the performed symbolic transformations. The traceability of symbolic transformations (particularly for state machines) is related to work conducted in OPENCPS WP 4 (particularly task T4.1 “Debugging of state machines”).

A Foundations of Modelica

One of the authors contributed to a technical report by Foster et al. [FTW15]. This section is taken from this author's contributions to [FTW15] and adapted to fit into the context of the current report.

A.1 Objectives of the Modelica Language

Modelica is language for describing the dynamic behaviour of technical systems consisting of mechanical, electrical, thermal, hydraulic, pneumatical, control and other components. The behaviour of models is described with

- *Ordinary Differential Equations (ODEs)*,
- *Algebraic Equations (AEs)*,
- *event handling* and *recurrence relations* (sampled control).

Object-oriented concepts are supported as a means of managing the complexity inherent to modern technical systems. Modelica can therefore be called an EOO language. That term, coined by Broman [Bro10], nicely subsumes the central, distinguishing language characteristics.

Equation-based: Behavior is described declaratively using mathematical equations.

Object-oriented: Objects encapsulate behavior and constitute building blocks that are used to build more complex objects.

The most recent standard version is the Modelica Language Specification (MLS) 3.3 [Mod14]. An important extension in MLS 3.3 is the addition of support for improved discrete-time modelling for control algorithms [Mod14, Chapter 16 and 17] inspired by synchronous languages semantics [BEH+03].

It is worth noting that directly describing models using *Partial Differential Equations (PDEs)* is currently beyond the scope of the Modelica language[15]. However, it is possible to use results of tools that support PDEs or discretise simple PDEs manually.

A.2 Semantic Foundation

Quoting from [Mod14, Section 1.2]:

> The semantics of the Modelica language is specified by means of a set of rules for translating any class described in the Modelica language to a flat Modelica structure. A class must have additional properties in order that its flat Modelica structure can be further transformed into a set of differential, algebraic and discrete equations (= flat hybrid DAE). Such classes are called simulation models.

[15] There has been research in that direction, *e.g.*, [Sal06, LZZ08], but so far no common agreement exists whether the Modelica language should be extended to support PDEs and how a such an extension should be done.

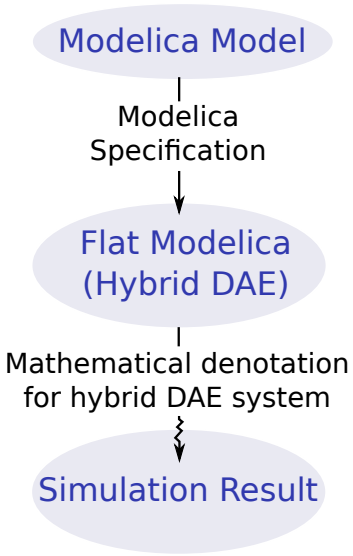


Figure 7: Modelica semantics. From model to simulation result. The squiggle arrow denotes a degree of fuzziness — a simulation result is an *approximation* to the in general inaccessible exact solution of the equation system and the specification does not prescribe a particular solution approach.

Figure 7 illustrates the basic idea. It suggests itself to discern the depicted two stages when giving semantics to Modelica models.

In practical Modelica compiler implementations, the translation to a flat Modelica structure is typically mapped to a "*front-end*" phase and the translation to an executable simulation to a "*back-end*" phase of the translation process. Figure 8 depicts a such a typical translation process that is classified into several stages.

Sometimes a dedicated "*middle-end*" phase is defined; it is responsible for the symbolic equation transformations performed during sorting and optimising the hybrid Differential Algebraic Equation (hybrid DAE) into a representation that can be efficiently solved by a numerical solver. In that case the "back-end" phase would only be responsible for code generation (typically C code) from the optimized sorted equations.

The following characterisation of the different stages is taken from [Thi15, p. 39]:

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) (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)

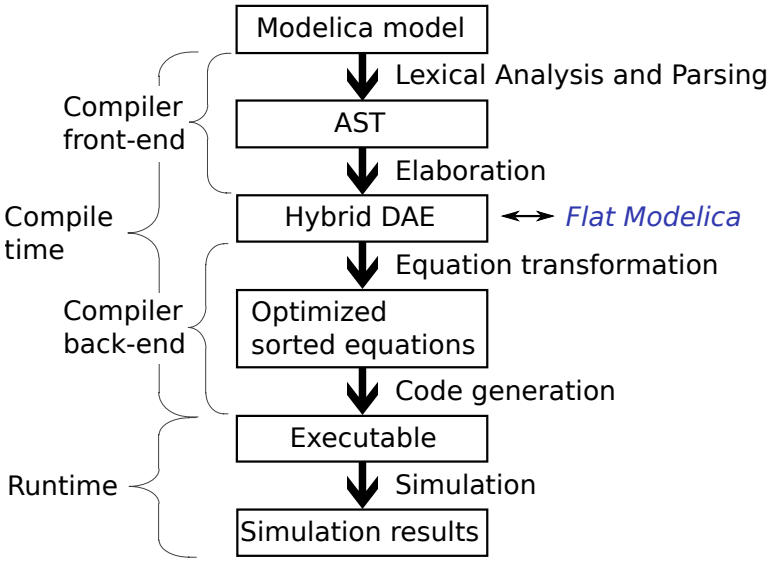


Figure 8: The typical stages of translating and executing a Modelica model.

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.

The Modelica Language Specification (MLS) is described using the English language, its semantics is therefore to some extent subject to interpretation. It appears also notable that the intermediate hybrid DAE representation (denoted as *Flat Modelica* in Figure 8) is not formally specified in the MLS, *i.e.*, no concrete syntax description, *e.g.*, in Extended Backus–Naur Form (EBNF), is provided (despite the conceptual importance of Flat Modelica). The translation from an instantiated Modelica model to a Flat Modelica representation is called *flattening* of Modelica models (hierarchical constructs are eliminated and a "flat" set of Modelica "statements" is produced).

The MLS [Mod14, Appendix C] discusses how a "flat" set of Modelica "statements" is mapped into an appropriate mathematical description form denoted as *Modelica DAE*. It is this mapping to a mathematical description that allows to associate dynamic semantics to the declarative Flat Modelica description. This will be further discussed in Section A.4.

A.3 Translation Approaches

This section gives a brief (non-exhaustive) account of previous (published) work related to translation and formal specification approaches of the Modelica language.

A.3.1 Compiler Generation from Operational Semantics Specifications

A first attempt to formalise aspects of the Modelica language was made at a time where no complete implementation was available and was performed by Kågedal and Fritzson to discover and

close gaps in the early specification documents [KF98]. Kågedal and Fritzson distinguish between *static semantics* (which describes how the object-oriented structuring of models and their equations work) and the *dynamic semantics* (which describes the simulation-time behaviour) of Modelica models. Like most later work on formalizing Modelica semantics the work by Kågedal and Fritzson [KF98] addresses the *static semantics* and does not intend to describe the equations solving process, nor the actual simulation. The formal semantics provided in their work is expressed in a high-level specification language called Relational Meta Language (RML) [Pet95], which is based on natural semantics (an operational semantics specification style). A compiler generation system allows generating a translator from a language specification written in RML. The paper explains the basic ideas behind that approach, but it does not list the complete RML source-code that was written during that early effort.

Development and usage of efficient language implementation generators has been a long-term research effort at the Programming Environments Laboratory (PELAB)[16] at Linköping University. In [FPBA09], Fritzson et al. report on practical experience gained with various approaches. The biggest effort was developing a complete Modelica compiler using RML as the specification language. This implementation formed the base for the *OpenModelica* environment [FAL+05]. A remarkable observation reported in the paper is the enormous growth of the RML code over time; we reproduce the data below (where we refer to lines of RML code, including comments).

Lines OM 1998	Lines OM 2003	Lines OM 2006
8709	36050	80064

Nowadays the development of OpenModelica has swapped from RML to *MetaModelica*. MetaModelica is a language developed at PELAB that introduces language modelling features known from languages like RML into the Modelica language. One of its development motivations was to provide a language with a more gentle learning curve than RML, particularly in regard to prospective OpenModelica developers without a background in formal specification methods. MetaModelica [PF06] has evolved over the years and, more recently, a bootstrapping version has been used in the OpenModelica development [SFP14].

A.3.2 Compiler Generation from Reference Attribute Grammars

Åkesson et al. report on leveraging modern attribute grammars mechanisms to support the *front-end* phase of a Modelica compiler [ÅEH10]. They use the attribute grammars metacompilation system JastAdd for implementing their Modelica compiler. The Modelica compiler of the open-source JModelica.org[17] platform is based on this technology.

A.3.3 Translational Semantics

A substantial extension in the MLS 3.3 was the integration of language elements motivated by synchronous languages like Lustre, Esterel, or Signal [BEH+03]. The synchronous-language paradigm has been particularly successful in the development of safety-critical control software like flight control systems. Thiele et al. [TKF15] discuss an approach that leverages the

[16] http://www.ida.liu.se/labs/pelab/ (Oct, 2015).
[17] http://www.jmodelica.org/ (Oct, 2015).

synchronous-language elements extension of Modelica to enable a MBD process for safety-related applications. For that, they aim at mapping a Modelica subset for digital control-function development to a well understood synchronous data-flow kernel language. That mapping is established by providing a *translational semantics* from the Modelica subset to the synchronous data-flow kernel language.

A.3.4 Summary

Most tools translate to C-code that is compiled and linked together with a numerical solver into an executable file. However, also interpretation of Modelica models is possible and is, *e.g.*, supported by the commercial SimulationX[18] tool.

Every Modelica tool has its specific approach of translating Modelica models. Several approaches on translating Modelica models can be found in the literature, however, one needs to be aware of the scopes and restrictions of the presented methods.

The translational semantics approach (Section A.3.3) is restricted to a discrete-time subset of the Modelica language, deemed suitable for safety-related applications. The subset is strongly restricted in order to lower tool qualification efforts for a code generator that is based on that subset (trade-off between language expressiveness and tool qualification efforts). There is no notion of differential equations within that language subset. Consequently, semantic models for continuous-modelling languages is not within the scope of that previous work.

Both the operational semantics approach based on RML/MetaModelica (Section A.3.1) and the reference attribute-grammar approach based on JastAdd (Section A.3.2) are targeted at describing the instantiation and flattening of Modelica models (*static semantics*). The result after that stage is basically the Flat Modelica representation (see Figure 8). In addition RML/MetaModelica is also used in the equation transformation ("middle-end") of the compiler. In both cases, the primary rationale of the formal description is generating compilers using language-implementation generators.

Neither the operational semantics approach nor the reference-attribute grammar approach targets the *dynamic semantics* of hybrid systems (*i.e.* the simulation-time behavior of dynamic systems that exhibit both continuous and discrete dynamic behavior) of Modelica models. It is the aim of the work started in [FTW15] to investigate the formalization of Modelica language aspects related to the dynamic semantics of hybrid system models.

The aim of this work is a translation validation and traceability concept from acausal hybrid models to generated code. Both, static and dynamic language semantics play an important role within this translation. Therefore, the next section will give a high-level overview over the dynamic semantics of Modelica models while following material will more specifically address aspects of the envisioned translation process.

A.4 Modelica Hybrid DAE Representation

[Mod14, Appendix C] discusses the mapping of a Modelica model into an appropriate mathematical description form. This section describes the principle form of the representation with-

[18] https://www.simulationx.com/ (Oct, 2015).

Symbol	Description
p	parameters and constants
$p^{\mathbf{B}}$	parameters and constants of type **Boolean**, $p^{\mathbf{B}} \subseteq p$
t	time
$x(t)$	vector of dynamic variables of type **Real**, *i.e.*, variables that appear differentiated at some place
$\dot{x}(t)$	differentiated vector of dynamic variables
$y(t)$	vector of other variables of type **Real** which do not fall into any other category (= algebraic variables)
$m(t_e)$	vector of discrete-time variables of type **discrete Real**, **Boolean**, **Integer**, **String**. Change only at event instants t_e
$m^{\mathbf{B}}(t_e)$	vector of discrete-time variables of type **Boolean**, $m^{\mathbf{B}}(t_e) \subseteq m(t_e)$. Change only at event instants t_e
$m_{\text{pre}}(t_e)$	values of m immediately before the current event at event instant t_e
$m^{\mathbf{B}}_{\text{pre}}(t_e)$	values of $m^{\mathbf{B}}$ immediately before the current event at event instant t_e, $m^{\mathbf{B}}_{\text{pre}}(t_e) \subseteq m_{\text{pre}}(t_e)$
$c(t_e)$	vector containing all Boolean condition expressions, *e.g.*, if-expressions
$v(t)$	vector containing all elements in the vectors $x(t)$, $\dot{x}(t)$, $y(t)$, $[t]$, $m(t_e)$, $m_{\text{pre}}(t_e)$, p

Table 3: Notation used in the Modelica hybrid DAE representation.

out aiming to cover every detail and special case[19]. Table 3 describes the symbols used in the mathematical description.

A.4.1 Equations

Flat Modelica can be conceptually mapped to a set of equations consisting of differential, algebraic and discrete equations of the following form (see Table 3 for brief description of the used symbols):

1. *Continuous-time behaviour.* The system behavior *between* events is described by a system of differential and algebraic equations (DAEs):

$$f\big(x(t), \dot{x}(t), y(t), t, m(t_e), m_{\text{pre}}(t_e), p, c(t_e)\big) = 0 \tag{1a}$$

$$g\big(x(t), y(t), t, m(t_e), m_{\text{pre}}(t_e), p, c(t_e)\big) = 0 \tag{1b}$$

2. *Discrete-time behaviour.* Behaviour at an event at time t_e. An event fires if any of condition $c(t_e)$ change from **false** to **true**. The vector-value function f_m specifies the Right-Hand Side (RHS) expression to the discrete variables $m(t_e)$. The argument $c(t_e)$ is made explicit for convenience (alternatively it could have been incorporated directly into f_m). The vector $c(t_e)$ is defined by the vector-value function f_e which contains all `Boolean`

[19] *E.g.*, the semantics of operators like `noEvent()`, or `reinit()` is not covered.

condition expressions evaluated at the most recent event t_e.

$$m(t_e) := f_m\big(x(t_e), \dot{x}(t_e), y(t_e), m_{\text{pre}}(t_e), p, c(t_e)\big) \tag{2}$$
$$c(t_e) := f_e\big(m^{\mathbf{B}}(t_e), m^{\mathbf{B}}_{\text{pre}}(t_e), p^{\mathbf{B}}, rel(v(t_e))\big) \tag{3}$$

where $rel(v(t_e)) = rel([x(t); \dot{x}(t); y(t); t; m(t_e); m_{\text{pre}}(t_e); p])$ is a Boolean-typed vector-valued function containing the relevant elementary relational expressions ("<", "<=", ">", ">=", "==", "<>") from the model, containing variables v_i, e.g., $v_1 > v_2,\ v_3 \geq 0$.

A.4.2 Simulation

Simulation means that an Initial Value Problem (IVP) is solved. The equations define a DAE which may have discontinuities, variable structure and/or which are controlled by a discrete-event system. Simulation is performed in the following way:

1. The DAE (1) is solved by a numerical integration method. Conditions c as well as the discrete variables m are kept constant. Therefore, (1) is a continuous function of continuous variables and the most basic requirement of numerical integrators is fulfilled.
2. During integration, all relations from (3) are monitored. If one of the relations changes its value an event is triggered, *i.e.*, the exact time instant of the change is determined and the integration is halted.
3. At an event instant, (1)–(3) is a mixed set of algebraic equations, which is solved for the Real, Boolean and Integer unknowns. New values of the discrete variables m and of new initial values for the states x are determined.
4. After an event is processed, the integration is restarted at 1.

There might have been discontinuous variable changes at an event that trigger another event. This case is described in the next section.

A.4.3 Event Iteration

At an event instant, including the initial event, the model equations are reinitialized according to the following iteration procedure:

known variables: $x,\ t,\ p$
unkown variables: $\dot{x},\ y,\ m,\ m_{\text{pre}},\ c$
// m_{pre} = value of m before event occured
loop
  solve (1)–(3) for the unknowns, with m_{pre} fixed
  if $m = m_{\text{pre}}$ **then break**
  $m_{\text{pre}} := m$
end loop

The iterative process of triggering events and solving the reinitialization problem is called *event iteration*. It is an example of a so-called fixed-point procedure, *i.e.*, the iterative process is assumed to converge to a fixed point.

A.4.4 Remarks on DAEs

DAEs are distinct from ODEs in that they are not completely solvable for the derivatives of all components of the function $x(t)$ because these may not all appear (*i.e.*, some equations are algebraic).

Consider following DAE given in the general form

$$f(x,\dot{x},y,t) = 0 \tag{4}$$

where $x = x(t), \dot{x} = \dot{x}(t) \in \mathbb{R}^n, y = y(t) \in \mathbb{R}^m, f : G \subseteq \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}^{n+m}$.

For a set of initial conditions $(x_0, \dot{x}_0, y_0, t_0) = 0$ to be consistent, it must satisfy system (4) at an initial time t_0:

$$f(x_0, \dot{x}_0, y_0, t_0) = 0. \tag{5}$$

The consistent initialization of such systems is the problem that Pantelides considered in his famous paper [Pan88].

Note that "initial conditions" here refers to the vector $(x_0, \dot{x}_0, y_0)$ rather than simply (x_0, y_0), hence elements of $\dot{x}_0$ may appear in (5). There is another complication as [Pan88] describes:

> Differentiating some or even all of the original equations produces new equations which must also be satisfied by the initial conditions. This need not necessarily constrain the vector $(x_0, \dot{x}_0, y_0)$ further: differentiation can also introduce new variables (time-derivatives of $\dot{x}$ and y) and it may well be the case that the new equations can be satisfied for all possible values of the initial conditions and appropriate choices of values for the new variables. Thus, in this case no useful information is generated by differentiation.

Pantelides then proposes an algorithm to analyze the structure of the system equations and determine the minimal subset for which differentiation may yield useful information in the sense that it imposes further constraints on the vector of initial conditions. This algorithm is not only useful for the initialization of a DAE system, but also for transforming the DAE system to a corresponding ODE system. The algorithm is further detailed below.

An important property for DAEs is the notion of the *DAE index*. There exist different definitions for DAE index in the literature. The following informal definition is from [Fri14, Section 18.2.2]. A more formal definition can be found in [AP98, Definition 9.1, p. 236].
Definition A.1. DAE index or differential index. The *differential index* of a general DAE system is the minimum number of times that certain equations in the system need to be differentiated to reduce the system to a set of ODEs, which can then be solved by the usual ODE solvers.

The DAE index can therefore be seen as measurement for the distance between a DAE and a corresponding ODE.

An ODE system in *explicit state-space form* is a DAE system of *index 0*:

$$\dot{x} = f(x,t) \tag{6}$$

The following semi-explicit form of DAE system:

$$\dot{x} = f(x,y,t) \tag{7a}$$
$$0 = g(x,y,t) \tag{7b}$$

is of *index 1* if $\frac{\partial g(x,y,t)}{\partial y}$ is non-singular, because then one differentiation of (7b) yields $\dot{y}$ in principle.

DAEs with an index > 1 are called *higher-index DAEs*.

Bachmann et al. [BAF06, Section 4.4] describe the typical approach in which higher-index DAEs are solved by a Modelica tool:

1. Use Pantelides algorithm to determine how many times each equation has to be differentiated to reduce the *index* to one or zero.
2. Perform *index reduction* of the DAE by analytic symbolic differentiation of certain equations and by applying the method of dummy derivatives [MS93]. The method of dummy derivatives for index reduction augments the DAE system with differentiated versions of equations, and replaces some of the differentiated variables with new algebraic variables called dummy derivatives [Fri14, Section 18.2.4.1].
3. Select the core state variables to be used for solving the reduced problem. These can either be selected statically during compilation, or in some cases selected dynamically during simulation.
4. Use a numeric ODE solver to solve the reduced problem.

A.5 Super-Dense Time

The FMI 2.0 standard for model exchange discusses a mathematical description for ODEs in state space form with event handling denoted as a "*hybrid ODE*" [FMI14, Section 3.1]. Modelica tools contain symbolic algorithms for DAE index reduction. These algorithms allow reduction of the DAE index in order to solve DAEs using numerical reliable methods and therefore also allow transform of the DAE formulation to an ODE formulation (transforming the DAE to an ODE for simulation is indeed the preferred method for some Modelica tools).

Studying the FMI description for hybrid ODEs is also interesting in the context of providing a dynamic semantics to Modelica models for the following reasons:

1. DAE index reduction methods allow transformation of a Modelica hybrid DAE to an FMI hybrid ODE.
2. It allows relation of the mathematical description used in the FMI standard [FMI14, Section 3.1] with the mathematical description from the Modelica standard [Mod14, Appendix C].

The mathematical description of FMI's Model Exchange interface [FMI14, Section 3.1] employs the concept of *super-dense time* for giving semantics to hybrid ODEs.

Definition A.2. Super-dense time The independent variable time $t \in \mathbb{T}$ is a tuple $t = (t_R, T_I)$ where $t_R \in \mathbb{R}, t_I \in \mathbb{N} = \{0, 1, 2, \ldots\}$, see *e.g.*, [LZ07].

Super-dense time provides a suitable formalism to reason about the dynamical evolution of hybrid system variables. Table 4 describes the ordering defined on super-dense time. Figure 9 depicts the concept graphically.

Operation	Mathematical meaning	Description
$t_1 < t_2$	$(t_{R1}, t_{I1}) < (t_{R2}, t_{I2}) \Leftrightarrow (t_{R1} < t_{R2}) \vee \big((t_{R1} = t_{R2}) \wedge (t_{I1} < t_{I2})\big)$	t_1 is before t_2
$t_1 = t_2$	$(t_{R1}, t_{I1}) = (t_{R2}, t_{I2}) \Leftrightarrow (t_{R1} = t_{R2}) \wedge (t_{I1} = t_{I2})$	t_1 is identical to t_2
t^+	$(t_R, t_I)^+ \Leftrightarrow (\lim_{\varepsilon \to 0}(t_R + \varepsilon), t_{Imax})$	right limit at t. t_{Imax} is the largest occurring Integer index of super dense time
^-t	$^-(t_R, t_I) \Leftrightarrow (\lim_{\varepsilon \to 0}(t_R - \varepsilon), 0)$	left limit at t
$^\bullet t$	$^\bullet(t_R, t_I) \Leftrightarrow \begin{cases} ^-t & \text{if } t_I = 0 \\ (t_R, t_I - 1) & \text{if } t_I > 0 \end{cases}$	previous time instant (= either left limit or previous event instant)
v^+	$v(t^+)$	value at the right limit of t
^-v	$v(^-t)$	value at the left limit of t
$^\bullet v$	$v(^\bullet t)$	previous value (= either left limit or value from the previous event)

Table 4: Ordering defined on super-dense time $\mathbb{T}$.

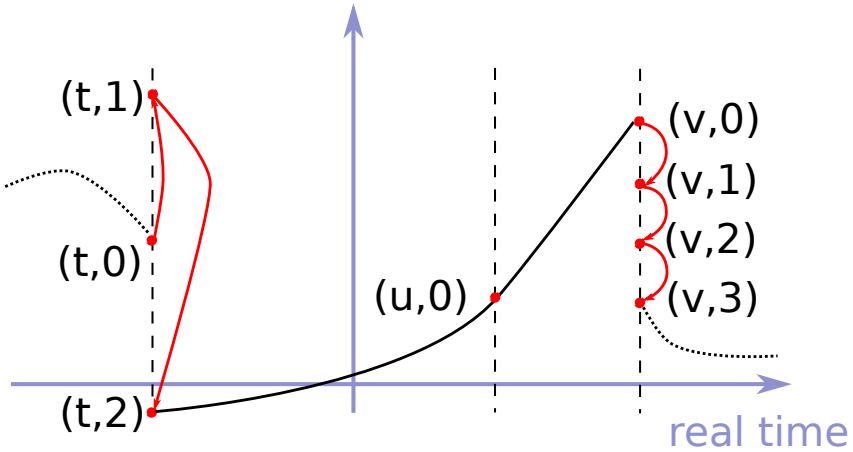


Figure 9: Super-dense time modelling $\mathbb{T} = \mathbb{R} \times \mathbb{N}$.

In particular, note that super-dense time allows signals to have an ordered sequence of values at the same time instant. This is important since a time model based on the set of real numbers ($t \in \mathbb{R}$) is not semantically rich enough to capture an ordered sequence of signal values during an event iteration as described in Section A.4.3.

B Requirements for Model-Based Function Development

This section is a copy of Chapter 3 "Requirements for Model-Based Function Development" from

Bernhard Thiele. *Framework for Modelica Based Function Development*. Dissertation, Technische Universität München, 2015. http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20150902-1249772-1-2.

which has been slightly adapted to integrate into the format of this report and is included in this report for convenience.

This chapter is an extensively revised and extended version of Section 2 and 3 of the publication

> Bernhard Thiele, Stefan-Alexander Schneider, and Pierre R. Mai. A Modelica Sub- and Superset for Safety-Relevant Control Applications. In 9th *Int. Modelica Conference*, Munich, Germany, September 2012. http://dx.doi.org/10.3384/ecp12076455.

An increasing number of embedded software components is specified in models representing the so-called *high-level application* which is then automatically transformed into embedded target code (Figure 2). The specification model is designed using a high-level, domain-oriented language which is in the following succinctly referred to as the *modeling language*. In this chapter, I try to identify requirements imposed on such a language with a special attention to safety requirements, which are indispensable for many real-world applications.

The modeling language (and the software development process) has to provide a balance between rigidness and flexibility: on the one hand, the user may not be restricted too much and must still have room for creativity and principal control over all development activities since too many restrictions reduce the acceptance, 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.

In order to understand the different requirements imposed in this chapter, it is beneficial to consider the various stake-holders which participate in the development in different roles with different requirements and expectations. Consequently, this chapter starts with introducing the most relevant roles within the intended development process.

In the following sections, the requirements resulting from the introduced roles are developed. An essential implication of these requirements is that they only can be met by specific restrictions and extensions of the domain language Modelica considered in this work. This will be elaborated upon in following chapters.

Another important aspect is the intention to allow for *code reviews being done entirely on model level* (as opposed to embedded C source code level). Code reviews are a well established activity in development processes targeting safety relevant software (see, e.g., [SCFD06, SWC05, The94, ISO11]). A model-based development process may require that automatically generated code is treated with the same scrutiny as handcrafted code, in particular performing code

reviews on automatically generated code may be necessary [SCFD06, BOJ04]. However, in order to fully benefit from a model-driven approach (and thereby reducing costs) it is desirable to make C code level reviews redundant and perform solely *model reviews*. This becomes feasible as soon as a *suitably qualified/certified toolchain* is available [SLM09, BOJ04].

For some modeling languages, the language semantics is entirely defined on the graphical level (e.g., Simulink). However, for Modelica, the modeling language considered in this work, the language specification defines the semantics on the textual level and defines *annotations* for storing extra information like the graphical representation of the model. As a consequence, the requirements at the modeling language have been split into two parts: Section B.2 defines general requirements for a modeling language deemed suitable for model-based (control) function development. In addition, Section B.3 introduces requirements that are targeted specifically to the graphical representation of modeling languages whose semantics is originally defined at the textual level.

B.1 Development Roles

Various stake-holders participate in the development having different roles with different requirements and expectations. A suitable domain-oriented language will have to support at least the following roles in the development process [TSM12, Section 2]:

Role 1 - Developer. Developer of the embedded control system. This role requires a sufficiently expressive modeling language based on 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. 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 sufficiently 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 see [Thi15, Section 5.2.1].

These different roles have partly coincident (semantic aspects) and partly contradictory (expressiveness of the language) requirements to the modeling language and its associated development tools.

B.2 Modeling Language Requirements

This section establishes general requirements for modeling languages that shall be used as base for a high-level application model. The requirements are to some extent subjective, however a clear rationale is provided for any stated requirement.

It is assumed that the modeling language shall be used for high-level applications that have an open- or closed-loop control nature and provide potentially safety-relevant functionality. Due to this safety aspect, it is important that the language supports high assurance designs. Therefore, the feasibility of tool qualification is of high importance and similarly the suitability for efficient and effective model reviews.

For modeling languages whose semantics are specified on the textual level, it is, at first glance, natural to do the functional reviews on the textual level. Even if a graphical representation is also available in some form, (as is the case for Modelica) the graphical level may hide important implementation 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. Beyond this, it is as well desirable to allow reviews to be done *entirely* on the graphical level. For this purpose Section B.3 provides additional requirements with the aim to ensure that the graphical representation is clear without ambiguity. It is then left to the reviewer and his or her preferences to either perform a textual or a graphical review.

As has been explained in [Thi15, Section 2.3], it is of paramount importance in the model-based development of complex (cyber-physical) systems that also the physical system parts can be adequately modeled. However, the requirements for modeling the physical system parts are not considered in this section. This section solely targets the language requirements in respect to the high-level application model. Nevertheless, the modeling language for the high-level application should integrate seamlessly into system models that contain physical system parts. This is reflected in the "zeroth requirement":

> *Requirement 0 - Support for System Design and Early Problem Solving.* The language should integrate seamlessly into a model-based system design approach in order to support early problem solving and validation and verification activities.

The following requirements are specifically targeted to properties that need to be met by a *modeling language for the high-level application model*. Due to its importance to all roles, the first requirement is of particular importance:

Requirement 1 - Formally Sound Language Set. The language must be formally sound.
Rationale: Ambiguity and impreciseness in the language must be eliminated in order to avoid different interpretation possibilities. The language should be well suited to support (formal) validation and verification activities.
Mainly demanded by: All roles.

Requirement 2 - High-Level Domain Oriented Notation. The language must have enough expressiveness to allow the clear and concise specification of discrete open- and closed-loop control algorithms and their related support logic.
Rationale: Embedded control developers need a language that provides good support for implementing relevant control algorithms.
Mainly demanded by: Developer.

Requirement 3 - Suitable Model of Computation (MoC). The utilized MoC behind the language should be clearly defined, intuitive to the control system developer, and well-understood and accepted by (certification) authorities.
Rationale: MoCs provide mental models to express, understand, discuss and analyze computational execution (cf. [LSV98, EJL+03]). Depending on the problem at hand, a particular MoC

might result in a superior abstraction than another. On the one hand the utilized MoC should be intuitive to control system developers, on the other hand it should be well-understood and accepted by (certification) authorities to facilitate tool qualification activities.
Mainly demanded by: Developer and Tool Qualifier.

Requirement 4 - 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.
Rationale: Low-level hardware-aspects play an important role for dedicated code generators and high-level languages targeting microcontrollers or digital signal processors, which strive to produce optimized, target-aware code [HKK+99, HRW+99, Erk09]. The major motivation is to lower costs of required hardware by optimizing the memory and runtime efficiency of the software. Despite of optimizing code generators developers may still need full manual control over data types, data storage, memory alignment and the implementation of interfaces in the generated code in order to optimize the code generated for a particular target [Rau02, p. 78].
Mainly demanded by: Developer and Tool Developer.

Requirement 5 - Compile Time Analysis. The language should allow compile time analysis of important properties in order to reject problematic models.
Rationale: Compile time checks are means to increase the degree of confidence that one may have in the correctness of a model/program. Possible properties that can be checked by compile time analysis include: missing/incompatible initial values, type checking, clock analysis, cyclic definitions that result in algebraic loops, and equality of the number of equations with the number of unknown variables.
Mainly demanded by: Developer and Reviewer.

Requirement 6 - Modularity. The language must support constructs that allow modular modeling.
Rationale: Modularisation is a key technique to cope with the complexity of software. It improves understandability and thus reviewability of complex models. It also facilitates its implementation and maintenance.
Mainly demanded by: Developer, Reviewer and Tool Qualifier.

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 7 - Restricted Language Scope. The language should be as simple and clear as possible. This shall be achieved by restricting the scope of the language to a (preferably small) core relevant for the addressed problem domain. Particularly, simplicity and clarity of the language is to be preferred over feature richness.
Rationale: Facilitate tool qualification activities.
Mainly demanded by: Tool Qualifier.

The next requirements concern the interplay between automatic code generation and the modeling language. It could be argued that requirements on the code generation is orthogonal to the modeling language. However, this is not entirely true since semantics of the modeling language needs to be faithfully reproduced by the generated code. Thus the modeling input language will naturally affect the target code structure that can be automatically generated from it. Therefore, the formulated requirements do have impacts on the modeling language.

Requirement 8 - Automatic Code Generation (ACG). The language should permit automatic generation of target platform C-code that is: a) efficient, b) adheres to good software

engineering practice, c) is traceable, and d) integrates smoothly into embedded systems software architectures.
Rationale: Automatic target code generation is crucial to optimize the benefits gained from a model-based development process (see [Thi15, Section 2.3]). Automatic C-code generation is stipulated, since C-code is the most popular language for targeting embedded systems and (certifiable) compilers are available. However, this should not be understood as if the use of other target languages or the direct generation of binaries was inferior. *Efficiency* is a natural requirement for code that is meant to run on a cost-efficient embedded system. For safety-related applications it is often required that also generated code "*adheres to good software engineering practice and follows the standard styles for the target language*" [WH99]. For generated C-code in the automotive area that typically means that conformance to some coding standards, e.g., MISRA AC AGC [The07] is required. *Traceability* refers to the property that 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. Traceable code facilitates validation activities, e.g., manual inspections, automated analysis of the generated code[20], and testing [WH99]. Finally, the generated code typically needs to be *integrated into a given software architecture* (see [Thi15, Section 2.6]).
Mainly demanded by: Developer, Tool Developer and Tool Qualifier.

Requirement 9 - Tangible Fixation of Automatically Deduced Properties. It must be possible to fixate all properties of a model that influence code generation in a tangible, reviewable form.
Rationale: In order to ensure reproducibility of code generation and reviewability[21], 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.
Mainly demanded by: Reviewer and Tool Qualifier.

Requirement 10 - Modular Code Generation. The language should support modular code generation. Modular code generation in this context is understood as: (1) code for a modular structure in the modeling language should be generated *independently from the context* in which that structure is used, and (2) if a modular structure is composed of several other modular structures, only minimal knowledge about that structures (their respective *interface information*) should be needed. E.g., presume the language has a structuring construct similar to the blocks typically encountered in the block diagrams used by control engineers (i.e., hierarchical synchronous data-flow block diagrams). Further on, presume such a block is composed by connecting several "building" blocks. For modular code generation it should then be possible to generate a minimal set of transition functions (preferably one) for each of the respective building block definitions and produce the overall transition function(s) by their composition.
Rationale: Imperative languages (like C) use functional decomposition as a means for modularization, abstraction and structuring. Modular code generation allows to map modular structures at the model level to modular structures at the target code level. Although modular code generation should not be considered as an axiomatic requirement to a code generator for safety-relevant software it offers several advantages (see also [Thi15, Section 5.6]):

[20] One example is to perform code coverage analysis on the target level but mirror back the results onto the model level.

[21] 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.

1. Modular code generation preserves structure. Given that generated code is well-structured and obeys similar guidelines as handcrafted code, generated code may be treated just like handcrafted code artifacts in the further development process. In particular, that allows to directly reuse the validation and verification methods and criteria that are already accepted for handcrafted code. Note that this can be an attractive approach, especially if the code generator itself is not (yet) sufficiently qualified for the intended purpose at hand and qualification would be too costly.
2. Modular code generation helps to establish a good traceability between model and generated code (supports Requirement 8, *traceability*).
3. Modular code generation may decrease the size of the generated code, since generated transition functions can be potentially reused at different parts of the model (supports Requirement 8, *efficiency*).
4. Modular code generation allows for *separate compilation* of the generated code artifacts which has two advantages: First, it allows to distribute that modules without giving away the source code (in order to protect intellectual property) and second, it avoids processing all source code every time the binary is built which in turn saves development time.

Mainly demanded by: Developer, Tool Developer and Tool Qualifier.

B.3 Graphical Representation Requirements

For a modeling language which is specified at the textual level (like Modelica) a suitable graphical representation often provides an abstraction that significantly eases the comprehension of the intended model semantics. Therefore, a model review done on the graphical level can be potentially more efficient than one done at the textual level (since the relevant details can be understood faster).

However, if the semantics of a modeling language is described on a textual level, but there also exists a graphical representation the following question arises:

> *How can it be avoided that details that are not obvious or even hidden in the graphical representation prevent the reviewers from performing an efficient and effective graphical review?*

For establishing an effective graphical code review it has to be ensured that the full semantics of the textual representation is also available in the graphical representation. This motivates the following additional requirement for the graphical representation that should allow developers and reviewers *to mostly work at a graphical level*:

Requirement 11 - Encoding of Full Semantics in the Graphical Representation. Any semantics associated with the graphical representation of language elements 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 graphical diagram.
Rationale: Code reviews of models should be feasible as far as possible at the graphical level. Consequently, the semantics of a model should be completely evident by inspecting the graphical diagram layer.
Mainly demanded by: Reviewer.

Note, that for Modelica it is typical that a *library* developer uses the textual Modelica language to code basic functionality 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. Therefore, there is no direct correspondence between basic textual language elements and the graphical representation. For such languages, requirements to the language design translate into requirements to the library design.

The first of the additional requirements for the design of such libraries assures that the graphical representation of *library components* is close to notations typically used by control engineers. It is therefore analog to Requirement 2.

Requirement 12 - Intuitive Graphical Representation. Library components and their (graphical) compositions should not exhibit any behavior which would be deemed surprising or non-obvious by embedded control systems experts.
Rationale: In order to minimize the risk of misunderstanding it is important that the graphical representation is intuitive for domain experts.
Mainly demanded by: Developer and Reviewer.

The restriction on allowed basic library components reflects the textual requirement 7:

Requirement 13 - Restricted Set of Allowed Basic Library Components. A high-level application model is only allowed to be composed from a set of *approved* (thoroughly tested and validated) *basic library components* whose semantics have been completely specified.
Rationale: In order to allow a review on the graphical level the full semantics of all the used library components must be unambiguously specified. Hence, the set of allowed basic components needs to be restricted and it needs to be ensured that their behavior conforms to their specified semantics. Note that this does not automatically rule out that developers design additional components using the underlying textual language — this just has the consequence that the review for such components must be done on the textual level.
Mainly demanded by: Reviewer and Tool Qualifier.

The last requirement is not restricted to graphical modeling but applies equally to modeling on the textual level. It can be seen as a strategy to meet some of the requirements formulated above (and therefore as already being a part of the solution space rather than belonging into the requirement section), but at the same time it can be seen as a requirement on its own. It is presented here as distinct requirement, because, in contrast to the already presented requirements, it calls for the *establishment of rules* on how an "approved" language is to be *used by developers*, rather than "solely" guiding the design of the language or the design of the "approved basic library components". This is an important aspect, particularly for allowing effective model reviews.

Requirement 14 - Adaption of Modeling Guidelines. Suitable modeling guidelines should be devised and adhered to.
Rationale: The adoption of modeling guidelines or coding guidelines is highly recommended in safety-related development projects (e.g., [ISO11]). Stürmer et al. [SWC05] list the following advantages of modeling guidelines: (1) increase of comprehensibility (readability), (2) maintainability, (3) reusability and extensibility, and (4) ease of testing.
Mainly demanded by: Developer and Reviewer.

References

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

[ÅEH10] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Implementation of a Modelica compiler using JastAdd attribute grammars. *Science of Computer Programming*, 75(1–2):21–38, 2010. Special Issue on ETAPS 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA '06 and '07). doi:10.1016/j.scico.2009.07.003.

[AP98] Uri M. Ascher and Linda R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM, Philadelphia, 1998.

[BAF06] Bernhard Bachmann, Peter Aronsson, and Peter Fritzson. Robust initialization of differential algebraic equation. In 5th *Int. Modelica Conference*, Vienna, Austria, September 2006. URL: https://www.modelica.org/events/modelica2006/Proceedings/sessions/Session6a2.pdf.

[BEH+03] 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. doi:10.1109/JPROC.2002.805826.

[BOJ04] M. Beine, R. Otterbach, and M. Jungmann. Development of Safety-Critical Software Using Automatic Code Generation. In *SAE World Congress*, 2004.

[Bro10] 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. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-58743.

[Con09] Mirko Conrad. Testing-based translation validation of generated code in the context of IEC 61508. *Formal Methods in System Design*, 35(3):389–401, 2009. doi:10.1007/s10703-009-0082-0.

[CSW05] Mirko Conrad, Sadegh Sadeghipour, and Hans-Werner Wiesbrock. Automatic Evaluation of ECU Software Tests. In *SAE 2005 Transactions, Journal of Passenger Cars—Mechanical Systems*. SAE International, March 2005. doi:10.4271/2005-01-1659.

[EJL+03] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, J.Xiaojun Liu, Joszef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127 – 144, jan. 2003. doi:10.1109/JPROC.2002.805829.

[Erk09] T. Erkinnen. Fixed-Point ECU Code Optimization and Verification with Model-Based Design. In *SAE 2009 World Congress & Exhibition*, Detroit, Michigan, United States, April 2009. SAE Technical Paper 2009-01-0269. doi:10.4271/2009-01-0269.

[FAL+05] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Software Development Environment. *Simulation News Europe*, 44(45), December 2005.

[FGH+08] Stephan Frank, Martin Grabmüller, Petra Hofstedt, Dirk Kleeblatt, Peter Pepper, Pierre R. Mai, and Stefan-Alexander Schneider. Safety of Compilers and Translation Techniques – Status quo of Technology and Science. In *Automotive – Safety & Security*, 2008.

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

[FPBA09] P. Fritzson, A. Pop, D. Broman, and P. Aronsson. Formal Semantics Based Translator Generation and Tool Development in Practice. In *Software Engineering Conference, 2009. ASWEC '09. Australian*, pages 256–266, April 2009. doi:10.1109/ASWEC.2009.46.

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

[FTW15] Simon Foster, Bernhard Thiele, and Jim Woodcock. Differential Equations in the Unifying Theories of Programming. Technical Note D2.1c, INTO-CPS project, HORIZON 2020, Grant Agreement 644047, December 2015. URL: http://into-cps.au.dk/publications/.

[HKK+99] H. Hanselmann, U. Kiffmeier, L. Koster, M. Meyer, and A. Rukgauer. Production quality code generation from simulink block diagrams. In *Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design*, pages 213–218, Kohala Coast-Island of Hawai'i, Hawai'i, USA, August 1999. IEEE. doi:10.1109/CACSD.1999.808650.

[HRW+99] U. Honekamp, J. Reidel, K. Werther, T. Zurawka, and T. Beck. Component-node-network: three levels of optimized code generation with ASCET-SD. In *Computer Aided Control System Design, 1999. Proceedings of the 1999 IEEE International Symposium on*, pages 243–248, 1999. doi:10.1109/CACSD.1999.808655.

[HWK+14] John Hatcliff, Alan Wassyng, Tim Kelly, Cyrille Comar, and Paul Jones. Certifiably Safe Software-Dependent Systems: Challenges and Directions. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, Hyderabad, India, May 31 – June 7 2014. ACM. doi:10.1145/2593882.2593895.

[IEC00] IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related systems, Parts 1 through 7, Edition 1.0, 1998–2000.

[ISO11] ISO 26262-6:2011, Road vehicles – Functional safety – Part 6: Product development at the software level, November 2011.

[KF98] David Kågedal and Peter Fritzson. Generating a Modelica compiler from natural semantics specifications. In *Proceedings of the 1998 Summer Computer Simulation Conference (SCSC'98)*, 1998.

[LSV98] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on CAD*, 17(12):1217–1229, December 1998.

[LZ07] Edward A. Lee and Haiyang Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 114–123, New York, NY, USA, 2007. ACM. doi:10.1145/1289927.1289949.

[LZZ08] Zhihua Li, Ling Zheng, and Huili Zhang. Solving PDE models in Modelica. In *Information Science and Engineering, 2008. ISISE'08. International Symposium on Information Science and Engieering. ISISE*, volume 1, pages 53–57. IEEE, 2008.

[Mod14] Modelica Association. Modelica - A Unified Object-Oriented Language for Systems Modeling - Version 3.3 Revision 1. Standard Specification, July 2014. URL: http://www.modelica.org/.

[MS93] Sven Erik Mattsson and Gustaf Söderlind. Index Reduction in Differential-Algebraic Equations Using Dummy Derivatives. *SIAM Journal on Scientific Computing*, 14(3):677–692, 1993. doi:10.1137/0914043.

[Pan88] Constantinos C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231, 1988. doi:10.1137/0909014.

[Pet95] Mikael Pettersson. *Compiling Natural Semantics*. Doctoral thesis No 413, Department of Computer and Information Science, Linköping University, Sweden, 1995.

[PF06] Adrian Pop and Peter Fritzson. Metamodelica: A unified equation-based semantical and mathematical modeling language. In DavidE. Lightfoot and Clemens Szyperski, editors, *Modular Programming Languages*, volume 4228 of *Lecture Notes in Computer Science*, pages 211–229. Springer Berlin Heidelberg, 2006. doi:10.1007/11860990_14.

[PSA+14] Adrian Pop, Martin Sjölund, Adeel Ashgar, Peter Fritzson, and Francesco Casella. Integrated Debugging of Modelica Models. *Modeling, Identification and Control*, 35(2):93–107, 2014. doi:10.4173/mic.2014.2.3.

[PSS98] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'98*, pages 151–166, Lisbon, Portugal, April 1998.

[Rau02] Andreas Rau. *Modellbasierte Entwicklung von eingebetteten Regelungssystemen in der Automobilindustrie*. PhD thesis, Fakultät für Informatik, Eberhard-Karls-Universität Tübingen, 2002.

[Sal06] Levon Saldamli. *PDEModelica - A High-Level Language for Modeling with Partial Differential Equations*. Doctoral thesis No 1016, Department of Computer and Information Science, Linköping University, Sweden, May 2006. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-7281.

[SCFD06] Ingo Stürmer, Mirko Conrad, Ines Fey, and Heiko Dörr. 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, pages 45–52, New York, NY, USA, 2006. ACM. doi:10.1145/1138474.1138483.

[SFP14] Martin Sjölund, Peter Fritzson, and Adrian Pop. Bootstrapping a Compiler for an Equation-Based Object-Oriented Language. *Modeling, Identification and Control*, 35(1):1–19, 2014. doi:10.4173/mic.2014.1.1.

[Sjö15] Martin Sjölund. *Tools and Methods for Analysis, Debugging, and Performance Improvement of Equation-Based Models*. Doctoral thesis No 1664, Linköping University, Department of Computer and Information Science, 2015. doi:10.3384/diss.diva-116346.

[SLM09] Stefan-Alexander Schneider, Tomislav Lovric, and Pierre R. Mai. The Validation Suite Approach to Safety Qualification of Tools. In *SAE World Congress*, Detroit, MI, USA, April 2009. SAE International. doi:10.4271/2009-01-0746.

[SWC05] Ingo Stürmer, Daniela Weinberg, and Mirko Conrad. Overview of existing safeguarding techniques for automatically generated code. In *Proceedings of the Second International Workshop on Software Engineering for Automotive Systems*, SEAS '05, pages 1–6, New York, NY, USA, 2005. ACM. doi:10.1145/1082983.1083192.

[TB16] Bernhard Thiele and François Beaude. Concept for customizing code generation including flexible adaptation to different target systems. Technical Note D3.3, OPENCPS project, ITEA3, Project 14018, December 2016.

[The94] The Motor Industry Software Reliability Association. Development Guidelines for Vehicle Based Software, 1994. http://www.misra.org.uk. URL: http://www.misra.org.uk.

[The04] 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.

[The07] 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.

[Thi15] Bernhard Thiele. *Framework for Modelica Based Function Development*. Dissertation, Technische Universität München, 2015. URL: http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20150902-1249772-1-2.

[TKF15] Bernhard Thiele, Alois Knoll, and Peter Fritzson. Towards Qualifiable Code Generation from a Clocked Synchronous Subset of Modelica. *Modeling, Identification and Control*, 36(1):23–52, 2015. doi:10.4173/mic.2015.1.3.

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

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

[TSM12] Bernhard Thiele, Stefan-Alexander Schneider, and Pierre R. Mai. A Modelica Sub- and Superset for Safety-Relevant Control Applications. In Martin Otter and Dirk Zimmer, editors, 9th *Int. Modelica Conference*, Munich, Germany, September 2012. doi:10.3384/ecp12076455.

[WH99] M.W. Whalen and M.P.E. Heimdahl. On the requirements of high-integrity code generation. In *High-Assurance Systems Engineering, 1999. Proceedings. 4th IEEE International Symposium on*, pages 217–224, 1999. doi:10.1109/HASE.1999.809497.