TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Echtzeitsysteme und Robotik

Framework for Modelica Based Function Development

Bernhard Amadeus Thiele

Vollständiger Abdruck der von der Fakultät der Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. rer. nat. Alexander Pretschner
Prüfer der Dissertation:	1. Univ.-Prof. Dr.-Ing. Alois Knoll
	2. Hon.-Prof. Dr.-Ing. Martin Otter
	3. Prof. Marc Pouzet, Université Pierre et Marie Curie in the Computer Science Department (DIENS) of École normale supérieure (ENS), Paris/Frankreich

Die Dissertation wurde am 15.04.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 03.08.2015 angenommen.

Zusammenfassung

Rasante Fortschritte in der Leistungsfähigkeit von elektronischen Steuerungsgeräten führen zu immer umfangreicheren Softwarefunktionen, die innerhalb des Entwicklungsprozesses beherrscht werden müssen. Eine besondere Herausforderung sind hierbei auf vernetzte Recheneinheiten verteilte Softwarefunktionen, welche in Rückkopplung mit physikalischen Prozessen stehen.

Für die Entwicklung und Validierung derartiger Funktionen ist es nicht länger ausreichend, weitgehend autarke Einzelkomponenten zu betrachten, ohne die wechselseitige Beeinflussung innerhalb des Gesamtsystems (einschließlich der physikalischen Prozesse) zu berücksichtigen. Aufgrund von unterschiedlichen wissenschaftlichen Gemeinschaften hat sich jedoch der Stand der Technik für Modellierungssprachen im Bereich physikalischer Systeme sehr unterschiedlich zum Stand der Technik im Bereich der Modellierung von Software entwickelt. Dies erschwert ganzheitliche Entwicklungsansätze.

Diese Arbeit befasst sich mit der Entwicklung einer Methodik, welche eine verbesserte Integration der zwei Welten auf den jeweils aktuellen Stand der Technik ermöglicht: Die Modellierung von (sicherheitsrelevanten) *digitalen steuerungs- und regelungstechnischen Funktionen* und die Modellierung von *physikalischen Multidomänen-Systemen*. Als Basis dient die primär für physikalische Systeme entwickelte Modellierungssprache Modelica, welche auf folgende Aspekte der Funktionsentwicklung hin untersucht wird: a) Die Voraussetzungen, die erfüllt sein müssen, um *Funktionsmodelle* innerhalb eines Entwicklungsprozesses für sicherheitsrelevante Systeme implementieren zu können, und b) ihre Nutzung für eine durchgängige Entwicklungsmethodik, angefangen von der *logischen Systemarchitektur* bis herunter auf die *technische Systemarchitektur*.

In einem ersten Schritt werden allgemeine Anforderungen an eine domänenorientierte Modellierungssprache für die sicherheitsrelevante modellbasierte Funktionsentwicklung formuliert. Im zweiten Schritt wird Modelica im Hinblick auf die formulierten Anforderungen analysiert und bestehende Defizite werden identifiziert. Auf diese Analyse aufbauend, werden auf die Modellierung von digitalen steuerungs- und regelungstechnischen Funktionen zugeschnittene Spracheinschränkungen und -erweiterungen vorgeschlagen. Die dadurch spezifizierte Sprache wird als ACG-Modelica bezeichnet und als geeignete Basis für die Implementierung hochwertiger Seriencode-Generatoren angesehen. Im dritten Schritt wird dies durch eine Realisierbarkeitsstudie belegt, in deren Mittelpunkt die Angabe einer translatorischen Semantik steht, welche eine repräsentative Teilmenge von ACG-Modelica in eine ausgewählte Zielsprache transformiert.

Abstract

Electronic control technology is evolving leading to more complex software functions that need to be managed efficiently within a development process. To manage the complexity of the various stages of development, model-based function development has become a standard approach. A particular challenge are software functions running on distributed, networked computing devices which are in feedback loops with physical processes.

For the development and validation of these functions it is no longer sufficient to solely consider a single, self-contained component without taking the interaction of this part within the whole system (including the physics!) into account. However, due to different communities, state-of-the-art modeling languages for software functions have evolved very differently from modeling languages targeted for physical systems. This impedes integrated holistic modeling approaches.

This thesis concerns the problem of a development methodology that is capable of integrating the two worlds: State-of-the-art modeling for (safety-related) digital control functions and state-of-the-art modeling for multi-domain physical systems. Based on the physical modeling language Modelica the thesis addresses two aspects of function development: a) the conditions that need to be fulfilled to allow the usage of Modelica for implementing *function models* in a safety-related development process with automatic code generation, and b) the use of Modelica as a basis for a seamless development methodology starting from the *logical virtual model* down to the *technical system architecture*.

In a first step requirements to a high-level, domain-oriented language targeted for safety-related, model-based function development are established. Second, Modelica is analyzed with respect to the established requirements and present shortcomings are identified. Based on this, language restrictions and extensions are proposed, effectively leading to a language sub- and superset for discrete-time controller function modeling, which is deemed suitable as basis for a high-integrity automatic code generator (ACG). Third, the feasibility of implementing such an ACG is demonstrated.

Acknowledgements

This work would not have been possible without the support of many people to whom I am deeply indebted. First I would like to thank my supervisor Prof. Alois Knoll for accepting me as an external PhD student and for giving me the opportunity to write this thesis. I highly appreciate his valuable feedback, ideas, and guidance that supported my work. I would also like to thank Prof. Marc Pouzet for accepting to be my external reviewer.

Most of this work originates from my time as research scientist at the German Aerospace Center (German: Deutsches Zentrum für Luft- und Raumfahrt e.V., abbreviated DLR) at the Institute of System Dynamics and Control and I am indebted to the director of the institute, Dr. Johann Bals for the opportunity to work on this thesis. I am particularly indebted to Prof. Martin Otter and Dr. Dirk Zimmer who supervised my thesis work at DLR and provided indispensable and comprehensive feedback, support and encouragement that kept me going on. Additionally I would like to thank my colleagues at DLR for the many inspiring discussions and the good laughs that we had together, particularly my long-time office colleague Dr. Tobias Bellmann with whom I shared a passion for Modelica, nifty programming solutions and LaTeX.

I would also like to warmly thank Prof. Peter Fritzson who is leading the group that develops the OpenModelica compiler at the Programming Environment Laboratory (PELAB) at Linköping University. He gave me the opportunity to join his group after my affiliation to DLR ended and supported my pursuit to finalize this thesis.

As a researcher I was part of the Modelica Community and I would like to thank the members of this community for the friendly atmosphere and the good discussions and inspiring ideas during conferences and during the Modelica design meetings that I attended.

Last but not least, I would like to thank my friends and my family for supporting me and cheering me up at appropriate times. Thanks to Thomas van Marwick for giving me insight into the use of the Ada programming language for safety-relevant systems. Thanks to my mother, Anna Thiele, for proofreading my thesis in order to spot English language mistakes. And finally, my warmest thanks go to Mihaela who stood by my side at good and bad days and who gave me all support and love that makes life so much more worth living.

Contents

1. Introduction

1.1. Motivation

Electronic control technology is evolving leading to more complex software functions that need to be managed efficiently within a development process. For example, within the automotive industry the following trends are visible:

- More functions are integrated into a single Electronic Control Unit (ECU) to reduce the overall number of ECUs per vehicle.
- Complex functions are distributed on several networked ECUs.
- Software functions provide safety-related functionality.

For the development and validation of these functions, it is no longer sufficient to solely consider a single, self-contained component without taking the interaction of this part within the whole vehicle system into account. A more global approach is needed for efficient development and validation of such functions.

Various tools and methodologies exist for supporting the development of embedded systems (freely available or as commercial product). However, they usually focus on improving the *coding and debugging* process for single, self-contained entities (i.e., a single ECU). The interaction of the ECU within a complex environment consisting of physical components with highly nonlinear behavior and other networked control systems is not within the scope of these tools.

On the other hand, the modeling and simulation (M&S) community has improved their methodology and tools to efficiently handle complex multi-domain problems by supporting a modeling style that allows to plug models of system components together in the same way, as the real components are being assembled in manufacturing plants. The paradigm that drove this development is termed *object-oriented physical system modeling*.

Of great significance within that M&S community was the forming of an international standard committee (the Modelica Association) in the late 1990s. This group designed a new physical modeling language (called Modelica) that had the goal to unify concepts and notations used by diverse (research) groups to define object-oriented languages for physical modeling.

Since that time, the Modelica language has evolved and proved to be a powerful tool to model large-scale, multi-physics systems in industrial real-world use cases. The language also supports the modeling of sampled data systems. Because of that, it provides a particularly good basis to model the dynamic behavior of complex systems composed of physical subsystems together with computing and networking (e.g., by modeling a complete *virtual vehicle* including the logical controller functions). These kind of systems are in the following denoted as cyber-physical

systems [1] (CPS). Given that qualities, Modelica seems to be particularly well-suited to enable the global approach to model-based development stipulated at the beginning of this section.

However, up to now, the use of Modelica for embedded systems development has actually been very limited. This can partly be attributed to a somewhat too limited expressiveness in modeling discrete-time controller functions, and partly to the lack of a seamless development approach from the controller model comprising the *logical functions* to the *technical system architecture* (i.e., code running on the target platform).

The goal of this thesis is to enable a Modelica based development process that removes these limitations. Figure 1.1 contrasts a model-based development process using the current state-of-the-art technology with an improved process that takes advantage of results from the research efforts of this thesis. Note the gap in the transition from Modelica control function specification models to models used during the design and implementation phase. Modelica is currently not well-suited to be used in that phase. This makes it necessary to manually rebuild the models within a more dedicated embedded software modeling tool (with suitable automatic code generator) or to convert the models manually to C code. In either way, the *manual conversion is time-consuming and error-prone*, especially if several design iterations are required.

Closing this gap in order to enable a more integrated approach to the development of cyber-physical systems is the main motivation for this thesis.

1.2. Main Contributions

The main result of this thesis is a Modelica based framework for model-based control function development, which integrates state-of-the-art modeling for digital control functions with state-of-the-art modeling for physical systems.

The essential approach to achieve this is by transferring concepts from synchronous languages for safety-related real-time applications into the Modelica language. This enables to close the gap in the transition from Modelica control function specification models to models used during the design and implementation phase (see Figure 1.1). My main contributions in this context are:

1. Identification of the current shortcomings of Modelica with respect to established requirements in safety-related, model-based function development.

2. A proposal for language restrictions and extensions dedicated to modeling discrete-time controller functions, that is deemed suitable as basis for a *high-integrity* automatic code generator (ACG), resulting in a language denoted as *ACG-Modelica*.

3. Formulation of a *translational semantics* that maps a representative subset of ACG-Modelica to a *synchronous data-flow kernel language*.

[1] The term "cyber-physical system" (CPS) is rather new and attributed to Helen Gil who coined the term around 2006 at the National Science Foundation in the United States [65, p. 5]. There is no complete consensus about the precise meaning of CPS within the academic world yet. My use of the term CPS coincides with the definition of Lee [63]: "A cyber-physical system (CPS) is an integration of computation with physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa.".

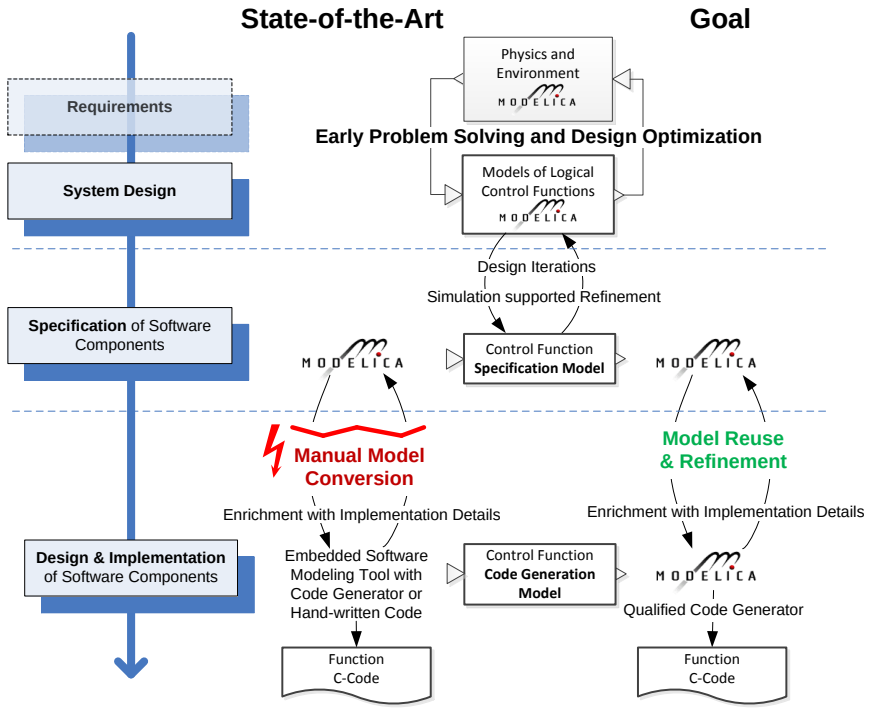


Figure 1.1.: Comparison of a state-of-the-art model-based development process using Modelica (**State-of-the-Art**) to the improved development process intended by this research effort (**Goal**). Note that the specification of software components can be divided into the specification of the *data model*, the *behavioral model* and the *real-time model* [95, p. 182]. Modelica is used for the *behavioral model* specification.

4. Prototypical realization of a translator implementing the formulated translation equations and an elaborate example that illustrates the interplay between physical modeling and controller modeling within the envisioned Modelica based function development framework.

The target language of the translational semantics is reminiscent to data-flow kernels used in formalizing aspects of the compilation of synchronous block diagrams as found in SCADE/Lustre[2]. The SCADE code generator KCG has already been successfully certified/qualified for safety-critical applications (see e.g., [50]). Hence, I provide a formal translational semantics to a data-flow language kernel that is understood enough to be accepted by certification authorities. Consequently, this yields a strong argument for the feasibility of developing a *certifiable/qualifiable* code generator for the considered Modelica subset. Furthermore, that translational semantics could be used as a base to create a gateway from Modelica to SCADE, similar to what has been reported for Simulink/Stateflow in [31].

[2] `http://www.esterel-technologies.com/` (May, 2014).

1.3. Thesis Outline

The thesis addresses two aspects of function development with Modelica: first the conditions that need to be fulfilled to allow the usage of Modelica for implementing *function models* in a safety-related development process with automatic code generation, and second the use of Modelica as a basis for a seamless development methodology starting from the *logical virtual model* down to the *technical system architecture*.

Chapter 2: Background The chapter describes the state-of-the-art and defines relevant terms.

Chapter 3: Requirements for Model-Based Function Development This chapter identifies (to some degree arguably subjective) requirements to a high-level, domain-oriented language that need to be fulfilled in order that the language is suitable to be used in a safety-related, model-based function development process. This includes in particular:

- The identification of several key stake-holders within the development process including their specific requirements that impact the requirements imposed on a suitable domain-oriented language.
- The identification of general requirements to a domain-oriented language for model-based (control) function development
- Additional considerations regarding the effectiveness and efficiency of model reviews for languages that are (formally) specified at the textual level but also have a graphical representation (as is the case for Modelica).

The so established requirements guide the language design decisions in the following chapters.

Chapter 4: Modelica's Synchronous Language Elements Extension The synchronous language elements extension was introduced in Modelica 3.3, May 2012 in order to improve the language support for modeling sampled data systems. This chapter describes this extension, compares it with the previous approach of modeling sampled-data systems in Modelica and concludes with a brief note about the historical development that led to that recent extension including contributions that can be attributed to the author of this thesis.

Chapter 5: Enabling Modelica for Function Development The Modelica language is analyzed with respect to the identified requirements for a domain-oriented language for model-based control function development. Based on this analysis current shortcomings are identified and practical proposals to mitigate the shortcomings are worked out. In particular:

- Established technologies and procedures for safety-related function development with automatic code-generation are briefly presented and contrasted to current Modelica technology. Based on this, I propose an adaption of current Modelica technology.

- I propose specific *language restrictions* to ease the challenge of tool qualification.
- Also, I work out proposals for *language extensions* in order to increase the suitability of Modelica to model discrete-time controller functions. These include proposals for:
  - Supporting most relevant target data types and operations.
  - Modularization of dynamic execution aspects.
  - Manual block scheduling.
  - Causal data-flow semantics

The result of this effort is a restricted and extended Modelica language set for safety-related digital control applications that I denote as *ACG-Modelica*.

Finally, I present a *translational semantics* from a representative subset of the ACG-Modelica language to a synchronous data-flow kernel language. This kernel language allows to resort to established automatic code generation techniques for data-flow languages that are, in particular, understood enough to be accepted by certification authorities.

Chapter 6: Servo System Example The chapter presents a typical control engineering task: the design of a digital controller for an electric drive system (servo control system). The example illustrates the usage of Modelica and ACG-Modelica in a framework for model-based function development.

Chapter 7: Discussion and Conclusions A final discussion of the essential contents of the individual chapters and a brief conclusion regarding the main results of this thesis.

1.4. Related Publications

The research results presented in this thesis are partly based on the following peer reviewed conference and journal papers:

- 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. http://dx.doi.org/10.4173/mic.2015.1.3
- 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
- Martin Otter, Bernhard Thiele, and Hilding Elmqvist. A Library for Synchronous Control Systems in Modelica. In 9th *Int. Modelica Conference*, Munich, Germany, September 2012. http://dx.doi.org/10.3384/ecp1207627
- Bernhard Thiele and Dan Henriksson. Using the Functional Mockup Interface as an Intermediate Format in AUTOSAR Software Component Development. In 8th *Int. Modelica Conference*, Dresden, Germany, March 2011. http://dx.doi.org/10.3384/ecp11063484
- Hilding Elmqvist, Martin Otter, Dan Henriksson, Bernhard Thiele, and Sven Erik Mattsson. Modelica for Embedded Systems. In 7th *Int. Modelica Conference*, pages 354–363, Como, Italy, September 2009. http://dx.doi.org/10.3384/ecp09430096

2. Background

2.1. The Complexity Challenge at the Example of the Automotive Sector

The rapid advances in electronic control technology lead to more complex software functions that pose a challenge to the development process. This development is not restricted to specific industrial sectors. However, some sectors observed a significant higher growth rate of software complexity within the recent years than others. Among them is the automotive industry where the complexity[1] of embedded software code increased about fiftyfold within the last 15 years [37].

Modern automotive vehicles can have over 60 ECUs (Electronic Control Units) which are interconnected by a heterogeneous network of automotive buses, like CAN, FlexRay, LIN, and MOST for communication purposes [37]. Handling such complex systems is a big challenge which is intensified by tightly scheduled development phases due to product market competition. Currently the development phase for vehicles is estimated to be about three years [95]. The increased use of virtual engineering methods has the potential to further reduce the required development time span.

An additional important requirement for embedded automotive systems is safety. Today, ECUs increasingly provide safety-related functionality. Provided functions range from situation analysis (e.g., speedometer display) and giving situation assessment (e.g., black-ice warning) to active intervention in braking or steering actions (e.g., in a vehicle equipped with active front steering (AFS) [61]). Future vehicle designs might even go a step further and dispense mechanical back-up linkages to the braking and steering system (Break-by-Wire, Steer-by-Wire) [38, 28, 9]. At the same time, safety requirements requested by authorities are raising, which is reflected in safety regulations like IEC 61508 (*"Functional safety of electrical/electronic/programmable electronic safety-related Systems"*) and its application specific derivations, most notably the recent ISO 26262 (*"Road vehicles – Functional safety"*) for automotive applications. As a consequence, virtual prototyping methods need to be amenable to support activities that improve confidence in the functional safety of the system.

In order to develop and validate functions for cyber-physical systems, it is no longer sufficient to solely consider single, self-contained components without taking the interaction of the function within the whole system into account. Technology that is capable of seamless integration of physical system models and safety-related software functions within a virtual prototyping environment is mandatory to cope with the inherent complexity of CPS, enable short development cycles, and at the same time, ensure safe products.

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

2.2. Modeling in (Control) Engineering and Computer Science Communities

When working in the interdisciplinary field of cyber-physical systems care has to be taken about the meaning of similar terms in different communities. As Henzinger and Sifakis note about the "education challenge" in the discipline of embedded systems design [55]:

> "The lack of a common cultural background also results in fragmented research. Different communities use different terminologies."

An important example is the term *"model"* which is omnipresent throughout this text.

Control engineers with mechanical or electrical engineering background typically expect models to abstract from reality by providing a mathematical description (equations!) of the behavior of the modeled object[2]. Computer programs that provide support for building such *analytical* models are also expected to be able to execute/simulate them. Typical modeling abstraction in control engineering are block diagrams (transfer functions composed by port connections with a data flow semantics) or declarative (acausal) equations[3].

Software engineers, tend to have a more abstract view of the term *"model"*. A model may be any form of (formal) notation that is capable of describing certain aspects of a matter of interest. They are also accustomed to the concept of *meta-models* (a term absent in more traditional engineering education), denoting a (formal) notation that is capable of describing another *model*. Consequently, software engineers also do not tend to implicitly expect that a model in a machine consumable format will be executable. A well known example for modeling within the discipline of software engineering is the UML language with its four-layers meta-modeling architecture [81].

This text embraces the more abstract view of the term *"model"*. Nevertheless, the considered models are mainly of analytic nature and therefore match well with what might be expected by readers with a control engineering background.

2.3. Model-Based Development

Model-Based Development (MBD) makes systematic use of (formal) models as primary artifacts throughout the overall development process. In classical software development (e.g., IT and business applications), this traditionally refers to the application of advanced software modeling technology to capture requirements, and software structure [18]. However, in the embedded (control) systems community the term is usually more narrowly understood as (analytical) executable system models (consisting of control algorithm models as well as the corresponding physical plant models) that are the primary artifacts for the control function design process. Starting with capturing the requirements, the system model is continuously elaborated and constitutes the heart of design, implementation and testing activities. This view, the model as center of all development and design activities, is also often referred to as *model-based design* [97].

[2] Of course there exist model abstractions used in engineering where equations are not in the foreground. Notably geometry models, as used in engineering drawings, as well as wiring or hydraulic diagrams etc. However, a control engineer usually doesn't associate that kind of models when using the term *modeling*.

[3] Acausal means that the causality of how to solve the equations is not decided at modeling time

Nowadays, model-based development has become a well established development approach in the domain of embedded (control) systems and the original promise of model-based development, to provide a more rapid and economic development process, seems to be confirmed in industrial practice [24].

In automotive software engineering the term *Model-based Function Development* is commonly used to indicate that the traditional function development process is replaced by a model-based approach – using analytical (graphical) models for developing the behavioral part of the embedded software.

Using this more precise term allows to distinguish between *Model-based Function Development* and *Model-based System Development*, where the later term encompasses the use of models to describe system and software architecture models [78].

Therefore, the use of formal models within the development process is multifaceted, however, this work is particularly concerned with *Model-based Function Development* using Modelica as modeling language of choice. The aim is to improve the suitability of Modelica from a modeling language for the physical environment, to a modeling language that is equally suited as base for the so-called *high-level application model* (or *specification model*) and *code-generation model*[4] as indicated in the V-model of Figure 2.1. The V-model is a conceptual model for systems

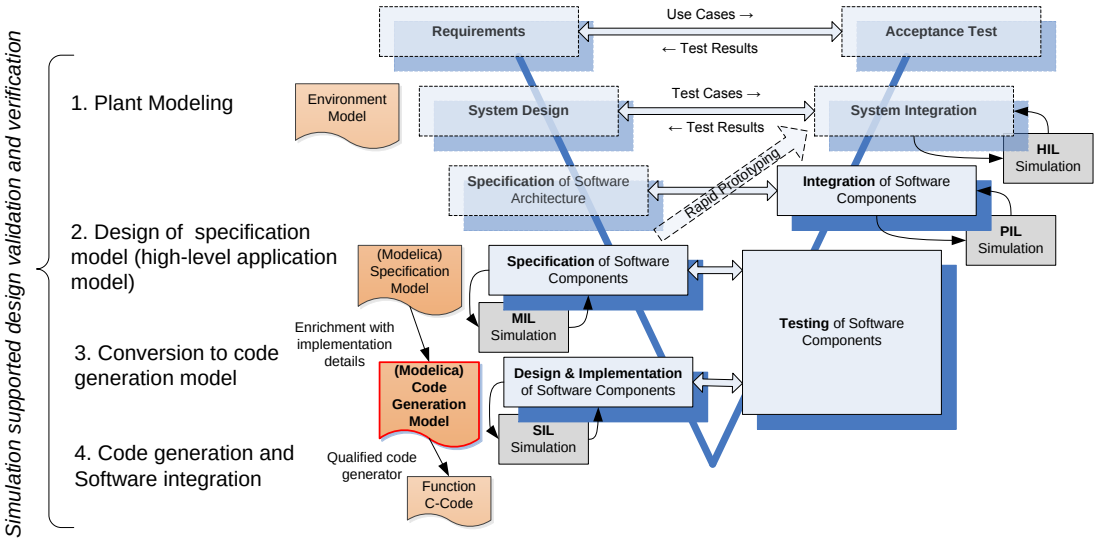


Figure 2.1.: Model-based function development embedded in the V-model for system development.

development that is routinely used in various industries, including the automotive industry. Note that there are many variations of the V-model in use, but this is beyond the scope of this work.

[4]There is no common agreement about the naming of the different categories of models that arise during the MBD process. Some authors use the term *physical model* instead of *specification model* and *implementation model* instead of *code-generation model*. For this document I adopt the terms *specification model* and *code-generation model*, however the reader should be aware that there are different terms in use.

In order to optimize the benefits gained by a model-based development process as illustrated in Figure 2.1, it is crucial that formally specified high-level applications can be automatically transformed (usually by using generated embedded C-code) into executable binary code for respective embedded platforms, thus eliminating error prone and expensive manual recoding of the application into a general-purpose programming language. Note that for accentuating that formal models are transformed into executable code (as opposed to "just" being used for specification/illustration purposes) the term *Model-Driven Software Development*[5] (MDSD) is also commonly used in the literature [98].

Figure 2.2 shows a typical model-based development toolchain including an automatic code generator. The specification model is designed using a high-level domain-oriented modeling tool. These specification models are typically enriched with implementation details resulting in so-called *code generation* or *implementation* models. A code generator (also known as *autocode tool*) transforms code generation models into C-code, that the cross compiler translates into object code. The different object codes, including legacy and basic software code, are then finally linked to a binary that can be executed by an embedded target.

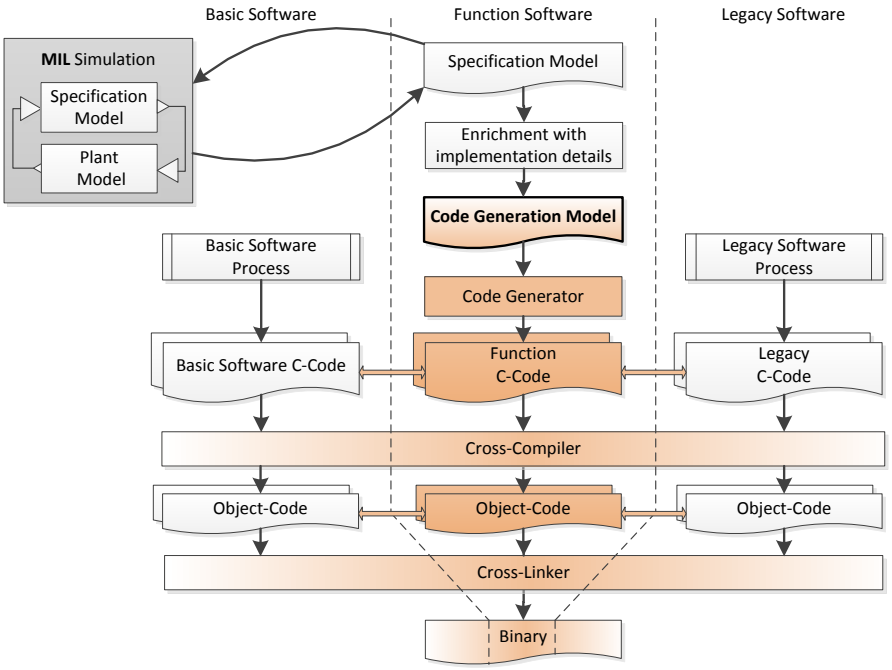


Figure 2.2.: The generic build process for a model-based development toolchain with an automatic code generator (adapted from [94]).

In the domain of control engineering *simulation* plays a prominent role within model-based design. That prominent role is indicated in Figure 2.1 and 2.2 by the boxes entitled MIL, SIL,

[5] Equally the terms *Model-Driven Development* (MDD), *Model-Driven Engineering* (MDE) or *Model-Driven Software Engineering* (MDSE) are in use.

PIL, and HIL simulation (meaning Model-in-the-Loop, Software-in-the-Loop, Processor-in-the-Loop and Hardware-in-the-Loop simulation).

Especially as systems become more complex (as typical for cyber-physical systems) MIL simulation becomes an indispensable tool for early design validation and verification. In particular, given an executable model of physical aspects (plant model) and software aspects (high-level application model) of the complete system, it becomes feasible to

- obtain very fast feedback about the effect of system design decision in an early development phase;
- evaluate and incrementally improve designs, even without available prototypical hardware;
- apply (multi-criteria) computational optimization methods in order to find a better balance between possible design variations;
- test and evaluate system behavior under extreme conditions that otherwise would be too costly, dangerous or (physically) unfeasible with real prototypes;
- synthesize and analyze high-performance (model-based) controllers.

2.4. Modeling Tools and Languages for Embedded Systems Design

The most prevalent domain-specific software for model-based design of embedded software (including code generation) on the market is Matlab/Simulink [10, 103] from MathWorks. Matlab is a numerical computing environment, suitable for algorithm prototyping, data analysis, and advanced data visualization capabilities. Simulink is primarily a graphical block diagram language that is tightly coupled with Matlab. Block diagrams are a popular domain specific language (DSL) used within control systems and digital signal processing (DSP) development. It is possible to generate C code from Simulink models for various (embedded) target platforms using code generators offered by MathWorks or third party companies (most notably the dSPACE TargetLink tool [52]).

However, the modeling capabilities of Simulink for physical systems are rather primitive. In Simulink physical systems need to be modeled using the same graphical block diagrams that are common in control engineering and DSP design. Yet, block diagrams are not up to the state of the art in physical modeling. MathWorks finally reacted to this deficiency by offering a Simulink toolbox named SimScape to enable object-oriented physical modeling (SimScape made its debut in the R2007A release of Matlab/Simulink). Unfortunately, instead of using the open Modelica language standard, MathWorks decided to create its own, proprietary, physical modeling language. In effect, the huge pool of high-quality (free and commercial) physical modeling libraries already available for Modelica cannot be reused in SimScape.

In contrast to Simulink that has its roots in control systems and DSP engineering and was later extended to offer decent support for physical modeling, the development of the Modelica language started from the beginning with the aim of creating a language that excels in multi-domain

physical modeling. Modelica was conceived in the late 1990s by the Modelica association[6]. Its initial design was heavily influenced by the pioneering work of Elmqvist [40]. From the beginning Modelica embraced a truly object-oriented physical modeling style and was actively maintained and developed as open standard by the non-profit Modelica Association. Today, several implementations (commercial and free) of Modelica are available[7]. The best known commercial implementations include Dymola[8] and SimulationX[9], a decent open-source implementation is available with OpenModelica[10].

Modelica proved to be a powerful tool to model large-scale, multi-physics systems in industrial real-world use cases. Thus, it is well suited to model the dynamic behavior of complex cyber-physical systems (CPS) (e.g., by modeling a complete *virtual vehicle* including the logical controller functions [11, 36, 32]), which is a key requirement for a model-based, global development approach.

However, up to now the use of Modelica for the design of control algorithms has been very limited. Available records of using Modelica for control algorithm development are restricted to high-level prototypical work for advanced (model-based) controller designs [68, 5, 111, 112, 102]. After prototyping the control algorithms in Modelica, the Simulink toolchain or manually written code is used for deploying the control logic to an embedded system target.

A notably exception is the direct use of executable code generated from Modelica within a model predictive control (MPC) framework for optimized start-up of power plants reported by Franke [46]. However, control on the process management level (with time horizons between minutes to hours) is very different to the embedded control systems that are considered in this thesis (with time horizons of the magnitude of (sub-)milliseconds).

Published reports of using Modelica as source format for direct generation of code for embedded system targets have been, so far, of purely academic nature [42, 2, 20]. This can be partly attributed to a somewhat **too limited expressiveness in modeling discrete-time controller functions**, and partly to the **lack of a flexible, seamless development methodology** from the controller model comprising the *logical functions* to the *technical system architecture* (i.e., code running on the target platform).

Although there exist a lot of concepts, technology and software targeting embedded systems (both industrial and academic), the available technology usually either focuses on modeling the embedded software, or focuses on modeling the environment.

It is important to mention that there exist quite a lot of industry relevant languages that target embedded systems development. However, they either focus on modeling the embedded software, *or* focus on modeling the physical environment.

- OMG SysML [80] is a language specifically designed to support system engineering. Although it provides a holistic system view, the abstraction level and expressiveness is not well suited for physical modeling and simulation[11].

[6] Modelica Association (2012), https://www.modelica.org/.

[7] A good overview is provided by the Modelica Association (2012), https://www.modelica.org/tools/.

[8] Dymola, Dassault Systèmes AB (2012), http://www.dymola.com/.

[9] SimulationX, ITI Gesellschaft für ingenieurtechnische Informationsverarbeitung mbH (2012), http://www.simulationx.com/.

[10] OpenModelica, Open Source Modelica Consortium (OSMC) (2012), http://www.openmodelica.org/.

[11] Note that ongoing research tries to integrate the descriptive power of SysML with the analytic and computational

- Synchronous languages [16] are an established technology for modeling, specifying, validating, and implementing real-time embedded applications. Prominent members of the synchronous family include the Lustre [30], Esterel [21] and Signal [62] language. The greatest industry relevancy can be attributed to the Lustre-based commercial SCADE tool [91] that is especially used for the development of safety critical software functions. Synchronous languages have no notion of continuous time and are therefore not suited for physical modeling.

- VHDL-AMS [7] is an extension to the VHDL electronic systems hardware description language. It is capable of multi-domain (physical) modeling. Its root in electronic design circuits is still noticeable, consequently its main focus is currently the verification of analog, mixed-signal integrated circuits. It is not intended for generating code.

Note the lack of decent combined support of both, *physical* **and** *functional modeling*, consequently these languages are not suitable for the envisioned integrated approach to CPS development.

There is also a considerable amount of academic research activities relevant within the research subject. Naturally that includes work conducted in the field of hybrid modeling and simulation as well as in the field of languages for (formal) control algorithm specification. Of particular interest in this work is the interplay between control system modeling and physical modeling.

- The Ptolemy project [39] studies different aspects of the design of embedded systems. A key concern is the interaction between well-defined *models of computation (MoC)*. It is possible to model hybrid systems by combining a continuous MoC with one or several of the provided discrete-time MoC [66]. The continuous modeling is similar to the block diagrams used in Simulink and thus, physical modeling capabilities are rather constrained.

- Scicos [75] is a graphical system modeler and simulator for hybrid dynamical systems [76]. Its model of execution is based on the synchronous language Signal extended to continuous-time systems. Interestingly, it allows to integrate a subset of the Modelica language to enrich its continuous-time dynamics modeling capabilities. However, the partial support of Modelica severely limits potential reuse of available Modelica libraries. Furthermore, a really seamless integration of Modelica into Scicos is missing (a preprocessing step is required in which C code conforming to an ordinary external Scicos block needs to be generated from a Modelica model which then needs to be compiled and linked against Scicos before simulation).

- A lot of research in hybrid systems targets the formal verification of system properties that depend on the interaction of discrete control algorithms and the physical environment (i.e., plant to be controlled) see e.g., [4, 3, 54]. It should be noted that applying these verification methods to large scale models is still an unsolved problem due to state explosion issues.

power of Modelica models [92].

- Within the synchronous language community recent studies attempt to establish alternative semantic foundations for hybrid systems modeling [12, 14]. Benveniste et al. [14] proposes to use *non standard analysis* [90] as mathematical base for establishing the semantics of a hybrid simulator. Bauer and Schneider [12] consider a semantically well defined hybrid extension of the synchronous language Quartz [93] to enable simulation and formal verification. The emphasis of this work is rather in mathematical rigor and fundamentals than in providing a practical methodology for the virtual prototyping of real-world systems.

- All of the aforementioned work primarily target functional aspects of embedded systems development. In practice, non-functional aspects like process management, communication or fault-tolerance mechanisms are an integral part of the complete system. Integration of that aspects within a model-based development process is considered in [26, 27, 25]. I share the spirit of this contributions in the sense that these aspects need to be considered within a virtual prototyping methodology for cyber-physical systems.

2.5. Safety Relevant Software Functions

Naturally, customers and users of a technical product (be it a household appliance or a civil aircraft) may expect that the product is *"safe"*. In order to ensure the absence of non reasonable risks related with the use of a product, legislative authorities have established rules and regulations that are mandatory for producers, distributors, and others who make the products available to the public. This regulations can include approval requirements by official authorities as a prerequisite for making the product available or product liability laws that held producers liable for harm caused because of defects within their products.

2.5.1. Terminology

Product liability (in regard to person injury or death) is the area of law in which producers are held liable for harm caused because of *defects* within their products.

A product has a *defect* if it does not provide the *safety* a person is entitled to expect.

A product is considered *safe* if the *risks* caused by faults/failures of the product and interactions with the product are within reasonable limits.

Risk is the combination of the occurrence probability of a failure and the severity of the harm that may be expected by the failure (e.g., injury or death of persons).

Functional safety is the absence of not reasonable risk caused by malfunctioning behavior of E/E safety-related systems and interaction of these systems[12].

Functional safety standards give guidelines about the limits of what the majority of experts consider a reasonable risk, as well as actions that can be taken to reduce risk to a level that is considered reasonable.

[12]Functional safety as defined by ISO 26262 does not address hazards related to electric shock, fire, smoke, heat, radiation, toxicity, flammability, reactivity, corrosion, release of energy, and similar hazards unless directly caused by malfunctioning behavior of E/E safety-related systems [58, ISO 26262-3:2011, p. 1].

Safety-relevant software functions are software functions that are part of an E/E safety-related system. They are required to fulfill certain *safety integrity levels* in order to reduce the risk stemming from the software and its interaction with the environment to a reasonable level.

2.5.2. Functional Safety Standards

The international standard IEC 61508 *Functional safety of electrical/electronic/programmable electronic safety-related systems* is intended to be a base norm for functional safety with general applicability in industry. However, the mapping of the generic base standard to specific application can be difficult. For that reason, application specific standards exist that interpret IEC 61508 in the context of a specific domain.

Routinely, standards are adapted to fit national requirements and native language. Figure 2.3 displays several derivations of IEC 61508 that particularly apply to Germany (though often similar/identical standards apply to other western countries). IEC 61508 itself is available as German standard DIN EN 61508.

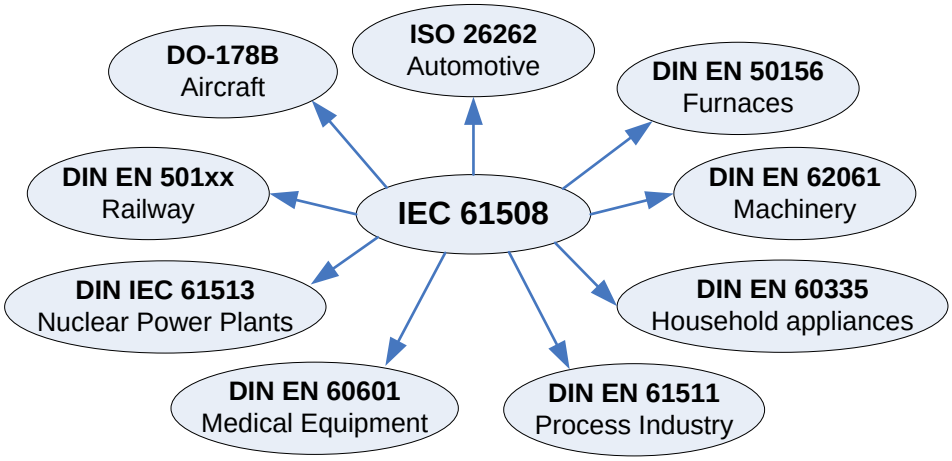


Figure 2.3.: Several standards derivating from IEC 61508. Figure adapted from [70, Figure 2-2].

2.5.3. Consequences

As a consequence of safety related requirements *model-based design approaches with automatic code generation targeting safety-relevant software functions need to ensure that required safety integrity levels are met*.

This has a crucial impact on methods and development tools that are applicable for the development task and it is essential to consider when deciding about the use of model-based development approaches and tools within a project.

Section 5.1 will return to that point and propose a qualifiable sub- and superset for model-based development using the Modelica language.

2.6. Software Architectures

Different definitions exist in literature for what is meant by the term "Software Architecture". In Clements et al. [33] following definition is provided:

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

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

A well known examples of a reference software architectures for embedded systems in the automotive domain is **AUTOSAR** (AUTomotive Open System ARchitecture[13]).

But how do software architectures affect the high-level application models considered in this text? Functions developed for complex systems (i.e., systems similar to the systems discussed in Section 2.1) will need to integrate into an (already existing) software architecture. That naturally has consequences for the tooling, e.g., the code generator has to produce code that conforms to the interface expected by the software architecture. In addition, it also has consequences for the environment model required during virtual prototyping of the function. "Non-functional" aspects like performance (time) models of (distributed) functions, scheduling policies, and bus communication characteristics have essential impact whether a design works as expected.

The term *Architecture-Driven Development* (ADD) has emerged to emphasise the central role of software architecture during such a development process. Niggemann et al. provide a notable discussion about the effects of ADD on established model-based function development practices and tools [77], stating

> "Most behaviour modeling and code generation tools, e.g., TargetLink, have so far mixed at least three different concerns: algorithmic models, function networks, and ECU configuration. Since such tools now focus on the "algorithmic model" concern in an architecture-driven development process, some modification become necessary.".

Since the use of architecture has become the state-of-the-art to enable improved reuse and integration of (embedded) software components I have in mind that my proposed framework for Modelica based function development integrates well in ADD processes.

[13] AUTOSAR development cooperation (2015), http://www.autosar.org/.

3. Requirements for Model-Based Function Development

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 2.2). 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., [100, 101, 104, 59]). 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 [100, 13]. 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 [94, 13].

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 3.2 defines general requirements for a modeling language deemed suitable for model-based (control) function development. In addition, Section 3.3 introduces requirements that are targeted specifically to the graphical representation of modeling languages whose semantics is originally defined at the textual level.

3.1. 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 [109, 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 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.

3.2. 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 3.3 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 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. [64, 39]). 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 [52, 56, 44]. 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 [89, 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 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*" [115]. For generated C-code in the automotive area that typically means that conformance to some coding standards, e.g., MISRA AC AGC [107] 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[1], and testing [115]. Finally, the generated code typically needs to be *integrated into a given software architecture* (see 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[2], 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.

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

[2] 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.

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 Section 5.6):

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.

3.3. 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 devel-*

opers, 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., [59]). Stürmer et al. [101] 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.

4. Modelica's Synchronous Language Elements Extension

The synchronous language elements extension was introduced in Modelica 3.3, May 2012 [73] in order to improve the language support for modeling sampled data systems. This chapter describes this extension and compares it with the previous approach of sampled-data modeling. It concludes with a brief note about the historical development that led to that recent extension, accounting for the contributions that can be attributed to the author of this thesis.

The chapter assumes that the reader already has basic knowledge about the Modelica language and *equation-based object-oriented* (EOO) modeling. In case that this prerequisite is not fulfilled, please refer to the Modelica introduction available in Appendix A.

4.1. Activation of Discrete-Time Equations

The following discussion about activation of discrete-time equations was presented by me in [109, Section 4.2.3, *Activation of Discrete-time Equations in Modelica*]. It is reproduced (almost) literally in this section since it provides valuable context information to the development of the synchronous language elements extension.

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

Time events are scheduled by the solver along a global simulation time line. Time is a (physical) real number (as opposed to the principle of *multi-form time*[1] adapted by synchronous languages) that steadily increases during execution (simulation) of a Modelica model. The global simulation time can be accessed anywhere in a Modelica model by the built-in variable `time`[2].

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

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

In order to allow smooth integration of code generated from Modelica into embedded systems software projects, Modelica needs to allow *external* code to simply cause the evaluation of a set of (discrete-time) Modelica equations (without the internal participation of a hybrid systems

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

[2] Note that in the proposed language restrictions in Section 5.4.1, Rule 10, the use of the built-in variable `time` is forbidden.

solver that tries to detect whether the equations shall be evaluated or not). Nikoukhah and Furic [79] provide a notable discussion about the missing feature of external activation in context of using Modelica models within the Scicos[3] modeling environment which similarly applies to using Modelica models in embedded systems software projects.

To allow external activation of Modelica models Nikoukhah and Furic propose in [79] to add an `Event` type to the Modelica language and discuss the elements and semantics needed to integrate that new type in a general and backwards compatible way.

The latest Modelica 3.3 language standard added *synchronous language elements* particularly targeted at the implementation of control systems (see Section 4.3). They add *clock activation* as a third way of activating discrete-time equations that largely solves the hitherto criticised deficiencies.

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

Chapter 5 builds on the synchronous extension described here and extends it further to improve the suitability of Modelica as high-level modeling language for model-based function development that allows to generate code that integrates smoothly into embedded systems software projects.

4.2. Pre-Modelica 3.3 Support for Sampled-Data Systems

Previous (i.e., pre-Modelica 3.3) support for sampled data systems is built upon detection of an edge in Boolean conditions occurring in when-clauses. The equations within the when-clause are active instantaneously at the detected event points, otherwise they are inactive. A variable that is assigned by an instantaneous equation can only change its value at an event and keeps its value constant during events (implicit zero-order hold semantics). A special **`sample`** operator is provided to trigger time events.

Consider the simple controlled drive from Figure A.10. Setting aside the component hierarchy depicted in the figure, a *flat* Modelica model utilizing a sampled-data control version of the PI controller can be accomplished by using the sample operator and a when-clause which encapsulates the sampled variables[4] (see Listing 4.1).

Listing 4.1: Simple controlled drive with sampled PI control.

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

[4] Note that the actuator dynamics for translating the actuating variable into physical torque is omitted. In a more realistic model the actuator dynamics would need to be included. However, omitting it allows to keep the semantic discussion at this point succinct and readable.

```
1  model SCDSampled "Simple controlled drive with sampled PI control"
2    Boolean event = sample(0,0.1);
3    Real xd(start=0, fixed=true), y,ud;
4    Real w(start=0, fixed=true), ref(start=0, fixed=true);
5  equation
6    der(ref) = 1;
7    der(w) = 1/10*ud;
8    y = ref - w;
9    // sampled PI control:
10   when event then
11     xd = pre(xd) + y;
12     ud = 110*(xd + y);
13   end when;
14 end SCDSampled;
```

The operator `pre(y)` returns the "left limit" of variable $y(t)$ at a time instant t [73, p. 29]. Therefore `pre(xd)` in line 11 will return the value of `xd` from the previous event instant (and its start value at the first event instant).

The `y` variable (continuous-time input) is automatically sampled at $t = 0.0, 0.1, 0.2, \ldots$. Variables `xd` and `ud` are piecewise-constant and change their value only at event instants. Note that they are not time-discrete values, since their values can be accessed in-between events (see line 7). This is different to sampled-data systems theory, where conversion operators are required for signals that cross the boundary between continuous-time systems and (discrete-time) sampled-data systems.

4.3. Sampled-Data Systems with Synchronous Language Elements

The synchronous language elements extension introduced in Modelica 3.3 [73, Chapter 16] was introduced in order to improve modeling of sampled-data systems (see also [43, 84]). The design was influenced by work done by the synchronous language community [16], especially the work done on clock calculus and inference by Colaço and Pouzet [35] and the language Lucid Synchrone [87]. Note that despite the introduction of the new language elements, Modelica 3.3 provides backward compatible support for the "old" pre-Modelica 3.3 modeling style.

The semantic impact of the extension shall be demonstrated at the hand of the previous example in Section 4.2. Listing 4.2 shows a variant of that example using the new synchronous language elements.

Listing 4.2: Simple controlled drive with *clocked* PI control.

```
1  model SCDClocked "Simple controlled drive with clocked PI control"
2    Clock clk = Clock(0.1);
3    Real xd(start=0), yd, ud;
4    Real y, u, w(start=0, fixed=true), ref(start=0, fixed=true);
5  equation
6    der(ref) = 1;
7    u = hold(ud);
```

```
8    der(w) = 1/10*u;
9    y = ref - w;
10   yd = sample(y);
11   // clocked PI control:
12   when clk then
13     xd = previous(xd) + yd;
14     ud = 110*(xd + yd);
15   end when;
16 end SCDClocked;
```

The event generation in line 2 of Listing 4.1 is replaced by the construction of a *clock variable* using the **Clock**(**interval**) constructor with **interval** set to 0.1 (Listing 4.2, line 2). This creates a periodic clock which *ticks* (is active) every 0.1 seconds.

The *clocked* when-clause in line 12 denotes that all equations within the when-clause are clocked with the clock given by `clk`. The variables inside the clocked equations are only active if their associated clock ticks. In particular, they cannot be accessed in-between clock ticks. A *clock inference* mechanism is used to uniquely associate a variable to exactly one clock, i.e., if the clock is not explicitly defined, the clock of a variable is deduced forwards and backwards from the data flow. Variables that are associated with the same clock are said to be in the same *clock partition*.

As an example for the clock inference mechanism consider equations

$$e*x = a*\textbf{previous}(x) + b*yd; \qquad\qquad w = c*\textbf{previous}(x);$$

and assume that *yd* is known to be associated to a clock *clk*. In that case it can be inferred from the first equation that *x* must be on the same clock as *yd* and from the second equation that *w* must be on the same clock as *x*. Hence, it can be inferred that *yd*, *x*, and *w* must all be on the same clock *clk*.

Clock conversion operators are provided to access variables over clock partition boundaries. In the example listing the operator **hold** (line 7) provides the transition between the discrete-time partition and the continuous-time partition (acting as a zero-order hold element). Conversely, the **sample** operator is used for the transition from the continuous-time partition to the discrete-time partition (line 10). Note that the operator **sample** is overloaded and has *completely different semantics* in Listing 4.1, line 2! It is worth mentioning that **sample** uses the left limit of its input argument `y` when clock `clk` ticks (as a consequence, it introduces an infinitesimal delay which can act as a causality loop breaker as illustrated in Figure 4.1). Also note that the operator **pre** has been replaced by the operator **previous**(`xd`) (line 13) that returns the value of `xd` at the previous clock tick (and the start value of `xd` at the first clock tick).

Figure 4.2 outlines the concept of sampled-data systems using the synchronous language elements in a more general setting. The diagram shows that there exist additional overloaded versions of the operators **Clock** and **sample**. The **Clock**() in the when-clause means that the clock of the equations inside the when-clause needs to be inferred and that all equations need to be associated to the same clock. The **sample**(`y,` **Clock**(3)) samples the continuous variable `y` using a periodic clock with an interval of 3 seconds. As a result `yd` is a clocked variable and since it is accessed in the when-clause all equations inside the when-clause can be unambiguously inferred to be *on the clock* associated to `yd`. The figure also depicts the relation of the

```
model InfinitesimalDelayOfSample
  Real xd(start=0), y(start=0);
equation
  y = hold(xd);
  when Clock(0.2) then
    xd = sample(y) + 0.2;
  end when;
end InfinitesimalDelayOfSample;
```

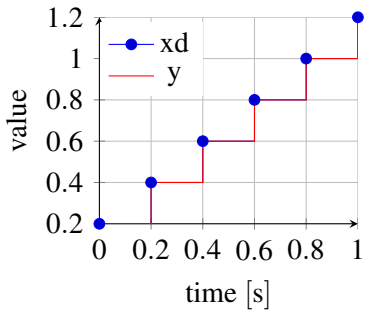


Figure 4.1.: The operator **sample**(y) uses the left limit of y whenever the clock ticks. This introduces an infinitesimal delay between the continuous-time partition and the clocked partition. As a consequence it breaks the apparent causality loop between x and y.

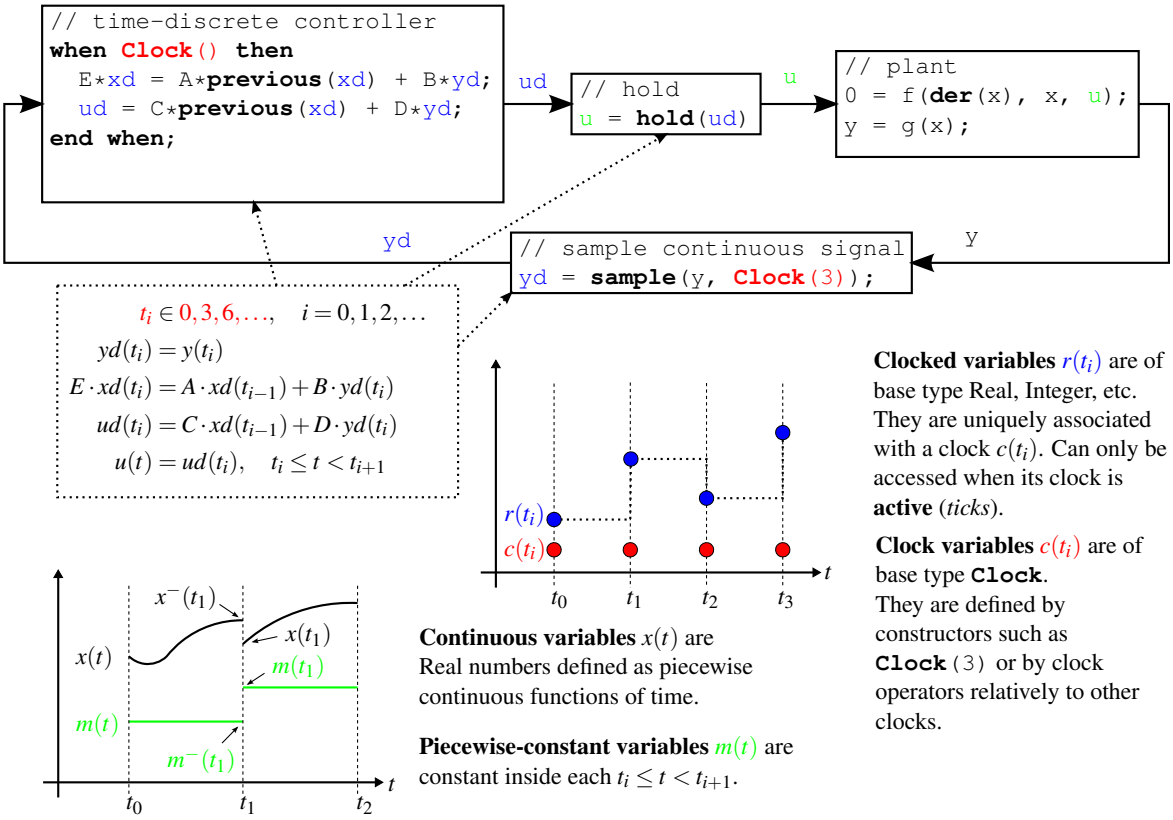


Figure 4.2.: Support of sampled-data systems in Modelica using the synchronous language elements (adapted from the diagrams in [73, p. 181, 184]).

different kinds of (discrete)-time variability supported in Modelica.

In addition to transitions between discrete-time and continuous-time partitions there is additional support for rate-transitions within discrete-time partitions. This is realized through the operators **subSample**[5] (fast-to-slow) and **superSample**[6] (slow-to-fast). As illustrative example consider the multirate cascade control structure in Figure 4.3.

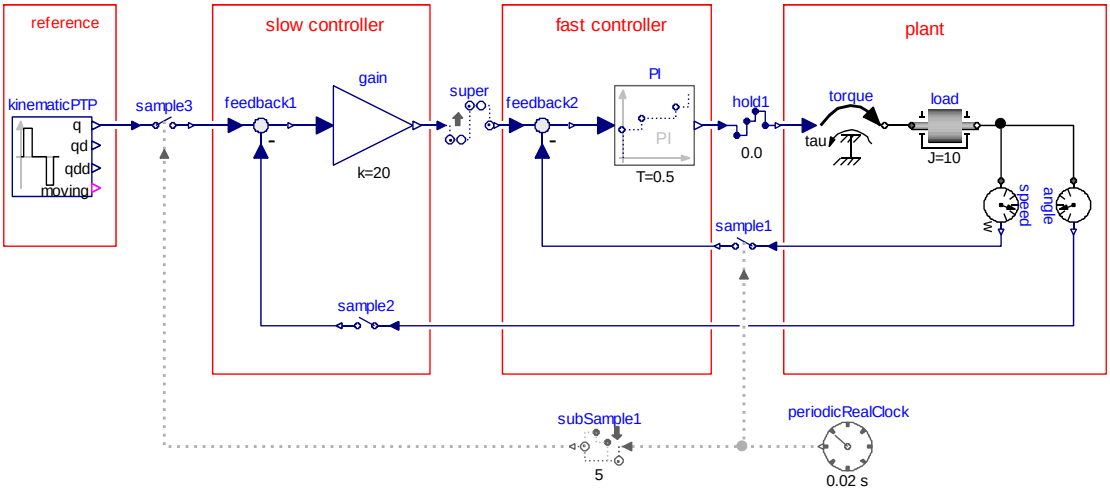


Figure 4.3.: Simple controlled drive with multirate cascaded controller.

The example is based on Figure A.10, but replaces the continuous-time PI-controller by a digital realization and adds an outer position control loop. Instead of using the introduced synchronous language elements directly, the example is built with the help of the *Modelica_Synchronous*[7] library. The library encapsulates the synchronous language elements in order to allow a convenient, graphical model definition of sampled-data systems [84]. Since the outer loop has a slower system dynamics than the inner loop, it is sufficient to use a lower sampling rate than for the faster system and still preserve a satisfying control performance. The example uses a fifth-time slower sampling rate for the outer position control loop than for the inner speed control loop. The blocks named `subSample1` and `super` encapsulate the **subSample** and **superSample** operators, the blocks starting with `sample` and `hold` encapsulate the **sample** and **hold** operators. The *base-clock* for the example is instantiated in the block named `periodicRealClock` and set to a sample period of 0.02 seconds.

Multirate sampling is relevant for practical controller implementations on hardware with finite computational capabilities. A lower sampling rate reduces the computational load imposed

[5] "The clock of `y=`**subSample**`(u, factor)` is factor-times slower than the clock of `u`. The first activation of the clock of `y` coincides with the first activation of the clock of `u`. If argument `factor` is not provided or is equal to zero, it is inferred", [73, p. 189].

[6] "The clock of `y=`**superSample**`(u, factor)` is factor-times faster than the clock of `u`. At every tick of the clock of `y`, the operator returns the value of `u` from the last tick of the clock of `u`. The first activation of the clock of `y` coincides with the first activation of the clock of u. If argument `factor` is not provided or is equal to zero, it is inferred", [73, p. 189].

[7] `https://github.com/modelica/Modelica_Synchronous` (accessed June, 2013).

on an electronic control unit and therefore allows a more economic realization of available resources. Further reasons for multirate sampling is the need to accommodate multirate sensor measurements and to handle distributed, networked systems [48, 17]. Support for multirate systems is therefore an indispensable element for a modeling language supposed to support CPS development as required in the motivation for this work (see Section 1.1).

The activation of the clocked, disrete-time control logic of the multirate system in Figure 4.3 is synchronized to a common base-clock (block `periodicRealClock`). The equations and variables synchronized to that base clock form a *base-clock partition*. Several base-clock partitions may coexist in an overall model. However, no precise[8] synchronization is assured between different *base-clock partition*. Communication between different *base-clock partition* needs to either route signals over the continuous-time partition (utilizing **`sample`** and **`hold`**), or use the **`noClock`** operator. At a tick of the clock associated with `y` in "`y = noClock(u)`" the operator will return the value of `u` from the last tick of the clock of u (or the start value of `u`, if the clock of `y` ticks before the clock of `y`).

Note that there is a subtle difference between the statements "`y = noClock(u)`" and "`y = sample(hold(u))`" due to the infinitesimal delay introduced by **`sample`** (remember Figure 4.1). This is illustrated in Figure 4.4. The example has two base-clock partitions driven

```
model NoClockVsSampleHold
  Clock c1 = Clock(0.1);
  Clock c2 = Clock(0.2);
  Real x(start=0), y(start=0), z(start=0);
equation
  when c1 then
    x = previous(x) + 0.1;
  end when;
  when c2 then
    y = noClock(x);
    z = sample(hold(x));
  end when;
end NoClockVsSampleHold;
```

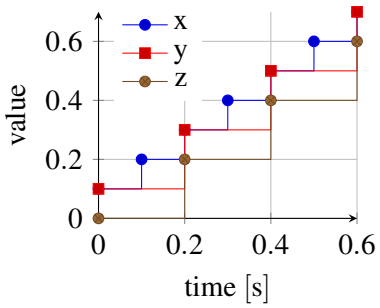


Figure 4.4.: Example comparing "**`noClock`**`(x)`" with "**`sample`**`(`**`hold`**`(x))`".

by clock `c1` and `c2`, respectively. The base-clock partitions need not to be sorted to each other, so at the time events $t = k \cdot 0.2$, $k \in \mathbb{N}$ either of the both partitions may be evaluated first. Since **`sample`**`(u)` returns the left limit of `u` (and the left limit of a variable doesn't change during event iteration) variable `z` will always hold the "delayed" value of `x` (regardless of the evaluation order). In contrast, the value returned by **`noClock`** will be dependent on the evaluation order, since it will just take the most recent value. In the example the base-clock partition corresponding to clock `c1` is evaluated before `c2`, so that the value of `y` is identical to `x` at $t = k \cdot 0.2$, $k \in \mathbb{N}$. Note, that it would be equally valid to evaluate the partition corresponding to `c2` first, which would result in `y` being equal to `z`!

[8] "Precise" in the sense that the synchronous model of computation described in Section 5.3.3 holds.

Besides the described operators introduced so far, additional operators exist. They are very briefly presented in Table 4.1. Details need to be looked up in the specification. They are omitted here for brevity reasons.

Table 4.1.: Other operators

Operator Syntax	Description
interval(u)	Returns the time interval (duration) between the previous and present clock tick.
shiftSample(u, shiftCounter, resolution)	Returns the the value of u, but shifts the activation of the associated clock in time ("delay").
backSample(u, backCounter, resolution)	Analog to **shiftSample**, but shifts the clock "backwards" in time. Returns the value of u from the last tick of the clock of u, or the start value of u before the first tick of the clock of u.

Compared to the pre-Modelica 3.3 support for sampled-data systems the new synchronous language elements extension provides several advantages. The most prominent ones are:

- *Improved modeling safety*. The compiler can *detect modeling errors* that arise due to unintended connecting blocks with different sampling period. This was not possible in pre-Modelica 3.3 due to the automatic sample and hold semantics of variables occurring in "old" when-clauses.
- *More general equations*. The "old" when-clause allows only a restricted form of equations since the left-hand side (LHS) must be a variable. Clocked equations do not have that restrictions (see Figure 4.2, upper left block).
- *Guaranteed synchrony*. Equations in different ("old") when-clauses are only guaranteed to be synchronous if triggered by the *same* event (e.g., events generated by "`when sample(0,3) then...`" are not guaranteed to be synchronized with events generated by a second statement "`when sample(0,3) then...`" since the statements contain two *different* event source instances "`sample(0,3)`"). This is error prone and potentially surprising. In contrast, clocking information of variables used in clocked equations is propagated by *clock inference* and synchronous activation of respective equations is ensured.
- *Improved inverse-model usage*. Clocked equations provide improved support for inverse-model usage in advanced controllers. This rather special topic is demonstrated in [84].
- *More efficient simulation*. Due to cleaner decoupling of continuous and discrete parts hybrid models with clocked equations allow a more efficient simulation scheme at event points, see [43, p. 24]
- *Improved memory efficiency*. All variables assigned in an "old" when-clause may be accessed outside the when-clause before they have been assigned a value for the first time. In this case the variable needs a start value and needs to be treated similar to a "discrete-time

state" variable. It is difficult for a compiler to deduce whether this situation applies, so that in practice usually all such variables are treated similar to "discrete-time state" variables, although in reality only a subset of them actually needs an initial state [43, p. 24]. This isn't the case for clocked variables, so that clocked variables can be handled more memory efficient.

- *Simplified initialization*. The initialization of a (hybrid) Modelica model is usually a global system-level issue involving many intricacies which require considerable expert knowledge to fully understand[9]. For improved comprehensibility, the initialization rules for clocked partitions follows a simplified scheme that is independent from global system level aspects (see [73, p. 197]).

4.4. Notes to the Development of the Synchronous Extension

The development of the synchronous language elements extension to Modelica started at the end of 2010 and was initiated and decisively driven by Hilding Elmqvist (see fundamental paper [43]), though a larger group of people contributed to the discussion, development and testing.

During the discussion phase the author of this thesis emphasized that the synchronous extension should not only consider periodic activation of sampled data systems, but also provide means to allow aperiodic activation of sampled data systems during hybrid simulation (i.e., combined simulation of synchronous (discrete-time) and physical (continuous-time) model). A well-known practical example for aperiodic activation of (control) data-systems is a modern combustion engine management system in which fuel injection and spark ignition is synchronized with the engine crankshaft position. This significantly contributed to the introduction of the *boolean clock constructor* which allows to drive clock activation by events generated by a continuous-time expression of type Boolean. Its definition is

```
clock(condition, startInterval)
```

where `condition` is the event condition and `startInterval` is the value returned by the operator `interval()` at the first clock tick. A simple usage example is given below.

```
Clock clk = Clock(angle > 0.2, 0.1);
```

Also, using the synchronous language elements for modeling at the textual level seems to have a rather steep learning curve and easily leads to models that can hardly be understood. This triggered the development of the MODELICA_SYNCHRONOUS library that had the aim to provide a more usable interface to the new synchronous language extension [84]. For an example see Figure 4.3. This library in which I did essential conceptual and development work is now a candidate to become an official part of the Modelica Standard Library.

[9] Beside the relevant parts in the Modelica specification, e.g., [73, p. 90ff], Lennart and Bachmann [82] provide a comprehensive description of the initialization process.

5. Enabling Modelica for Function Development — *ACG-Modelica*

This chapter is an extensively revised and extended version of Section 4 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`.

5.1. Motivation

The aim of this chapter is to propose extensions and restrictions to the Modelica language that improve the suitability of Modelica as high-level modeling language for safety-related, model-based function development.

To achieve this, the impact of a safety-relevant development process (relying on validated tools) to high-level, domain-oriented modeling languages will be investigated. Based on this analysis I introduce a sub- and superset of Modelica that is simple in order to facilitate high assurance designs, yet expressive enough to allow modeling of many control strategies of practical relevance. *This sub- and superset is in the following referred to as* ***ACG-Modelica***.

A practical control example illustrates the use of the proposed language elements in a servo system model presented in Chapter 6.

5.1.1. Language Complexity

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

When discussing the use of Modelica in the context of control application, advanced control concepts based on inverted plant dynamics are often described [111], [112], [29]. Some Modelica tools are capable of automatically synthesising such controllers and generate code for them. This usually requires fairly sophisticated symbolic manipulation capabilities by the tool. For

example, it may require to differentiate a subset of the equations, select appropriate states and solve the resulting system of differential and algebraic equations numerically.

However, the intrinsic complexity in the resulting control algorithms imposes an additional burden if the topmost design goal is in providing high assurance control systems. Therefore the considerations that guide the design of *ACG-Modelica* are:

- Design trade-off is *biased to prefer high assurance control* over high performance control systems.
- Use of *language restriction* as primary means to keep the complexity of the language as low as possible.
- *Careful language extensions* where deemed necessary to satisfy typical requirements expressed by developers of embedded control systems.
- Modifications to established Modelica simulation tools in order to support *simulation* of *ACG-Modelica* models should be kept as small as possible (note that production code-generation from models may differ significantly from simulation of models as described in Section 5.2.2).
- The requirements specified in Chapter 3 need to be covered.

5.1.2. Terminology

The following list defines some key terms used subsequently.

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

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

Clocks Clocks provide an activation signal or *clock signal* used for synchronous scheduling of a set of equations activated by that clock signal. They recently entered the Modelica language standard (see Section 4).

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

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

5.2. System Compilation: From Function Model to Target Binary

Section 2.3 briefly reviewed a typical model-based development process and outlined the structure of a corresponding toolchain in Figure 2.2.

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

In order to be usable in (safety-relevant) industry applications a development process and its associated development tools need to satisfy the requirements formulated in Chapter 3. Typically, this places constraints on the characteristics of suitable code generation model and the corresponding development tool. Figure 5.1 illustrates the system compilation process. Note that the input to the system compiler, the code generation model, has been constrained to a sub- and superset of the Modelica language (i.e., *ACG-Modelica*). This is in order to allow satisfying requirements from Chapter 3 and will be explained in more detail later.

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

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

5.2.1. Tool Qualification

As noted in Section 2.5, the development of safety-relevant software needs to typically comply to rules described in functional safety standards (Section 2.5.2). This also affects the used development tools. A development tool may need to fulfil a certain level of qualification depending on the severity of the impact a malfunctioning in the development tool is expected to have on a safety-related item. For example, in the automotive domain the ISO 26262 [60, Section *11 Qualification of software tools*]) establishes a set of recommendations concerning the needed qualifications of such software tools.

In order to increase the confidence in the toolchain depicted in Figure 5.1 it is, in principle, possible to apply a number of techniques that have been proposed for ensuring safety of compilers. The survey of Frank et al. [45] gives an overview of existing techniques and also comments

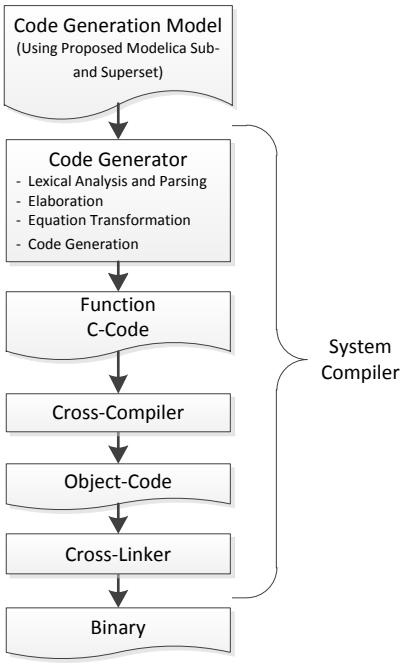


Figure 5.1.: The concatenation of a code generator and a (cross) compiler can be treated as a (system) compiler that directly transforms from the code generation model to the executable (target) code. The Figure indicates that the system compiler is based on a Modelica sub- and superset. Note that this system compilation process may be realized significantly different than a compilation process used in order to create *simulation* executables of models (see Section 5.2.2).

on the suitability of the presented methods for practical application. It identifies three classes of compiler safety approaches:

Language Restriction Restrict implementation languages.

Verification Prove program correctness.

Testing Test resulting programs to detect errors.

In the concluding part of [45] it is stated that "*the scaling problem in particular with respect to verification approaches is largely open*" and that "*verification approaches have been investigated typically for restricted case studies, that is for small prototypic compilers or generators, while the application in practical automotive systems needs a transfer of the results onto real languages and programs.*".

This may explain why today's industrial-strength compilers are typically qualified by comprehensive *test based* approaches. A practical test based approach which has been successfully applied for the qualification of development tools in the automotive industry is tool qualification by use of a *validation suite* [94]. However, also test based approaches are subject to scalability

issues. Therefore, suitable *language restrictions* are an important prerequisite before starting qualification efforts for complex real-world languages.

In order to keep the qualification effort under control, the input language of the code generator models (see Figure 5.1) should be as simple and well-defined as possible, especially with regards to the number and complexity of basic constructs in the language, while also minimizing the number and complexity of performed transformation rules in the code generation process. This has been pointed out by Thiele et al. and motivates the language subset for Modelica proposed in [109] which is further refined in this thesis.

5.2.2. Typical Modelica Code Generation

The typical Modelica code generation process is fairly complex which imposes an additional burden to efforts of tool qualification. This section provides a short overview over the typical code generation process in order to expose the difficulties.

Using an adequate language subset allows a simplified code generation process that facilitates tool qualification efforts. Section 5.9 describes the translation of a Modelica language subset to a synchronous data-flow kernel for which such simplified code generation techniques exists.

Compiling Modelica code usually involves substantial code transformation. Figure 5.2 gives an overview of the compilation and simulation process as described by Broman [22, p. 29].

The different phases are:

Lexical Analysis and Parsing This is standard compiler technology.

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

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

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

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

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

1. In the **Elaboration phase** the instance hierarchy of the hierarchically composed model is collapsed and *flattened* into one (large) system of equations, which is subsequently

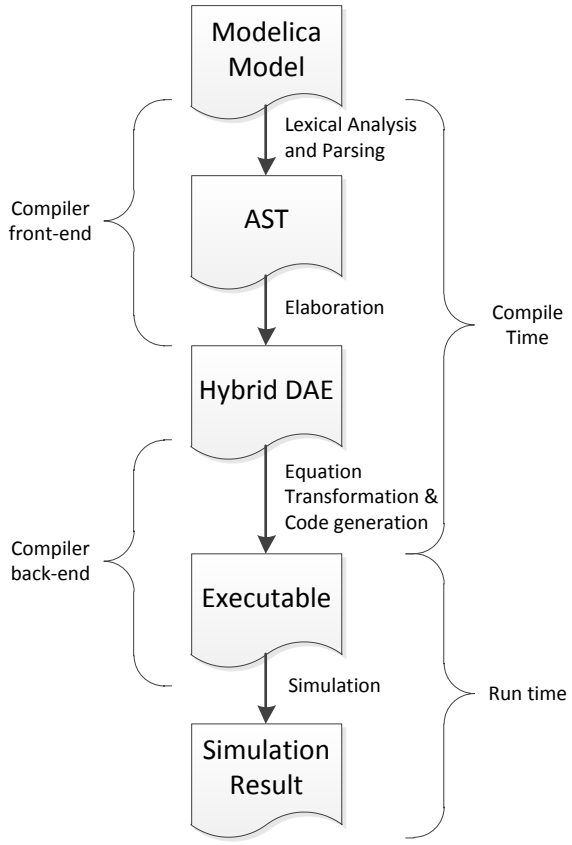


Figure 5.2.: Outline of a typical compilation and simulation process for a Modelica language tool [22, p. 29].

translated into one (large) chunk of C-code inhibiting modularisation and traceability at the C-code level. That conflicts with Requirement 10.

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

Therefore, the compilation process for *simulation* may be significantly different to the *target code compilation* process depicted in Figure 5.1. Not only because different compilers are used, but also because the target code generator may (need to) be an entirely distinct piece of software that may share only minimal to no amounts of code with the simulation code generator. In particular the target code generator depicted in Figure 5.1 is only required to understand the sub- and superset of the Modelica language intended for (discrete) software application models (*ACG-Modelica*).

5.3. Utilization of Suitable Model of Computation

5.3.1. Relevance of Formal Model Specifications

A *formal notation* is a description that has both its syntax and semantics completely defined. Why is this particularly relevant in the context of safety-relevant high-level application development?

A basic approach to minimize risk in safety-related settings is to prevent dangerous system failures or to ensure that they fail in a controlled manner. System failures can be caused by *systematic failures* or *random hardware failures*.

Systematic failures are failures due to human errors within different phases during the product life time cycle, i.e., specification-, design-, implementation-, installation-, and operating errors. *Random hardware failures* are failures due to the limited reliability of hardware elements.

The usage of formal model specifications and related methodology provides assistance in ***avoiding systematic faults***.

5.3.2. Models of Computation

In theoretical computer science a model of computation (MoC) is a formal abstraction mechanism of program execution in a computer. MoCs provide mental models to express, understand, discuss, and analyze computational execution.

Depending on the problem at hand, a particular MoC might result in a superior abstraction than another MoC. For that reason different MoCs coexist and for some scenarios a combination of several MoCs is useful, e.g., the use of continuous-time DAEs for physical plant modeling with discrete-time data flow for digital controller modeling. Research projects like Ptolemy [39] are explicitly committed to study the interaction of different MoCs, particularly in the context of embedded systems software.

Especially in the context of safety relevant applications it is important that the used computational abstractions allow to express relevant scenarios in an unambiguous, clean, comprehensible, and verifiable way. In the domain of sampled data systems the synchronous MoC is an established abstraction that has been utilized successfully, particularly for safety related applications, see e.g., [16]. For that reason it is also the MoC of choice for the language sub- and superset that I propose for modeling high-level control applications in Modelica.

Section 5.3.3 works out the essentials of the synchronous paradigm, while Section 4 provides an introduction to new synchronous language extensions to the Modelica language, to which the author has contributed. Based on these fundamentals further extensions (which have not yet entered the standard specification) are proposed and analyzed in order to fulfil the requirements expressed in Chapter 3 and support formal control design methods for safety relevant system functions.

The interaction of formal models of different domains in a control systems engineering context is illustrated in Figure 5.3. As a matter of fact, the high-level classification in *Software* and *Physics* needs to be refined by precise formal models in order to allow the validation, verification and code-generation tasks as indicated in the diagram.

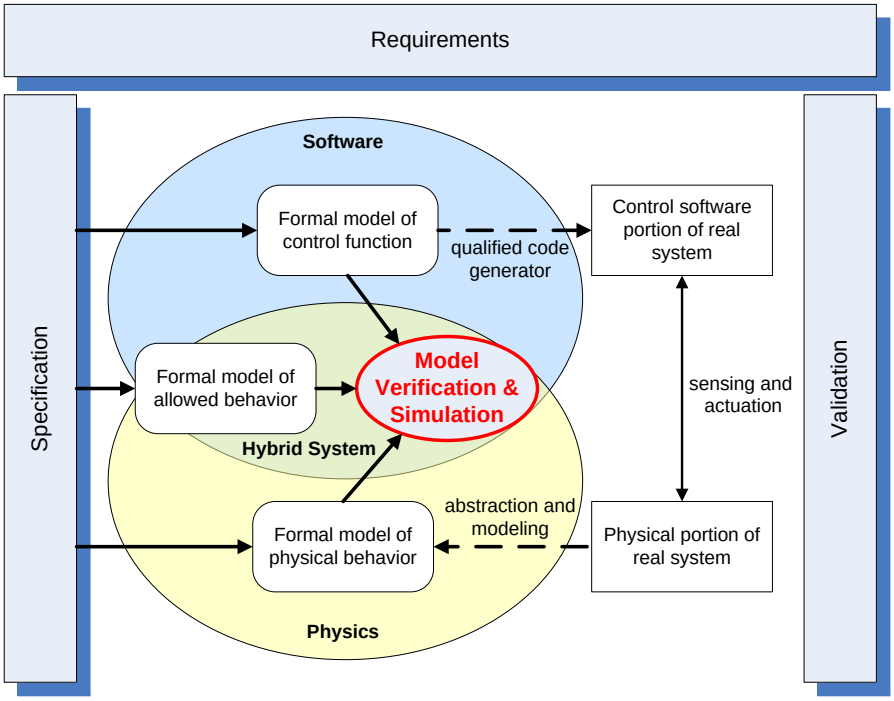


Figure 5.3.: Interaction of formal models in a typical control systems engineering context.

5.3.3. Modelica's Synchronous Model of Computation

While Modelica is a modeling language for hybrid systems, the MoC of the synchronous elements extension can be considered individually, without taking any effects of the continuous-time elements of Modelica into account. As already noted in the motivation of Section 5.3.2 it is usually beneficial to use a MoC that is targeted to the particular use-case, instead of striving for the most general one. The paradigm adapted by synchronous languages is particularly suited for the definition of the "*function specification model*" of embedded control systems, i.e., the input to the model-based development toolchain as indicated in Figure 2.2.

Unsurprisingly, the MoC of the synchronous language elements is similar to the semantics described for the respective synchronous languages that motivated its introduction to Modelica, i.e., [16, 35, 87] (see Section 4 for an informal introduction to Modelica's synchronous language elements extension).

In order to succinctly characterize the synchronous MoC used by Modelica I loosely follow the approach of Benveniste et al. in [15, Section 2.1. *The Essentials of the Synchronous Paradigm*] and work in the Modelica specifics.

1. A model executes by reacting to an infinite sequence of clock ticks (events), which can be informally written by the "pseudomathematical" statement

$$P \equiv R^{\omega},$$

where the superscript ω indicates the succession of activation instants due to clock ticks.

2. Variables only have a value (are active) if their associated clock ticks. Otherwise, they are absent and cannot be accessed.

3. Equations impose relations between variables with the same associated clock which have to be fulfilled concurrently. Rate transition operators exist to allow relations between variables associated to different clocks. Parallel composition is therefore given by the conjunction of associated reactions:

$$P_1 \parallel P_2 \equiv (R_1 \wedge R_2)^{\omega}.$$

4. Computation and communication at an event instant does not take time.

5. The total number of equations is identical to the total "number of unknown variables" (this is called the *single assignment rule* in Modelica [73, p. 88]).

The two commonly used implementation schemes for synchronous programs are sketched in Figure 5.4. Obviously, the right scheme is the implementation model that fits for the synchronous element extension in Modelica.

```
Initialize Memory
for each input event do
   Compute Output
   Update Memory
end
```

```
Initialize Memory
for each clock tick do
   Read Inputs
   Compute Outputs
   Update Memory
end
```

Figure 5.4.: Event driven (left) and sample driven (right) execution scheme for synchronous languages.

5.4. Language Restrictions

Suitable language restrictions have been identified as an important requirement in order to keep the complexity of the language as low as possible and hence, facilitate (tool) qualification activities (see Section 5.1.1 and Requirement 7). Restricting allowed language elements is typically an integral part of *modeling* or *coding guidelines*[1] (see also Requirement 14).

In this section I propose several (to some degree arguably subjective) rules, to enable the use of Modelica for safety-related control applications. The definition of the rules is guided by the requirements for model-based function development given in Chapter 3. The rules resemble

[1] The use of modeling/coding guidelines is a well established practice for safety-related software projects. For example the functional safety standard ISO 26262-6 recommends the use of coding guidelines for all ASIL levels and the functional safety standard IEC 61508-3 highly recommends the use of coding standards for SIL 3 and above [59, 57]. Several coding/modeling guides published by The Motor Industry Software Reliability Association (MISRA) provide standards specifically targeted (but not limited) at the automotive industry, the most famous one being MISRA C [106].

modeling guidelines, however for the following reasons they are *not* equivalent to the modeling guidelines referred to in Requirement 14:

- Section 5.4.1 is primarily focused on *defining a language subset* of Modelica that is deemed suitable to be used for safety-related function development and is not particularly concerned with *how* such an "approved" language is then used by developers!
- Section 5.4.2 provides additional rules for the graphical representation that are intended to enable a potentially more efficient development and review process by allowing developers and reviewers to work (mostly) at the graphical level. For that reason, it does also contain some "core" rules for graphical level modeling that are targeted at developers. However, these core rules would need to be extended (amongst others, considering project and toolchain specific characteristics) in order to produce sufficiently detailed and complete modeling guidelines in the sense of Requirement 14.

Note that to meet the requirements defined in Chapter 3 it is not sufficient to solely restrict the language elements to a subset of the current Modelica 3.3 standard specification. In addition it is necessary to introduce a few language extensions. Proposals for respective extensions are presented in Sections 5.5, 5.6, 5.7, and 5.8. The following rule definitions assume that these language extensions are available.

The rules *restrict* the allowed language elements, and hence define a *language subset*. They are presented in conjunction with a clear *Rationale* for the rule and a *Trace* information that references the particular requirements that motivated the definition of the rule.

5.4.1. Modeling Language Rules

This section is the counterpart to the modeling language requirements presented in Section 3.2. It has the aim to propose a set of rules that impose restrictions on the (textual) language level in order to ensure that Modelica meets the previously formulated requirements.

Rule 1 - Clocked Variables Exclusivity. Occurring variables and equations must be part of (discrete-time) clocked partitions. Note that this restriction implies that all allowed high-level applications have a purely time-discrete nature.
Rationale: Clocked variables and equations were introduced in Modelica 3.3 to provide improved support for implementation of (discrete-time) control systems. See Section 4.3 for details.
Trace: Requirement 2, 3, 5, 7, 8 and 9.

Rule 2 - Causal Data-Flow Exclusivity. (1) Equations shall be restricted to a form where a single variable (the unknown of that equation) is at the left-hand side and a flow expression is at the right-hand side, e.g., assume a is the unknown, then $a = b \cdot c$ is valid while $a + b = 0$ or $a^2 = b$ is invalid. (2) The data-flow between input and output connectors must be directed from the input connector to the output connector.
Rationale: (1) The potentially substantial symbolic transformations performed by a Modelica tool to generate sequential code from equations interferes with the ability of a developer to tightly control the evaluation of expressions. The imposed restriction avoids that hazard. (2) The data-flow direction between causal connectors can be inverted in Modelica under certain conditions.

This is potentially counterintuitive to embedded control system experts and is therefore ruled out. See Section 5.8 for a detailed discussion.
Trace: Requirement 3 and 7.

Rule 3 - No Algebraic Loops. There must be no algebraic loops (i.e., causality cycles) within the equations. Algebraic loops occur if a static scheduling of the equations according to data dependencies is not possible due to causality loops, e.g.,

$$x = 1 - y$$
$$y = x.$$

The compiler needs to detect (and reject) algebraic loops during causality analysis.
Rationale: In general, solving algebraic loops needs (potentially unbounded) numerical iteration. This is usually not acceptable for real-time control application. Note that symbolic manipulation would allow to solve the specific example from above ($x = 1/2$) and hence avoid the problematic numerical iteration. However, this would come to the price of a more complex code generation step.
Trace: Requirement 5 and 8.

Rule 4 - Simplified Modifications. At a component declaration a *modification*[2] can be applied to the subcomponents of that component, but not to deeper nested components, e.g., `b(c=2)` is allowed (one nesting level), while `a(b(c=2))` (two nesting levels) is not allowed. Additionally, modifications are restricted to parameter components, e.g., in `b(c=2)` subcomponent `c` must be declared in the class of `b` with the type prefix **`parameter`**.
Rationale: The correct handling of *modifications* is a difficult challenge during Modelica compilation (see [116] for a good exposition of the difficulties). A more restricted form of modifications simplifies the code generation process, hence reducing tool development and qualification efforts. Also, an unrestricted form of modifications impedes modular code generation, since the enclosing component needs to expose all modifiable elements of its subcomponents (and the subcomponents of its subcomponent and so forth). This would lead to potentially less readable and less (memory) efficient code.
Trace: Requirement 7, 8 and 10.

Rule 5 - Atomic Blocks for Highly-Cohesive Components. Blocks representing high-level components with strong cohesion should be preferably modeled as atomic blocks (see Section 5.6.2).
Rationale: The use of atomic blocks requires a conscious decision by the developer. The trade-offs that need to be considered for that decision are described in Section 5.6. Apart from design trade-offs there might also exist hard reasons that require the use of atomic blocks: (1) The generated code shall be executed as an atomic entity (e.g., because its behavior shall be encapsulated in a single C-function in order to allow smooth integration into a given software architecture framework like AUTOSAR), (2) the block should allow to be separately compiled in order to protect intellectual property.
Trace: Requirement 6 and 10.

[2] Modelica modifications may change the value of primitive variables of a subcomponent. See Figure A.4 for a shortened intuitive description, or refer to [116] and [73] for details.

Rule 6 - Reduced Set of Keywords. Table 5.1 reproduces the keywords from the Modelica specification [73, Section 2.3.3]. Keywords that are not allowed in the proposed Modelica subset are stroked through. The semantics of the remaining elements are maintained appropriately[3].
Rationale: Language simplification.
Trace: Requirement 7.

Table 5.1.: Reduced set of allowed Modelica keywords.

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

Rule 7 - Reduced and Restricted Set of Operators. The available Modelica operators are slightly reduced (no `.*` `./` `.+` `.-`) and the arithmetic operators are restricted to scalar types (see Table 5.2).
Rationale: Language simplification.
Trace: Requirement 7.

Rule 8 - Reduced Set of Built-in Functions and Operators with Function Syntax. Table 5.3 specifies the subset of supported built-in functions and operators with function syntax defined in [73, Section 3.7 and Chapter 10, 16 and 17].
Rationale: Language simplification.
Trace: Requirement 7.

Rule 9 - Supported Data Types. The scalar data types `Boolean`, `Real` and `Integer` are fully supported and are extended to support more fine grain control about the underlying hardware encoding as proposed in Section 5.5. `Enumeration` is not supported, the support of `String` is limited to parameter values and constants. Array support is limited. Most (overloaded) operators and built-in functions related to arrays are not supported (see Table 5.2 and 5.3).
Rationale: Language simplification, as well as increase of language expressiveness to satisfy common data type requirements from the embedded systems domain. Functionality provided by the array operators and built-in functions, e.g., scalar product, can be programmed by using the

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

Table 5.2.: Reduced set of allowed operators

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

[a] Note that contrary to [73, Section 3.4] the arithmetic operators `^ * / + -` are limited to operate on scalar types only and the elementwise operators `.* ./ .+ .-` are not available.

scalar operators and loops.
Trace: Requirement 2, 4, 7, and 8.

The built-in variable `time` (see [73, Section 3.6.7]) is not supported.
Rationale: Although this rule follows automatically from Rule 1 (since `time` is not directly accessible in clocked partitions), it seems justified to state it more explicitly in a self-contained rule. In the employed synchronous data-flow MoC "time" evolves by reacting to an infinite sequence of clock ticks (see Section 5.3.3). Any event generator can be the source for that clock ticks. In contrast, physical `time` is a quantity of continuous system simulation and therefore not directly accessible in clocked partitions. If an absolute wall-clock time is needed in the application logic, it has to be passed in from the external environment as a (**`Real`**) input signal.
Trace: Requirement 3 and 7.

5.4.2. Graphical Representation Rules

The following rules provide modeling guidelines for the graphical representation of Modelica models which are motivated by the requirements formulated in Section 3.3. The goal is to enable a potentially more efficient development and review process by allowing developers and reviewers to work (mostly) at the graphical level.

Rule 11 - Library of Approved Classes with Graphical Representation. A library of approved classes which are annotated with information for their graphical representation has to be established. The design of these classes needs to follow following guidelines:

Table 5.3.: Reduced set of built-in functions and built-in operators with function syntax

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

[a] No events are triggered from these functions for the proposed language subset (see [73, Section 16.8.1]).

[b] Only the "*Inferred Clock*" operator variant `Clock()` is supported. The other `Clock(..)` constructors as well as the `sample(u,c)` and `hold(u)` operators are not part of the language subset proposed for embedded target code generation (see Section 5.7 "Clock Blocks and Interaction with the Physical Environment").

[c] Not strictly necessary elements, especially in an ADD project (see Section 2.6) which allows to use the proposal described in Section 5.7.5. Note that `noClock(u)` can be used instead of `superSample(..)` but is additionally suited for the language extension proposal of Section 5.7.2.

[d] Present proposal excludes state machines (see Section 5.10).

- Lean and straight-forward encapsulation of language operators in synchronous data-flow blocks.
- Graphical representations for basic data structures (e.g., interface ports/connectors) and graphical representations for modeling the flow of execution (e.g., conditional constructs, signal routing).
- Additional "convenience" blocks encapsulating typical algorithms/operations used in discrete open- and closed-loop control.
- Blocks should be designed so that extensive testing of the blocks is easily possible, i.e., simple, lean designs are preferred over sophisticated, complex designs.
- Intuitive and unambiguous graphical representation (icon) for every approved class/block.
- Every class/block must have an identifying feature that allows to recognize that it is an element of the approved library.
- Semantics of every basic class/block needs to be completely and unambiguously documented/specified.
- Every basic class/block needs to pass through an adequate approval process which includes thoroughly testing and validation activities.

Rationale: Modelica's language elements have no inherent graphical representation. However, some elements (e.g., classes/blocks) can be annotated with graphical representation information. These elements can be collected in libraries and (given sufficient development tool support) can then be used by developers in a drag and drop style (see Section A.2.2). In order to work (entirely) on the graphical level, it is therefore necessary to provide a library with a sufficiently rich set of (annotated) classes/blocks which offer enough expressiveness to allow the clear and concise specification of discrete open- and closed-loop control algorithms and their related support logic.

The semantics of these blocks must be clearly defined and the graphical representation needs to be unambiguous in order to enable graphical reviews. Furthermore, there must be a visual clue that allows to identify a class/block as being part of the approved library in order to discern such block from classes/blocks which are defined by the developer of the embedded system.

Extensive tests increase the confidence that can be placed in the correct functioning of the blocks. Simplicity facilitates testing and ensures that the semantics can be unambiguously understood. Therefore, simple and lean blocks are preferred over feature loaded blocks.
Trace: Requirement 11, 12, and 13.

Rule 12 - Block Diagrams with Synchronous Data-Flow Semantics. Block diagrams with synchronous data-flow semantics must be used at the graphical model level.
Rationale: Essentially, the synchronous data-flow semantics on the graphical level follows naturally since the synchronous data-flow semantics of the underlying textual level is just "lifted" to the graphical level. Nevertheless, it seems appropriate to emphasis that consequence by stating it as a self-contained rule for the graphical representation.

Block diagrams with synchronous data-flow semantics are an established notation in the domain of control engineering. Also, the existing compilation/translation techniques for synchronous data-flow programs are understood enough to be accepted by certification authorities. Therefore, adopting that model of computation is an appropriate choice for safety-related open- and closed-loop applications.
Trace: Requirement 11 and 12.

Rule 13 - Causal Connectors Exclusivity. Only connectors with directed (causal) data-flow are allowed.
Rationale: Although this essentially follows from Rule 12, it seems justified to state it explicitly in a self-contained rule. In general, Modelica connectors don't denote the direction of the data-flow (see Section A.2.2 and 5.8). However, requiring block diagrams to have a synchronous data-flow semantics implies that data-flow between connected blocks is restricted to directed (causal) data-flow.
Trace: Requirement 11.

Rule 14 - Exclusive Use of Approved Library. Developers may only use elements from an approved library[4] (according to Rule 11). Deviations from this rule are possible — however, in that case the reviews for corresponding elements are required to be done on the textual level.
Rationale: It is only possible to put legitimate trust in the semantics of a composition of elements, if legitimate trust can be put into the correctness of any of the involved elements. This means that it needs to be ensured that any of the involved elements behaves as expected/specified. Hence, developers must exclusively use the elements provided by an approved library. Note that this rule must be part of the *modeling guidelines* requested in Requirement 14.
Trace: Requirement 11.

Rule 15 - Textual/Composite Block Exclusivity. Textual equation must not be mixed with graphical block instances. In exceptional cases it is allowed to deviate from this rule, however, such deviations must be clearly documented and the corresponding block(s) need clearly noticeable visual marks.
Rationale: Mixing textual equations with graphical block instances in one block can be very confusing and dangerous since the semantics of that block cannot be anymore deduced entirely from the graphical level. This clearly would violate Requirement 11. This rule must be part of the *modeling guidelines* requested in Requirement 14. Note, that it also seems highly advisable that a tool provides automatized checks that ensure conformance to this rule.
Trace: Requirement 11.

Rule 16 - Use of Intermediate "(De)mux" Blocks. Direct connections of a scalar connector to and from array connectors are not allowed. Intermediate "(De)mux" blocks must be used when scalar connectors shall be connected with array connectors. In exceptional cases it is allowed to deviate from this rule, however, such deviations must be clearly documented and the corresponding block(s) need clearly noticeable visual marks.
Rationale: Modelica allows to directly connect scalar connectors with array connectors by using array indices, e.g., `connect(a,b[2])`. This is no problem at the textual level, since the semantics is obvious. However, it is usually not possible to recognize the array index at the graphical

[4] Note, that it is of course also possible that a project uses *several* approved libraries

level. The explicit use of "(De)mux" blocks is therefore advisable in order to improve clarity and understandability of models. This rule must be part of the *modeling guidelines* requested in Requirement 14. Furthermore, it is advisable that a tool provides automatized checks that ensure conformance to this rule.
Trace: Requirement 11 and 12.

As already mentioned in the introduction to Section 5.4 the provided rules are not intended to work as complete modeling guidelines in the sense of Requirement 14. The presented rules target selected core issues that seem to be particularly important in order to allow developers and reviewers to work (mostly) at the graphical level. A complete set of practical modeling guidelines would certainly be more comprehensive and would also need to consider project and toolchain specific characteristics.

The high-level application example presented in Chapter 6 contains a discussion about how the rules established in this section relate to the graphical representation of a digital PID controller model (see Section 6.3.3 on page 100).

5.5. Support of Target Data Types and Operations

5.5.1. Limitations of Current Language Standard

Target-Specific Optimizations

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 [52, 56, 44]. 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 [89, p. 78]. Therefore dedicated code generators typically provide fine control for the used data types and support (cross-) compiler specific extensions and inline-assembler macros for producing code optimized for a specific target platform.

Modelica started as a high-level language for multi-domain (physical) modeling and simulation, so the code generated by Modelica tools has been traditionally targeted at conventional desktop computers. Therefore, low-level, hardware-related implementation aspects did not play a prominent role.

This is apparent when looking at the data types supported by Modelica and their mapping to C data types as specified in Modelica 3.3 [73, Section 12.9]:

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

This is not sufficient when targeting embedded systems. Developers will therefore need finer control over the used data types[5]. An additional difficulty is that low-level properties of the data types are not precisely specified. The specification recommends the range of **`Real`** numbers to have at least the range provided by IEEE double precision floating-point numbers and **`Integer`** numbers are recommended to have at least the range of a two's-complement 32-bit integer [73, p. 13]. In addition, there is a **non-normative** comment in [73, p. 52] that states:

> *"For external functions in C89, RealType by default maps to double and IntegerType by default maps to int. In the mapping proposed in Annex F of the C99 standard, RealType/double matches the IEC 60559:1989 (ANSI/IEEE 754-1985) double format. Typically IntegerType represents a 32-bit 2-complement signed integer."*

However, note that Modelica provides no access to some prominent features of IEEE 754 floating-points, like testing for special values such as *Infinity* or *NaN*. The specification [73, p. 16] states: "*If a numeric operation overflows the result is undefined. For literals it is recommended to automatically convert the number to another type with greater precision.*". So, there is no precisely specified behaviour for dealing with exceptional cases (e.g., undefined or unsupported mathematical operation, overflow, underflow, denormalization).

Target-Specific Functionality

Another, more functional motivation to include low-level hardware-aspects into the modeling language is to support accessing hardware devices like analog-digital converters (ADCs), Pulse-width modulation (PWM) units, timers, etc. as well as services due to a (possibly) available "middleware" (i.e., operating system and additional services, like communication services and data management) from within the model. Within certain limits the external function interface of Modelica [73, Section 12.9] allows to integrate such low-level hardware functionality as long as the functionality can be wrapped into a C function that is called from the model during model execution. However, note that direct calls to I/O drivers or other basic software layers is usually unwanted if following an architecture-driven development process. In that case it is necessary to support standardized middleware functions or macros that are "connected" in a subsequent software integration process with the required services provided by the basic software layer.

Also note, that the desired source of the control flow does not always stem from the model execution that causes the execution of code to access the peripherals (e.g., by reading or writing to an ADC), but may also stem from *external events* (e.g., hardware interrupts) that shall *cause the execution of specific control algorithms*[6] modeled in the high-level language. Modelica's

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

[6] A well known example is the execution of a control algorithm for precisely timing the fuel injection and spark ignition in a modern internal combustion engine. In this case the algorithm execution needs to be synchronized with the position of each piston and its working phase. Since the position of the crankshaft is related to each piston position, synchronization can be achieved by sensor technology which measures the engine's crankshaft angular position and triggers a hardware interrupt at specific reference positions. The interrupt *causes* control logic to be executed, e.g., setting timers that in turn cause precisely timed fuel injection and spark ignition at an "optimal" position.

synchronous language elements provide clocks to model activation conditions for a partition of model equations, but it is not possible to explicitly connect a clock to external events like hardware interrupts. Although it should be possible to extend the language to support that use case, this is not proposed for ACG-Modelica.

Instead, it is proposed to delegate the connection of clocks with external activation events into the subsequent software component integration step (see Figure 2.1 at the right-hand side). Therefore, software components, or parts of them, are mapped onto tasks using tools or notations that are specialized for that concern, e.g., AUTOSAR authoring tools when using an AUTOSAR software architecture. This separation of concern is characteristic for an architecture-driven development process as advocated in Section 2.6 (for a good exposition of behavior modeling tools within an architecture-driven development process see Niggemann et al. [77]).

Operators

Bitwise operators. Modelica provides adequate support for arithmetic, relational and logical operators, however, it has *no bitwise operators*. While bitwise operators are not needed for physical simulation, they are important for *efficient* bit manipulation in embedded systems software.

Conversion operators. Conversion between data types is automatic in Modelica if no loss of precision is expected, e.g., when converting from **`Integer`** (signed 32 bits integer) to **`Real`** (double precision floating-point, 53 bits of precision[7]). The inverse conversion needs an conversion operator, e.g,

```
Integer x = integer(1.2);
```

Although **`Boolean`** and `Enumeration` values are mapped to `int` values in the external C interface, explicit conversion between these values is required at the Modelica language level. In summery, Modelica's rules for data type conversion are comprehensible and intuitive.

However, there are subtle difficulties with the automatic conversion. Consider the model in Listing 5.1.

Listing 5.1: Difficulty of automatic conversion in equations.

```
1 model AutomaticConversion
2   Real a, b;
3   Integer c;
4 equation
5     c = a + b; // Does automatic conversion to Real apply?
6     a = 1;
7     b = 2;
8 end AutomaticConversion;
```

[7] The significand of IEEE double precisions floating-points is actually 52 bits, but the leading one bit of a normalized significand is not actually stored. The Modelica specification defines the minimal recommended range of floating-point numbers to that specified for the IEEE double precision floating-points [73, 2.4.1 Floating Point Numbers].

Line 5 declares an equation involving **Real** variables and one **Integer** variable. As described in Section 5.2.2, type checking is typically done *before* the equation causalization process. Consequently, it is not known during the type checking phase which variable a, b, or c, will be assigned to after causalization. However, this information is needed to decide whether automatic conversion of c to a **Real** value is reasonable!

Translating this model with the Dymola tool results in a rather confusing error message[8]. No typing error is detected, but the following error is reported:

```
...
The Real part has 2 unknowns and 3 equations.
The Integer part has 1 unknowns and 0 equations.
...
```

This suggests that the **Integer** variable in line 5.1 has been tentatively converted to **Real**, which makes it an equation for the "Real part". So, although an error is finally detected, it is not detected as type error as one might expect!

The core of the problem is that it is not known during type analysis, which unknown variable of an equation is determined by that equation. After transforming the model to a computable sequential code, this determined variable will be at the left-hand side of an assignment statement and its declared type consequently defines the type required at the right-hand side. While the expression at the right-hand side can be automatically converted to another type during evaluation, the type of the left-hand side is fixed by the variable declaration!

Extending the available basic data types as proposed in Section 5.5.2 leads to further complications. For example, consider an automatic conversion from a 32 bit **Integer** to a single precision **Real** (24 bits of precision[9]). This will loose precision for values greater than 2^{24}. Therefore, additional conversion operators need to be introduced to enable compile checks to prevent unconscious loss of precision and motivate the developer to think about possibly harmful conversion effects.

Safety-Related considerations

Although target-aware code optimization of code-generators and compilers is highly desirable for optimizing utilization of embedded system resources, it also introduces additional difficulties in respect to tool qualification. As Stürmer states in [99, p. 5] "authorities generally regard the qualification of an optimizing compiler or code generator for safety-relevant software as problematic, as it is, on the one hand, an extremely complex tool and, on the other hand, its behaviour when applied to optimizations cannot be clearly comprehended". To increase the confidence in C code "safe" subsets of the C language have been defined for use in safety-related projects. The most popular one in the automotive industry is MISRA C. Thomsen discusses the consequences of requiring a code generator to adhere to the (now superseded) MISRA C:1998 version of the standard, which essentially prohibits many target-specific optimizations introduced by code generators [105, 110]. More recent standards published by the "Motor Industry Software

[8] Tested with Dymola 2014.

[9] The significand of IEEE single precisions floating-points is actually 23 bits, but the leading one bit of a normalized significant is not actually stored.

Reliability Association" are more lenient with respect to deviations from the MISRA C rules of automatically generated code, since it has been recognized that code generators are not in the same way susceptible to common coding errors as human programmers are [107]. Nevertheless, a conflict remains between providing a maximal amount of (automatic) optimization and a maximal amount of reliability. As a consequence a trade-off needs to be made between feature completeness and validation costs. For example, it needs to be checked whether the saving gained by supporting a particular data type (e.g., because a cheaper ECU can be used) outweighs the additional costs in (tool) validation.

5.5.2. Proposal for Data Type Extension

A severe limitation of Modelica with respect to embedded system development is the absence of fine control over the used data type. This section will therefore propose a mechanism to extend the standard Modelica data types with more low-level hardware encoding information. In the automotive industry the use of fixed-point micro controllers units has a high significance, due to achievable cost savings in a high-volume mass-production setting. However, the engineering effort for realizing fixed-point designs is significantly higher than for floating-point designs (Vitkin et al. report a reduction of development time of 80% when using floating-point instead of fixed-point implementations and expects a significant shift toward floating-point targets even for high-volume products [113]). A notable alternative to dedicated fixed-point code is the use of floating-point code in conjunction with optimized compiler libraries for doing floating-point arithmetic on fixed-point processors. Anoop and Betta report that execution times achievable by using floating-point code with floating-point libraries can even outperform fixed-point code [6]. With respect to qualifying a code-generator it needs to be considered that the validation effort will raise for every supported data type and associated operations. Therefore, it needs to be carefully calculated whether a dedicated fixed-point implementation is profitable for a particular project.

Since the primary concern of the thesis is to establish the conditions that need to be met in order to enable safety-related function development using Modelica, it is necessary to make a *trade-off* between feature completeness and associated validation costs. Given that the additional effort introduced by fixed-point support for development *and* development tool validation may possibly not offset achievable cost savings due to cheaper hardware, a *fixed-point extension is not included* in the Modelica sub- and superset proposed for automatic code generation[10].

The relevant part of the Modelica specification defining the basic data types is [73, Section 4.8]. The notation in the specification is adapted to extend the definition of the `Real`, `Integer` and `Boolean` data types. The predefined types

`ModelicaServices.Types.ImplementationType` and

`ModelicaServices.Types.ImplementationRealType`

define the data types supported by the respective tool. The choices annotation is used to designate the possible string values (see [73, p. 78]). Since a string is used instead of an enumeration,

[10] Note that if fixed-point arithmetic is required, it is still possible (though not convenient) to use the proposed Modelica language extensions to implement and validate custom basic blocks that provide the required functionality or compile using software floating-point support as described above.

it is easy for different tools to extend the supported data types if needed. The following names of data types are proposed to be standardized.

```
type ImplementationType = String annotation(choices(
  choice="UInt8"  "8-bit unsigned integer",
  choice="SInt8"  "8-bit signed integer",
  choice="UInt16" "16-bit unsigned integer",
  choice="SInt16" "16-bit signed integer",
  choice="UInt32" "32-bit unsigned integer",
  choice="SInt32" "32-bit signed integer"
));

type ImplementationRealType = String annotation(choices(
  Single "IEEE 754 single precision floating type",
  Double "IEEE 754 double precision floating type",
));
```

Since every supported data type increases development and validation costs for a development tool it needs to be carefully considered whether only supporting a subset of the standardized data types is already sufficient.

Using the definitions, above the predefined types of Modelica are extended with the additional attribute `implementationType`. Some attributes specified in [73] are not supported and are crossed out. The rationale for not supporting an attribute is given in the corresponding footnote. Note that the types are defined with Modelica syntax although they are predefined data types in Modelica.

```
type Real
  RealType value; /* Accessed without dot-notation */
  parameter StringType quantity;[11]
  parameter StringType unit;
  parameter StringType displayUnit;[11]
  parameter RealType min=-Inf, max=+Inf;
  parameter RealType start;[12]
  parameter BooleanType fixed;[13]
  parameter RealType nominal;[14]
  parameter StateSelect stateSelect;[15]
  parameter ImplementationRealType implementationType = "Double";
end Real;
```

[11] Omitted for the sake of language simplification (Requirement 7).

[12] Missing required start values are detected during compile time analysis (Requirement 5). In case that a tool deduces some start values automatically, the resulting start values need to be fixated for code generation (in compliance with Requirement 9).

[13] The Attribute "`fixed`" cannot be applied on clocked discrete-time variables. It is true for variables to which the `previous()` operator is applied, otherwise false [73, Section 16.9].

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

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

```
type Integer
  IntegerType value; /* Accessed without dot-notation */
  parameter StringType quantity;[11]
  parameter IntegerType min=-Inf, max=+Inf;
  parameter IntegerType start;[12]
  parameter BooleanType fixed;[13]
  parameter ImplementationType implementationType = "Sint32";
end Integer;

type Boolean
  BooleanType value; /* Accessed without dot-notation */
  parameter StringType quantity;[11]
  parameter BooleanType start;[12]
  parameter BooleanType fixed;[13]
  parameter ImplementationType implementationType = "Sint32";
end Boolean;
```

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

Extending available data types also requires to extend the semantics of operators in respect to the new data types. From a high-level viewpoint, the semantics of available Modelica operators carry over naturally to the extension, since I "just" allow to approximate integer and real numbers with a different implementation type. The developer is responsible to ensure that no exceptional cases, e.g., arithmetic overflow or division through zero, can occur within the specified operation conditions.

However, there are some semantic subtleties when evaluating terms that involve integer or real expression of different ranges or precision. This will be discussed in the following section, in which necessary extensions to the available conversion operators are discussed.

5.5.3. Proposal for Operation Extensions

Conversion Operators

Despite the subtle difficulties of Modelica's automatic conversion rules pointed out in Section 5.5.1 it is possible to maintain those rules under appropriate conditions in *ACG-Modelica*. Due to the proposed restriction of the language sub- and superset to causal data-flow semantics (see Section 5.8.2), it is already known during type analysis which unknown variable is supposed to be determined by a particular equation. Consequently, type analysis can detect a type error in equations similar to the equation in line 5 of the Listing 5.1 on page 53.

Therefore, in line with the conversion philosophy of current Modelica, it is proposed to provide automatic conversion between data types if no loss of precision is expected. Otherwise, the use of explicit conversion operators is required. The already available conversion operator **`integer`**`(x)` is extended to take a second element that designates the data type that `x` shall be converted into (defaulting to `"SInt32"` if the argument is omitted which provides backwards compatibility). Additionally, the operator is overloaded, so that it can take any of the supported **`Integer`** or **`Real`** types. It returns the largest integer not greater than `x` of the specified `implementationType`.

```
integer(x, implementationType = "SInt32")
```

Similarly, a new conversion operator for conversion into **Real** values is introduced. Also this operator is overloaded, so that it can take any of the supported **Real** or **Integer** types, though the only conversion from an integer expression to a real expression that requires explicit conversion is that of converting from a 32-bit integer to a single precision floating-point (see Section 5.5.1). Rounding is performed according to the floating-point rounding mode of the target language[17].

```
real(x, implementationType = "Double")
```

Note that there is no need for a conversion operator for **Boolean** values, e.g., a **Boolean** value stored in an underlying `SInt32` implementation type can be assigned to a **Boolean** with underlying implementation type `UInt8` without the danger of overflow.

A remaining point is whether numbers may be automatically converted to another type with greater precision/range in order to perform an operation, e.g., what is the expected outcome of the following expression:

```
Integer u16a(implementationType=UInt16) = 32768; // = 2^15
Integer u16b(implementationType=UInt16) = 32768;
Integer u32a(implementationType=UInt32) = u16a + u16b;
```

The outcome depends on whether the addition is performed with 32-bit values (in which case the result will be 2^{16}), or whether the addition is performed with 16-bit values (in which case an overflow occurs and the result is not defined in Modelica).

If this code was translated to C in a straightforward way, e.g.:

```
uint16_t u16a = 32768, u16b = 32768;
uint32_t u32a = u16a + u16b;
```

the result of the addition would depend on the implemented size of `int`. If it was 32-bit, C would automatically convert the values to 32-bit before performing the addition (due to a C concept called *integral promotion*). However, if the implemented size of `int` was 16-bit, an overflow would occur and the result would be 0.

The proposal refrains from guaranteeing automatic widening of types before performing operations. If widening of types is required, it should be encoded explicitly, e.g.:

```
Integer u32b(implementationType=UInt32) =
  integer(u16a,"UInt32") + integer(u16b, "UInt32");
```

Bitwise Operations

Introducing bitwise operations is a bit delicate. Bit manipulation is a low-level task and it may be legitimately disputed whether they should be made available in a high-level modeling language. In order to keep the language as simple as possible I am inclined to do without them!

[17] I refrain from prescribing a specific rounding mode, since the available rounding modes may depend on the target hardware and the cross-compiler. However, the most common rounding mode is the "*round to even*" mode which is the default rounding mode of the IEEE 754 standard [49].

However, efficiency considerations will render them desirable for many developers. Therefore, the following tentative proposal is presented. Table 5.4 lists the proposed bitwise operators. Instead of the familiar operator symbols known from C or Java (`~ & | ^ << >>`), the operators from Table 5.4 have function syntax.

Table 5.4.: Proposed bitwise operators

Operator Syntax	Description
`bitNot(x)`	Bitwise complement of `x`
`bitAnd(x,y)`	Bitwise AND between `x` and `y`
`bitOr(x,y)`	Bitwise OR between `x` and `y`
`bitXor(x,y)`	Bitwise Exclusive OR (XOR) between `x` and `y`
`bitLeft(x,n)`	Left shift bits of `x` by `n` positions
`bitRight(x,n)`	Right shift bits of `x` by `n` positions

The operators from Table 5.4 are only defined for the unsigned `Uint8`, `Uint16` and `Uint32` types, and the return type equals the type in which the operation was performed.

5.5.4. Alternative Solutions

A rather suggestive alternative to the data type extension proposal of Section 5.5 is to extend the data types using Modelica's short class definitions to specialize the `Integer` type (similar definitions can be introduced for **`Real`** and **`Boolean`** data types), e.g.:

```
type UInt8 = Integer(min=0, max=255);
type SInt8 = Integer(min=-128, max=127);
  ...
type SInt32 = Integer(min=-2147483648, max=2147483647);
```

While a code-generator could use this information to generate respective target platform code (e.g., mapping `UInt8` to `unsigned char` when targeting C) a simulation tool could just check that no range violation occurs.

However, developing a (graphical) library would require to duplicate blocks that perform basic operation (e.g., addition or multiplication) for *any* of the supported data types (connector declarations need a commitment to a specific data type, see Section A.4). In contrast, the solution proposed in Section 5.5.2 allows to implement library blocks for which the "`implementationType`" of all the respective variables and interface declaration within a block can be adjusted by just changing one parameter which is passed down to all variable declarations.

The proposed solution is an admittedly ad hoc mechanism for a limited form of parametric polymorphism. A more ambitious solution would be to introduce full-fledged support for parametric polymorphism, and possibly a type inference mechanism into the language. However, these are fundamental changes to the language. They do not only impose a huge challenge in respect to language design implications, they also impose high implementation efforts when introducing that features into existing tools.

Another option is to use an already existing, more limited form of parametric polymorphism based on "*redeclaration*" (see Section A.3). Listing 5.2 illustrates the basic idea.

Listing 5.2: Block that can be adapted for different Integer types.

```
1  block IntegerPolymorphic "Block that supports different Integer types"
2    replaceable type ImplementationType=SInt32;
3    IntegerInput u;
4    IntegerOutput y;
5  protected
6    connector IntegerInput = input ImplementationType;
7    connector IntegerOutput = output ImplementationType;
8  equation
9    y = u;
10 end IntegerPolymorphic;
11
12 // Usage example:
13 model Test
14   IntegerPolymorphic a(redeclare type ImplementationType = UInt16);
15 end Test;
```

Line 2 declares the replaceable type `ImplementationType` which defaults to the standard **`Integer`** type used by Modelica (simulation) tools. The **`connector`** class declarations in line 6 and 7 define an input and output connector class using the replaceable type `ImplementationType`. Lines 3 and 4 instantiate these connector classes and therefore define the input and output connectors (ports) of the block. Finally, line 14 shows how the `ImplementationType` can be modified when instantiating the `IntegerPolymorphic` block. Please note that it is strictly necessary to use an **`Integer`** subtype in the redeclaration of line 14. It is *not possible to define a more generic type parameter* in line 2 that would allow redeclaration to either **`Integer`**, **`Real`** or **`Boolean`** data types, although the "=" operator in line 9 would work for any of these data types (as mentioned above, this would require to extend Modelica's support for parametric polymorphism)!

One disadvantage of the "*redeclaration*" based approach is that the underlying mechanism of *replaceable* classes is a rather huge and complex topic in the Modelica language specification. The example may appear straightforward, but it touches only the tip of the iceberg. A full implementation would impose a considerable hurdle for a safety-related code generator. However, it is conceivable that a restricted form, that encompasses the use case depicted in Listing 5.2, could be realized within reasonable efforts.

Another disadvantage is that conversion between the subtypes is automatic and no conversion operators exist that would allow to enforce type safety with respect to specialized types at compile time. Introducing a new type like,

```
type SInt8 = Integer(min=-128, max=127);
```

does *not* introduce a corresponding conversion operator for that type. To remedy that in order to assure strong typing properties requires another language extension. A possibility is to require that any new type declaration implies the availability of a corresponding conversion operator of the same name, e.g.:

```
SInt32 x = 4;
SInt8 y = SInt8(x); // Explicit conversion needed
```

One can argue that the presented alternative data type extension proposal integrates more smoothly into the current Modelica language. On the other hand it seems to imply a higher implementation complexity than the proposal described in Section 5.5.2. Therefore, in order to keep *ACG-Modelica* as simple as possible, the proposal presented in Section 5.5.2 is preferred.

A second, obvious alternative is to introduce the desired data types as fundamental data types of the language (i.e., **`SInt8`**, **`UInt8`**, ... **`UInt32`** would complement the existing fundamental data types like **`Real`**, **`Integer`** or **`Boolean`**). This has the same drawback in respect to designing graphical block libraries that has already been identified for the first alternative proposal (namely, block duplication for any supported data type). However, while the approach illustrated in Listing 5.2 mitigates that disadvantage for the first alternative proposal, it can only be used as long as the "redeclared types" are "subtypes" of the corresponding fundamental data type. Of course, additional rules in the specification could be imposed that allow "redeclaration of fundamental types under certain conditions, e.g., if both fundamental types have an integer type nature, etc.". However, fixing the data types in the specification in this style seems more intrusive and less consistent as the first alternative proposal. Compared to the preferred proposal of Section 5.5.2 the second alternative proposal has then the same disadvantage as the first alternative, namely, a seemingly higher implementation complexity.

As alternative to the bitwise operations presented in Table 5.4, two different options are discussed briefly. Since the syntax for bitwise operations used by the C language is typically well known by (embedded systems) developers it could be reused (in a slightly adapted form) for Modelica. Table 5.5 lists the proposed bitwise operators.

Table 5.5.: Alternative bitwise operators inspired by the C language

Operator Syntax	Description
`~x`	Bitwise complement of `x`
`x & y`	Bitwise AND between `x` and `y`
`x | y`	Bitwise OR between `x` and `y`
`x |^ y`	Bitwise Exclusive OR (XOR) between `x` and `y`[a]
`x << n`	Left shift bits of `x` by `n` positions
`x >> n`	Right shift bits of `x` by `n` positions

[a] The `^`-symbol is already used in Modelica as exponentiation operator, therefore it is not available for using it as bitwise XOR operator.

However, this syntax seems not to fit well into Modelica, where a more wordy syntax is typically preferred, e.g., instead of the operator symbols `&&`, `||`, and `!` used by the C language as logical operators, Modelica uses `and`, `or`, and `not`.

Another alternative is to overload the existing logical operators (which are only defined for **`Boolean`** values) for (unsigned) **`Integer`** values and introduce new operators for the remaining operations. This approach is similar to the approach taken by the Ada language. The resulting syntax is presented in Table 5.6.

Syntactically, this proposal blends well into the current Modelica language. The reason why the syntax in Table 5.4 is nevertheless preferred, is that its function syntax has a particularly low language design and implementation complexity.

Table 5.6.: Overloading the logical operators for bitwise operations

Operator Syntax	Description
`not x`	Bitwise complement of `x`
`x and y`	Bitwise AND between `x` and `y`
`x or y`	Bitwise OR between `x` and `y`
`x xor y`	Bitwise Exclusive OR (XOR) between `x` and `y`
`x shiftLeft n`	Left shift bits of `x` by `n` positions
`x shiftRight n`	Right shift bits of `x` by `n` positions

5.6. Modularization of Dynamic Execution Aspects

5.6.1. Limitations of Current Language Standard

Modularization plays an important role in software engineering. Safety standards like ISO 26262 define criteria to consider when selecting a suitable modeling or programming language. Among them is the requirement that a language must support "modularity, abstraction and structured constructs" [59, requirement 5.4.8]. 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.

For industrial automatic code generators, support for modular code generation is usually an intrinsic requirement [52, 16, 13, 19, 69, 88]. Similarly, the support of modular code generation (Requirement 10) is one of the requirements that has been established and motivated in Section 3.2.

Modularization support of Modelica is mainly based on type compatibility of (block)-interfaces. In most cases that sufficiently captures the principle of abstraction to hide all information, except what is needed to know by a user of the block. However, it is not modular in the sense that it would allow to generate code for a block that is independent from the context in which the block is used. The reason for this is that it provides

1. *no modular encapsulation of dynamic execution aspects*, and
2. *no modular encapsulated initial state*, since the initial state may generally depend on the initial state of the global model.

This has disadvantages:

- It is not possible to generate *modular imperative (C-) code* in which blocks/models are straightforwardly mapped to functions. This
  1. makes it difficult to generate code that is readable, maintainable, and testable (impedes use-cases in which the generated code should be used similar to handcrafted code artifacts in the further development process);
  2. makes it difficult to use Modelica to develop software components, or parts of them, that need to integrate smoothly into an architecture-driven development process (see

Section 2.6), e.g., because algorithmic models encapsulated in blocks need to be encapsulated in (C-) functions in order to map them onto tasks using tools or notations that are specialized for that concern;

3. degrades the size of code footprint since there is no possibility to generate reusable (C-) functions from Modelica blocks;
4. complicates the traceability from Modelica (components) to respective target code;
5. prohibits straightforward separate compilation of the generated code artifacts which has drawbacks in respect to scalability and protection of intellectual property;
6. violates Requirement 10 from Section 3.2.

- A block may give unobvious different results for exactly the same input stream depending on the global model in which it is instantiated (due to different start values chosen during initialization). The example in Figure 5.5 demonstrates the issue. However, note that *this problem doesn't arise for blocks using entirely clocked variables as required in Rule 1 (page 44)*, since the initialization of clocked variables was deliberately defined differently in order to mitigate the identified problem[18]. Dependencies to global model initialization
  1. diminish the value of results obtained during module testing and verification;
  2. lead to a wrong model of the real world. Except for the sensor readings, real-world control applications don't have any information about the global state of the plant in which they are embedded. Given the same sensor readings (input stream), they will always initialize to the same state regardless of the state of the plant in which they are embedded;
  3. is a behavior that can be assumed to be surprising for domain experts who would probably expect (directed) data-flow semantics for block diagrams (see also Section 5.8);
  4. violates Requirement 9 from Section 3.2.

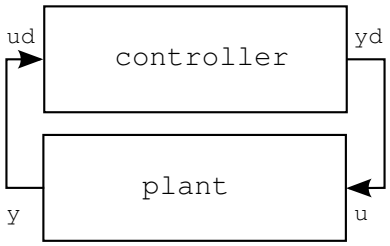
Finally, note that for MBD tools with automatic code-generation the requirements of a suitable modeling/programming style are not limited to the input of the code generators (the "*Code Generation Model*" in Figure 2.2), but to a certain extent also apply to the generated code. Whalen and Heimdahl request that "code output from the code generator must be in a form that adheres to good software engineering practice and follows the standard styles for the target language" [115]. An example for coding guidelines for generated C code is MISRA AC AGC [107], which is also referenced in the ISO 26262 standard [59, requirement 5.4.6].

The following proposal for atomic blocks provides a more restricted form of modularity enabling modular code generation for Modelica blocks as requested in Requirement 10. More details on the code generation step is then provided in Section 5.9.

The idea is to clean-up these issues by introducing the concept of *atomic* blocks. Atomic blocks will allow:

[18] However, there is no way to assure at the interface level that a block using clocked variables doesn't also use un-clocked variables (internally)!

```
1  model ModularityBreachingInitialization
2    Controller controller;
3    Plant plant;
4  equation
5    connect(controller.yd, plant.u);
6    connect(plant.y, controller.ud);
7  end ModularityBreachingInitialization;
8
9  block Controller
10   import Modelica.Blocks.Interfaces.*;
11   RealInput ud;
12   RealOutput yd;
13 equation
14   when sample(0.0,0.1) then
15     yd = pre(yd) + ud;
16   end when;
17 end Controller;
18
19 model Plant
20   import Modelica.Blocks.Interfaces.*;
21   RealInput u;
22   RealOutput y(start=0, fixed=true);
23 initial equation
24   u = 2; // initialization equation for input
25          // u will also affect initial value
26          // of connected output!
27 equation
28   der(y) = -1/2*y + u;
29 end Plant;
```



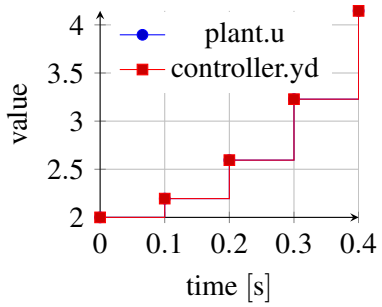


Figure 5.5.: Example for "breaching" the modularity of a control block during initialization. The initial equation in `Plant` requires `u` = 2 during initialization (Line 24). Since `u` is connected to the controller output `yd` (Line 5), this introduces an equality constraint `u` = `yd` which is also active during initialization. As a result, the initial value of discrete state `yd` is not set in the `Controller` block or through an explicit input to that block, but is set "indirectly" by constraints imposed in its embedding environment.

1. Modular encapsulation of dynamic execution aspects with the advantage of
   - improving the suitability to use Modelica as source language for automated code generation targeting safety related designs,
   - enabling the generation of reusable (C-) functions from Modelica blocks. This allows improved code sharing for models instantiating a block several times (reduces size of code footprint) and facilitates using generated code in combination with hand written code,
   - improving the traceability from Modelica (components) to respective target code, and
   - enabling modular compilation of atomic blocks.
2. Modular encapsulation of initial state.

The remaining sections will describe the semantics of atomic blocks in detail and propose syntax to designate atomic blocks.

5.6.2. Proposal for Atomic Blocks

Atomic Blocks

In Section 5.1.2 an *atomic block* is defined as a block that is executed as a single unit akin to a function call with input and output arguments.

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

```
atomic BlockModule a;
```

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

$$(\texttt{out1}, \texttt{out2}, \ldots) = \texttt{BlockModule}(\texttt{in1}, \texttt{in2}, \ldots); \tag{5.1}$$

As a consequence, it is required that input and output signals are active at the *same* clock signal. Note that conventional blocks are not executed as a unit which allows inputs, outputs and equations of the block to be active at completely unrelated times. This includes the possibility that a block with (clocked) discrete-time inputs and outputs may have continuous-time equations internally (that may interact with the clocked inputs and outputs using **`sample`** and **`hold`** operators). This is not possible in atomic blocks and a development tool will reject such models. Figure 5.6 provides an illustrative description about the difference between conventional and atomic block activation semantics.

Besides the constrains imposed on clock activation, atomic blocks also impose stronger causality constrains which will be discussed in the following section.

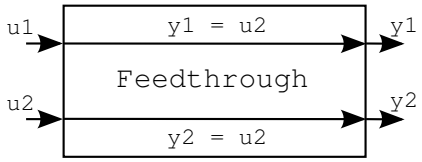


Figure 5.6.: Trivial feedthrough block. *If declared atomic*, it is required that input signals `u1` and `u2` are active *at the same clock ticks* allowing to conceptually transform the block to a (periodically called) function `(y1,y2) = Feedthrough(u1,u2)` (the block hierarchy is maintained at execution level). If *not declared atomic*, `u1=y1` and `u2=y2` are allowed to be *active at completely unrelated points in time* (the block hierarchy is flattened, see Section 5.2.2)!

Balancing conflicting goals in the design of the function model

On the one hand atomic blocks allow for improved *modularity*, *encapsulation*, and *simplicity* of dynamic execution aspects (and thus better *readability* and *comprehensibility* concerning dynamic design aspects), as well as potentially improved *code sharing* with the benefit of decreased code size (e.g., if an atomic block is mapped to a C function that can be reused at several instantiations of that block). On the other hand, they lower language *expressiveness* and potentially decrease *code execution efficiency* (due to the overhead of function calls if an atomic block is mapped to a C function).

Regarding language expressiveness atomic blocks impose stronger causality constraints that need to be respected during equation sorting. Consider the simple Feedthrough block in Figure 5.7. There is no causality loop and for a conventional (non-atomic) block the corresponding equations could be easily sorted: $\texttt{y1} := \texttt{u1}$; $\texttt{u2} := \texttt{y1}$; $\texttt{y2} := \texttt{u2}$. However, for an atomic block

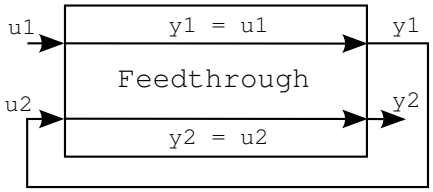


Figure 5.7.: Indirect feedthrough from `u1` to `y2`.

this would introduce an algebraic loop that needs to be solved since it is not possible to generate a single function from the block without introducing a circular dependency between input and output values (due to "$(\texttt{y1},\texttt{y2}) = \texttt{Feedthrough}(\texttt{u1},\texttt{y1})$"). Since algebraic loops are not allowed in ACG-Modelica (see Rule 3), the example would be rejected when using atomic blocks.

Another typical situation in which hierarchy preserving atomic block semantics lead to algebraic loops which do not exist on the flattened equation level is depicted in Figure 5.8. The $\frac{1}{z}$-block denotes a unit delay (corresponding to the Modelica equation `y = previous(u)`). Due to that delay, the feedback loop doesn't result in an algebraic loop at the flattened equation level. However, if the outer block is considered as atomic, an algebraic loop,

$$\texttt{y} = \texttt{AddAndDelay}(\texttt{u1},\texttt{y})$$

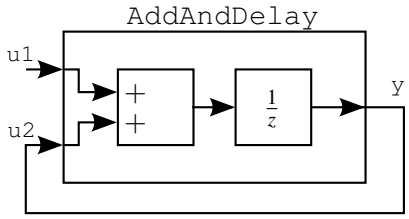


Figure 5.8.: Feedback loop with internal unit delay.

occurs. Breaking that algebraic loop would require to introduce an additional delay between `y` and `u2`.

Note, that the algebraic loop could be avoided if the atomicity requirement would be relaxed, e.g., by splitting the execution of the block in two parts: one for computing the output and one for updating state, e.g.,

$$\texttt{y} = \texttt{AddAndDelay\_Outputs}()$$
$$\texttt{AddAndDelay\_UpdateStates}(\texttt{u1}, \texttt{y})$$

This approach of *interleaved execution* to resolve cyclic dependencies is described by Mosterman and Ciolfi [74] and available in the tool Simulink. Lublinerman et al. [69] extend that concept by deriving an approach for the generation of an optimized *set* of *interface function* for modular code generation. They also prove that the associated optimization problem belongs to a family of problems that is NP-complete. Pouzet and Raymond [88] report that most real programs do not exhibit that complexity and propose a symbolic representation that allows to test whether a specific instance of the problem can be solved with an algorithm of polynomial complexity. Such extensions can clearly improve the flexibility of the code generation process. However, this flexibility comes to the price of additional complexity in the language and the corresponding code generation process. In order to keep both as simple as possible, these extensions are deliberately *not* part of the present proposal.

To sum up, in order to support a form of modularization for which the hierarchical structure of a Modelica model can be preserved when being transformed to sequential target code, the introduction of atomic blocks is proposed. A developer needs to balance conflicting goals when deciding whether atomic blocks shall be used (better readability and comprehensibility concerning dynamic execution aspects, better traceability to sequential target code, improved code sharing), or whether block hierarchies are allowed to be flattened (better language expressiveness and execution efficiency).

A practical design will likely combine both. It doesn't make much sense to use atomic blocks for blocks encapsulating primitive operations (e.g., `+, -, *, /, previous(..)`). However, if a block has considerable internal complexity, high internal cohesion and only loose coupling with surrounding blocks it is an indication to use an atomic block. Beside the intention to use atomic blocks to allow for the generation of better structured, more modular sequential code, they are also useful in use-cases that require to model the behavior of scheduler components that cause the execution of (control) functions (this is discussed in more detail in Section 5.7).

5.7. Manual Block Scheduling

The following section will use the synchronous language elements as a base to realize a mechanism that, sloppily speaking, allows to call `blocks` as functions. This extension is a premise to enable following two use cases:

1. Allow manual scheduling of block execution.
2. Allow smooth integration of generated code into external environments, e.g., AUTOSAR authoring environments.

5.7.1. Limitations of Current Language Standard

Whenever a clock ticks, all equations that are associated to that clock need to be fulfilled concurrently (synchronous model of computation, Section 5.3.3). This allows a declarative style of modeling and offloads the task of *how* to compute it to the Modelica tool. Borrowing from the characterisation of declarative programming given by Lloyd [67], it is stated *what* is to be computed, but not *how* it is to be computed. During the equation transformation phase (see Section 5.2.2) a Modelica tool will sort the equations according to the functional dependencies of the unknown variables, thereby transforming the declarative model into a presentation that can be evaluated by a computer in a step-by-step procedure.

Therefore, the actual computation is deduced from the logical computation model and the result is a *sequential* code that behaves similar to the original logical computation model. *Any computation sequence that respects the functional dependencies of the unknown variables is valid.* Data-flow causality dependencies in the logical computation model restrict possible computation sequences, but apart from that a modeler cannot further influence the computation sequence chosen by the Modelica tool.

However, interaction with external software components and real-time requirements can place additional restrictions on the execution sequence that *cannot be determined by data-flow only*. Therefore, manual scheduling can be necessary in practice. Additionally, if discussing a safety-relevant design with authorities it can often be beneficial to document that a human being has thought of the correct activation sequence rather than a machine.

As consequence, the following section will propose a potentially more controversial extension allowing manual scheduling of block activation. Figure 5.9 illustrates conceptually how a sequence of activations may be enforced by the modeler through the use of a manual scheduling component.

5.7.2. Proposal for Clock Priorities

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

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

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

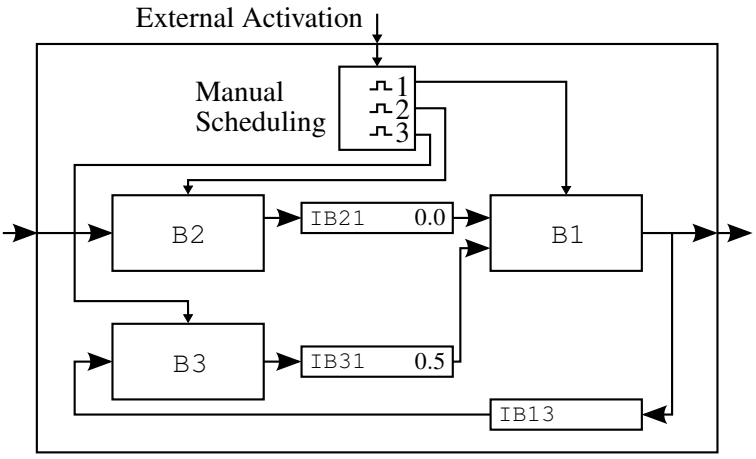


Figure 5.9.: Manual scheduling of blocks.

An example implementation for the scheduler block in Figure 5.9 is given below.

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

The semantics is that the equations activated by a higher priority clock must be executed first[19]. Note that due to the relative nature of clock priorities stated above, it is not possible that a clock has a higher absolute priority than the input clock. Furthermore, the specified priorities must be compatible to causality constraints stemming from dataflow. In order to allow a free choice of the activation sequence it is possible to break dataflow constraints between blocks by inserting decoupling blocks (Figure 5.9 blocks `IB13`, `IB21`, `IB31`). These blocks could be implemented by using the **`noClock`**`(u)` operator (see Section 4.3) with its semantics appropriately extended. If an input signal to a block is required before the signal is provided by the connected output block, it is necessary to specify initial values at the decoupling block (Figure 5.9 blocks `IB21` and `IB31`).

5.7.3. Example

Combining the proposed priority based clock activation with atomic blocks (Section 5.6.2) and modular code generation (building on the approach described in Section 5.9) allows to “call blocks as functions”.

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

To exemplify, assumed modular code is generated for blocks `B1`, `B2` and `B3` from Figure 5.9 and the manual scheduler is modeled by assigning appropriate priorities to the clock signals. A tool could generate following (conceptual) C-code.

```
typedef struct {
  double IB21_y; double IB31_y;
} EnclosingBlock_mem;

void EnclosingBlock_reset(EnclosingBlock_mem *self) {
  self->IB21_y= 0.0;
  self->IB31_y = 0.5;
}

double EnclosingBlock_step(double u, EnclosingBlock_mem *self) {
  double IB13_y;
  IB13_y = B1_step(self->IB21_y, self->IB31_y);
  self->IB21_y = B2_step(u);
  self->IB31_y = B3_step(IB13_y);
  return IB13_y;
}
```

5.7.4. Interaction with the (Physical) Environment

The proposal for manual block scheduling is targeted at the function model, but there is another important use case requiring manual scheduling for simulation purposes. If a function network needs to be simulated in a Modelica tool, it is necessary that the activation of functions by the *scheduler* of the targeted technical system architecture can be adequately modeled.

Software architectures like AUTOSAR provide a model of computation and communication that abstracts from the underlying hardware. The AUTOSAR development partnership has published dedicated documents that deal with the use of behavioral modeling tools within an AUTOSAR driven development process (e.g., [8]). Looking at AUTOSAR's activation model for application software components gives a good example why a mechanism like manual block scheduling is desirable in the context of a high-level, domain-oriented language for model-based development. Also, it is a good example how software architectures may affect the practice of model-based function development as stated in Section 2.6.

The application software components in AUTOSAR are composed of so called *runnables* that encapsulate the programming code. At the code level a runnable boils down to a C-function that is scheduled by the runtime environment. The scheduling is specified by defining certain events that trigger the execution of the runnable. Therefore, the sequence in which runnables are executed is not determined by data-flow dependencies, but by manual specification of activation events. Figure 5.10 shows a model in AUTOSAR notation for the conceptual example model displayed in Figure 5.9. The blocks have been transformed to runnables and the data exchange between the runnables is realized by so called interrunable variables. The diagram doesn't reveal any information about the scheduling of the runnables. The arrows only denote whether a runnable is reading from or writing to a variable. Hence, despite the similarities of Figure 5.10 with the notation used in synchronous data-flow diagrams, it may not be interpreted that way:

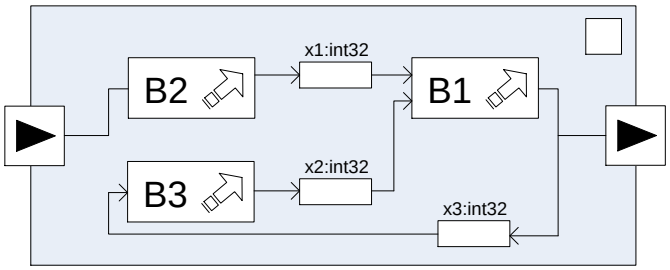


Figure 5.10.: Composition of AUTOSAR runnables: a conceivable mapping of the model displayed in Figure 5.9 to the AUTOSAR software architecture.

manually specified activation events determine the activation sequence of the runnables, not data-flow causalities!

From the code generation point of view, any block depicted in Figure 5.9 could be individually translated into a runnable that could then be imported into an AUTOSAR authoring tool. The assignment of activation events for that runnable could then be performed using an AUTOSAR authoring tool. In this case the activation logic would *not be part of the code generation model* (i.e., not be part of the ACG-Modelica model).

However, in order to capture the software component behaviour adequately within an overall behavior simulation it would be still necessary to have suitable modeling elements that allowed to model the activation events and their effects on the block scheduling. The proposed extension for manual block scheduling allows to model the case that the activation events are logically synchronous, but have to be scheduled in an order that is preset by the execution environment, e.g., because the scheduling order is specified using an AUTOSAR authoring tool. Therefore, the applicability of the proposed language extension is not limited to the code generation, it is also useful for convenient and efficient *simulation* of the sequence of computations performed in an computation environment.

According to Section 5.4, Rule 8 clock source blocks as well as the conversion operators **`hold`**`(..)` and **`sample`**`(..)` must not be part of the high-level application intended for code generation. They are elements needed to *simulate* the execution of the high-level applications within the simulation tool (depicted in Figure 5.11). Or formulated differently, they are idealized models of the environment that will execute the high-level application running on the ECU, e.g., a periodic scheduler of an operating system that activates the high-level application task.

Since clock blocks, sample(u) and hold(u) reside ***outside*** *the high-level application they are not part of ACG-Modelica!*

The Modelica Specification [73, Section 16.3] defines several overloaded **`Clock`**`(..)` constructors. A simple clock source block defining a periodic clock that is active every 0.1 seconds is modeled below.

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

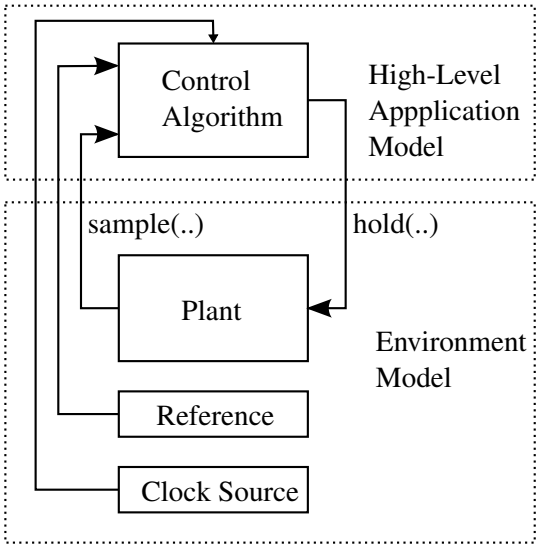


Figure 5.11.: Simulation of a high-level application model using a clock source block to model the execution of the application by its environment, e.g., by an operating system scheduler.

5.7.5. Alternative Solution

The solution proposed in Section 5.7.2 has an obvious drawback: *It increases language complexity considerably*.

As has been indicated in Section 5.7.4, it is possible to *shift the responsibility* for manual block scheduling from the behavioral modeling language to the software integration process and its associated software integration tools (e.g., an AUTOSAR authoring tool). Adopting that pattern, there is no need to support manual scheduling within ACG-Modelica. Furthermore, the support of accommodating data streams with multiple sample rates (e.g., due to sensors measurements or finite computation capabilities) becomes less important, since this function can also be provided by the software integration tool. Thus, one can even consider to dispense with the support of operators `subSample(..)` and `noClock(..)` altogether!

This has the potential to further reduce size and complexity of ACG-Modelica considerably, which is especially beneficial in terms of the expected reduced tool qualification efforts (see Section 5.1.1). However, even if that behavior is not modeled as part of the code generation model (i.e., ACG-Modelica) it must be possible to model that behavior for overall system simulation purposes (i.e., X-in-the-Loop simulation, see Section 2.3).

For the purpose of overall system simulation the complete set of Modelica may be used to create hybrid simulation models. This allows quite detailed models of computation and communication aspects, including the scheduling of software components. A notable account of how Modelica can be used for the simulation of distributed automation systems (including experienced difficulties and devised workarounds) is described by Wagner, Lui and Frey in [114].

However, while [114] is concerned with determining computation and communication delays within a networked automation system (e.g., to enable runtime or delay investigations), the focus

in this section is to model manual scheduling of blocks as motivated in Figure 5.9. This can be achieved by a combination of event iterations with *boolean clock constructors*[20].

Figure 5.12 shows how that idea can be realized in a Modelica component diagram. The

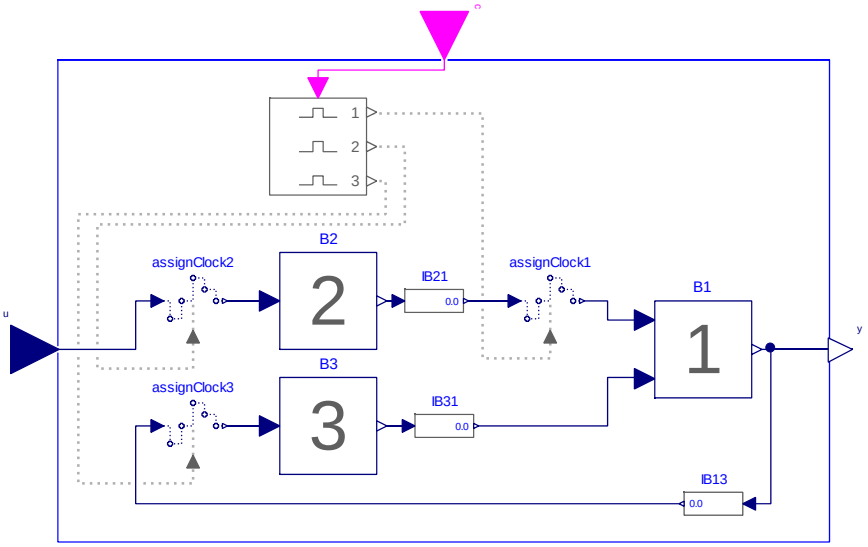


Figure 5.12.: Modelica implementation of the manual scheduling example of Figure 5.9. The clock assignment blocks are from the MODELICA_SYNCHRONOUS library [84], the decoupling blocks wrap the **`noClock`**`(u)` operator and the scheduling component is implemented as proposed in Listing 5.3.

most crucial component is the scheduler. Its implementation (stripped of its annotation and connector declarations for better readability) is given in Listing 5.3. The ordering of the logically concurrent events is achieved by using the **`pre`**`(m)` operator[21]. The operator **`pre`**`(m)` returns the "left limit" of a variable $m(t)$ during an event (e.g., **`pre`**`(m)` would return $m^-(t_1)$ at time t_1 in Figure 4.2 on page 29). At an event instant $m^-(t)$ is the value of m after the last event iteration at time instant t [73, p. 29]. These semantics is used in lines 11 and 13 to ensure that the events are triggered in an ordered sequence. The events are then used in the *boolean clock constructors* in lines 10, 12 and 14. Hence, the clocks are activated in the same order as the corresponding events are triggered.

As a consequence the components that are associated to the respective clocks (by using the "`AssignClock`"-blocks in Figure 5.12) are activated in the expected sequence.

Note that support for the utilized event handling is outside of ACG-Modelica since the Boolean event variables `e1`, `e2`, and `e3` are not *clocked variables* but *piecewise-constant variables* (Rule 1, page 44). Recall Figure 4.2 on page 29 that shows the difference between these two kinds of variables. Also note that it is not possible to use a similar mechanism with clocked variables, since clocked equations do not trigger an event iteration if a clocked variable `v` is changed and

[20] Boolean clock constructors are described in Section 4.4.

[21] The **`pre`**`(m)` operator is described in Section 4.2.

Listing 5.3: Scheduler that triggers event clocks by using an ordered event iteration.

```
model Scheduler
  input Boolean e1;
  output Clock c1;
  output Clock c2;
  output Clock c3;
protected
  Boolean e2(start=false,fixed=true);
  Boolean e3(start=false,fixed=true);
equation
  c1 = Clock(e1);
  e2 = pre(e1);
  c2 = Clock(e2);
  e3 = pre(e2);
  c3 = Clock(e3);
end Scheduler;
```

`previous(v)` appears in the equations.

5.8. Causal Data-Flow Semantics

Modelica can be called an equation-based object-oriented (EOO) language (see Appendix A). Equations are of acausal (or non-causal) nature in the sense that the causality of how the equations have to be solved need not be specified at modeling time (see Section A.2 for details). This is in contrast to the block diagrams used in control systems and DSP engineering applications. In block diagrams the response characteristics of individual components are depicted by blocks with input and output ports. Connections between blocks denote directed (*causal*) data-flow.

Although typical data-flow languages like Lustre also refer to *equations* on flow expressions which define the behaviour of a program, these equations are more constrained than the equations referred to in EOO terminology. The equations definitions in data-flow languages consist of **one** *variable* on the left-hand side and a flow *expression* on the right-hand side. These equations only prescribe causal data-flow (the right-hand side expression is assigned to the left-hand side variable), e.g., in a data-flow assignment equation $a := b \cdot c$ the variable a is assumed to be *unknown*, while b and c are assumed to be *known*. Note that in contrast to assignments in imperative languages the order in which the data-flow equations appear in the program has no relevance. A *single assignment rule* ensures that a variable has no more than one defining equation at a time instant and the set of data-flow equations defining the program behavior is sorted during compilation in order to satisfy data-flow causality constraints between individual equations.

Now, although the foundation of Modelica is rooted in modeling using acausal equations there is also support for a *causal block diagram style of modeling* (see Section A.4).

However, in the context of a language considered for safety-related model-based function development, there are subtle problems with the approach taken by Modelica:

1. There are corner cases where the semantics of Modelica block diagrams deviate from semantics expected for data-flow based block diagrams (contradicts Requirement 12).

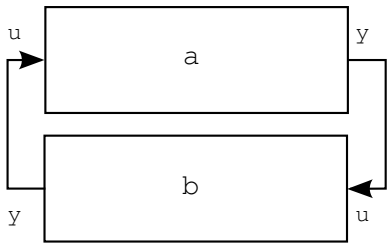
2. The automatic transposition of equations performed by a Modelica tool may interfere with the ability of the developer to tightly control the evaluation of expressions in order to control finite precision issues in calculations or to safeguard potential divisions through zero (Requirement 2 and 8).

5.8.1. Limitations of Current Language Standard

Deviations from Expected Block Diagram Semantics

Figure 5.13 shows an example where the actual behavior of a Modelica block diagram deviates from the semantics typically expected for block diagrams.

```
1  model DataflowInversion
2    A a;
3    B b;
4  equation
5    connect(a.y, b.u);
6    connect(b.y, a.u);
7  end DataflowInversion;
8
9  block A
10   import Modelica.Blocks.Interfaces.*;
11   RealOutput y(start=-0.25);
12   RealInput u;
13 equation
14   when Clock(0.1) then
15     y = previous(y) + u;
16   end when;
17 end A;
18
19 block B
20   import Modelica.Blocks.Interfaces.*;
21   RealInput u;
22   RealOutput y(start=0);
23   input Boolean c = true; // fixed input
24 equation
25   u = if (c) then 0.4 else y;
26 end B;
```



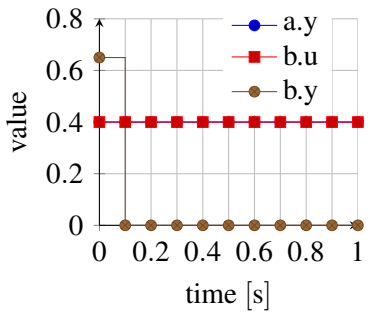


Figure 5.13.: Example for deviation of expected data-flow semantics of block diagrams. The equation for the input `b.u` (line 25) determines the value of the output `a.y`. Therefore, the expected data-flow is inverted.

The output `a.y` is set equal to the input `b.u`, but `b.u` is already fixed to 0.4 by the equation

declared in line 25[22]. The resulting system of equations is:

```
a.y = previous(a.y)+a.u;
b.u = if (true) then 0.4 else y;
b.y = a.u;
b.u = a.y;
```

This system of equations can be easily transformed into a computable sequence of assignments (note that the start values of `a.y` and `b.y` are -0.25 and 0, respectively):

```
// Initial section
a.y_previous := -0.25;
b.y_previous := 0;

// Output at clock ticks
b.u := 0.4;
a.y := b.u;
a.u := a.y - a.y_previous;
b.y := a.u;
a.y_previous := a.y;
```

However, this inversion of input and output causality is not expected for block diagrams that indicate a directed data-flow from `a.y` to `b.u`. Therefore, the observed behavior contradicts Requirement 3 and 7. Note that the example is somewhat artificially constructed (due to line 25), but less obvious situations that lead to unexpected data-flow inversion cannot be completely ruled out by current Modelica semantics. It is worth mentioning that the possibility to invert the data-flow is not a disadvantage per se: it is the prerequisite to allow automatic generation of *inverse* models for advanced controllers [111, 84].

Effects of Symbolic Transformations of Acausal Equations

A consequence of the declarative equation based modeling style is that a Modelica tool may perform substantial symbolic transformations in order to generate sequential code (see Section 5.2.2 and A.2). The symbolic equation transformations performed may interfere with the ability of the developer to tightly control the evaluation of expressions in order to control finite precision issues. It is well known that numerical precision affects the results of computations and the order in which computations are performed matters. For example, for addition and multiplication of floating-point numbers neither associative nor distributive properties hold in general. An illustrative example where the associative law of algebra does not hold for floating-point numbers (of double precision) is given by David Goldberg [49, p. 30]: "$(x+y)+z$ has a totally different answer than $x+(y+z)$ when $x = 10^{30}$, $y = -10^{30}$, and $z = 1$ (it is 1 in the former case, 0 in the latter)".

Please note that numerical precision issues are not a problem specific to EOO languages or Modelica. They are generally encountered in language design and compiler implementa-

[22] In order to have *locally balanced* models the Modelica specification requires that a class/block has the same number of unknowns and equations [73, Section 4.7]. Since inputs are considered known "**block** B" requires one equation for its output `y`. Therefore `y` needs to appear formally in the equation in line 25 in order to make the model valid.

tion (Goldberg provides a notable discussion about the interaction of languages, compilers and floating-point arithmetic in [49]). However, due to the extend of symbolic transformation in Modelica the abilities of a developer to exert manual control over the way computations are performed in the final program is more limited than in more low-level languages like C.

An EOO language specific issue is that the use of equations instead of assignments may render potential divisions through zero less visible. As a consequence, necessary precautions might be *overlooked*. An equation like

$$a = b \cdot c$$

may appear safe at first glance, but of course this depends on the unknown variable which is defined by the context in which the equation is declared. E.g., if the unknown is c a tool will transform the equation to the assignment

$$c := a/b$$

which is unsafe if $b = 0$ cannot be excluded.

Improvement Goals

The aim of the following proposal is:

- Ensure that the semantics indicated by block diagrams do not exhibit any behavior deemed surprising or non-obvious by a domain expert (Requirement 12).
- Restrict equations semantics to the semantics known from typical data-flow languages in order to reduce potential detrimental effects due to automatic transposition of equations (Requirement 2).

5.8.2. Proposal for safer, causal data-flow semantics

Mitigation for the aforementioned limitations can be achieved by diverse means. The preferred one is by extension of the language restrictions imposed in Section 5.4. Alternative solutions are briefly outlined and compared to the preferred solution.

Restrictions on Equations and Causal Connectors

In addition to Modelica's synchronous model of computation described in Section 5.3.3, following restriction applies:

> Equations shall be restricted to a form where a single variable (the unknown) is at the left-hand side and a flow expression is at the right-hand side, e.g., $a = b \cdot c$ is valid while $a + b = 0$ or $a^2 = b$ is invalid.

This restriction does not only prevent potentially undesirable symbolic transformations (as described in Section 5.8.1), but also affects the connections between causal connectors. Together with the *balanced model principle* formulated in the Modelica specification [73, Section 4.7], inversion of data-flow is excluded. The *balanced model principle* requires that the local number

of unknowns[23] (e.g., the unknowns within a block) is equal to the local number of equation. Therefore, any output will require an "assignment equation" while at the same time it is illegal to have an "assignment equation" for inputs, i.e., constructs like line 25 in block `B`, Figure 5.13, would be illegal.

Although inversion of data-flow between directed connectors is prevented by the restriction formulated above, the presented argument is rather subtle and indirect. For improved clarity the following explicit supplementary restriction is proposed:

> The data-flow between input and output connectors must be directed from the input connector to the output connector.

5.8.3. Alternative Solutions

Use of algorithm sections

Recalling standard capabilities of Modelica another option would be to use *algorithm sections* to prevent potentially detrimental symbolic transformation. Instead of equations, *imperative statements* are used in an algorithm section. As illustrative example consider the model in Figure 5.14.

```
1  block AlgorithmExample
2    output Real x(start=0), y(start=0),
3                z(start=0);
4  algorithm
5    x := previous(x) - 1;
6    x := y;
7  algorithm
8    y := previous(y) + 1;
9  algorithm
10   when Clock(0.1) then
11     x := z;
12   end when;
13 end AlgorithmExample;
```

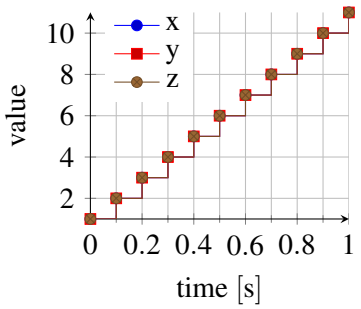


Figure 5.14.: Execution of algorithm sections in a model. The when-statement in line 10 is used to associate a clock to the equations (due to clock inference all equations and variables are assigned to that clock).

The example shows following:

1. *Assignments* are used in algorithm sections, as opposed to equations in equation sections.
2. The statements in the *same* algorithm section are executed in the order of appearance, several assignments to the same variable are possible (always overwriting the previous value).

[23] A connector variable with causality **`output`** is considered as unknown, while a variable with causality **`input`** is considered to be known when counting

3. Statements appearing in *different* algorithm sections are not executed together. Instead, any algorithm section is conceptually viewed as an atomic vector equation $(y_1, y_2, \ldots) = f(u_1, u_2, \ldots)$ where y_i are variables appearing on the left-hand side of assignments, while u_i are right-hand side variables. These equations are then sorted according to data-flow dependencies.

While the use of algorithm sections instead of equations also effectively prevents potentially detrimental symbolic transformation, I see following disadvantages compared to the preferred solution:

- If one prescribes the use of algorithm sections like shown in line 4–6 of the example favorable properties inherent to the single-assignment rule are lost. In this respect I agree with the rating by Ackerman [1, p. 20]: "However, the advantages of single assignment languages, namely, clarity and ease of verification, generally outweigh the "convenience" of reusing the same name".

- To eliminate the disadvantage expressed above, one could require that any assignment statement needs to have its *own* algorithm section as it is the case for the statements in line 8 and 11 of the example. This would give the desired semantics, but to the price of an awkward, unpleasant syntax.

Dedicated language elements for causal data-flow semantics

Instead of imposing additional restrictions on existing elements, another alternative is to define new language elements to achieve the specified goals. Conceivable is to specify a new prefix **`causal`** which can be applied to **`block`** declarations in order to restrict the form of the allowed equations within the block similarly to the rule formulated in the preferred proposal, e.g.:

```
causal block A
...
end A;
```

In order to explicitly ensure directed data-flow semantics between input and output connectors the prefixes **`causalInput`** and **`causalOutput`** can be introduced as an alternative to the existing **`input`** and **`output`** prefixes. The definition of adequate connectors is then similar to the definition presented in Section A.4 for the `RealInput` and `RealOutput` connectors, but instead uses the **`causalInput`** and **`causalOutput`** prefixes, e.g.,

```
connector RealCausalInput = causalInput Real;

connector RealCausalOutput = causalOutput Real;
```

These dedicated language elements have the advantage that they give the developer the freedom to decide on a situation aware basis about the appropriate modeling style. However, the disadvantage is a bloating of the language that hampers comprehensibility, increases language implementation efforts and renders the language less appealing in terms of conceptual clarity and consistency.

This is particular evident in the proposal for directed connectors, where the semantics of the causal variants are almost identical to the existing language elements, except for some very special cases. Therefore, enforcing moderate restrictions on the valid semantics for safety-relevant applications (in order to guarantee the absence of certain kinds of faults), as specified in the favored proposal, seems to be the better solution.

5.9. Translation to a Synchronous Data-Flow Kernel

The language sub- and superset ACG-Modelica was presented as a possible solution to allow the usage of Modelica for safety-related model-based function development. The proposed language restrictions and extensions have been substantiated by providing detailed rationales, including references to open literature where possible. However, despite all these efforts the evidence that it is practically feasible to develop an automatic code generator for ACG-Modelica that is qualifiable for safety related developments remains vague.

To mitigate that deficiency a formal *translational semantics* from a subset of ACG-Modelica (further denoted as mACG-Modelica) to a synchronous data-flow kernel language (further denoted as SDFK) was developed as part of these thesis efforts (Thiele et al. [108]). The synchronous data-flow kernel language is borrowed from Biernacki et al. [19], where Biernacki et al. describe code generation from the synchronous data-flow kernel language into imperative code and note that "The principles presented in this article are implemented in the RELUC compiler of SCADE/LUSTRE and experimented on industrial real-size examples". Furthermore, that synchronous data-flow kernel is considered as a "basic calculus that is powerful enough to express any Lustre program".

The commercial Lustre-based SCADE tool provides a code generator, KCG, that *has already been* successfully qualified for safety-critical applications (see e.g., [50]). Hence, I provide a translational semantics to a data-flow language kernel that is understood enough to be accepted by certification authorities. Consequently, this yields a *strong argument for the feasibility of developing a qualifiable code generator for the considered Modelica subset*. Furthermore, that translational semantics could be used as a base to create a gateway from Modelica to SCADE, similar to what has been reported for Simulink/Stateflow in [31].

The SDFK language semantics as well as the formal translation from mACG-Modelica to SDFK (as described in [108]) is reproduced in Appendix B. The following sections explain that translation *informally* by using an illustrative example. The formal translation in [108] does not include support for Modelica rate transition operators. Section 5.9.4 presents how support for Modelica rate transition operators can be added to the formal translation, i.e., the mACG-Modelica language subset is further extended in order to allow modeling of multirate digital control systems.

The translation to the SDFK language is rather complex. To keep the translation manageable it is subdivided into several steps (see Section B.3) with the two major steps *normalisation* and *translation*. The normalisation is a source-to-source transformation in which mACG-Modelica (see Section B.2 for the language definition) is transformed into a representation that allows a more straight-forward mapping to SDFK. The output of the normalization step is again Modelica code denoted as *normalized* mACG-Modelica. In comparison to ACG-Modelica it has a simpli-

fied grammar (see Section B.5). In the second step *normalized* mACG-Modelica is translated to SDFK.

5.9.1. Intuitive Normalization

The normalization of a simple PI-controller model shall serve as an *intuitive* example. The *formal* translation equations can be found in Section B.4. The PI-controller example is depicted in Figure 5.15. The parametrization of the PI-controller is uninteresting from a control theory point of view, but demonstrates some key concepts regarding modifications. The example consists of input and output connector declarations (line 1 and 2), the declaration of the `PI` block (line 3–12), and block `A` that instantiates the `PI` block (line 13–22).

The right side in Figure 5.15 shows the result of the normalization. The following transformations can be observed:

1. Generation of connection equations.
   - Connector declarations (line 1–2) are replaced by their corresponding short class definition (e.g., `In` $\mapsto$ **`input Real`**, compare line 4 $\mapsto$ 24 and `Out` $\mapsto$ **`output Real`**, compare line 5 $\mapsto$ 25).
   - The connect equations are replaced by simple equations of the form "$x = e$", e.g., **`connect`**`(ya, pi.y)` in line 21 is replaced by `ya = pi.y`. However, since right-hand side (RHS) component dot accesses are extracted and replaced by fresh substitute variables (see next steps further below), the intermediate result `ya = pi.y` is not directly visible in the example. Instead lines 44–45 show the result after the complete normalization.
2. Stripping of parameter modifications, normalizing component modifications and extracting component dot accesses appearing in expressions.
   - Parameter modifications in a block are stripped away (e.g., line 6–7 $\mapsto$ 26–27).
   - All parameters of an instantiated block are extracted, merged with applicable component modifications and introduced as new parameters with a modification expression (compare line 18 $\mapsto$ 38–40).
   - Component dot accesses in RHS equations are extracted and replaced by fresh substitute variables, e.g., the intermediate result `ya = pi.y` from generating the connect equations (see above) is transformed into lines 44–45.
3. Creating a fresh block that instantiates the top-level block as a component with normalized component modifications.
   - If a block is the top-level block for code generation, it needs a special treatment: Parameter modifications in that block should not be lost by stripping them away.
   - To achieve that without requiring a special case treatment in the preceding translation step, a fresh block that instantiates the top-level block as a component with normalized component modifications is inserted.

Connector declarations:

```
1  connector In  = input Real;
2  connector Out = output Real;
```

`PI` block.

```
3  block PI
4    In u;
5    Out y;
6    parameter Real kd = Td*2;
7    parameter Real Td = 0.1;
8    Real x(start=0);
9  equation
10   x = previous(x) + u/Td;
11   y = kd*(x + u);
12 end PI;
```

Block `A` instantiating `PI`.

```
13 block A
14   In ua;
15   Out ya;
16   parameter Real k;
17   parameter Real Td=0.2;
18   PI pi(Td=Td);
19 equation
20   pi.u = ua*k;
21   connect(ya, pi.y);
22 end A;
```

⟼

Normalized `PI` block.

```
23 block PI
24   input Real u;
25   output Real y;
26   parameter Real kd;
27   parameter Real Td;
28   Real x(start=0);
29 equation
30   x = previous(x) + u/Td;
31   y = kd*(x + u);
32 end PI;
```

Normalized `A` block.

```
33 block A
34   input Real ua;
35   output Real ya;
36   parameter Real k;
37   parameter Real Td;
38   parameter Real _pi_kd=_pi_Td*2;
39   parameter Real _pi_Td=Td;
40   PI pi(kd=_pi_kd, Td=_pi_Td);;
41   Real _t1;
42 equation
43   pi.u = ua*k;
44   _t1 = pi.y;
45   ya = _t1;
46 end A;
```

Normalized **instance** block instantiating `A`.

```
47 block _Inst_A
48   input Real ua;
49   output Real ya;
50   parameter Real _a_k;
51   parameter Real _a_Td=0.2;
52   A a(k=_a_k, Td=_a_Td);
53 equation
54   a.ua = ua;
55   ya = a.ya;
56 end _Inst_A;
```

Figure 5.15.: Normalization example for a simple PI-controller model. The left side displays the source models, the right side displays the normalized models including a “fresh” block `_Inst_A` that instantiates the (top-level) block `A` as a component with normalized component modifications.

- Lines 47–56 show the block that is inserted if block `A` is considered as top-level block.
- Block `_Inst_A` instantiates `A` as component `a` in line 52.
- The inputs and outputs of `A` are replicated in `_Inst_A` (lines 48–49) and delegated to the corresponding input and output ports of component `a` by introducing appropriate equations (lines 54–55).

5.9.2. Intuitive Translation

After *normalization*, the model is available in the *normalized* mACG-Modelica language (see Section B.5 for the *normalized* mACG-Modelica language definition). This form allows a more straightforward translation to the SDFK language (see Section B.1 for the SDFK language definition).

Again, the simple PI-controller model introduced in Section 5.9.1 shall serve as an *intuitive* example for the translation to the SDFK language. The *formal* translation equations can be found in Section B.6. The left-hand side of Figure 5.16 shows the PI-controller model after normalization, hence it is the right-hand side of Figure 5.15). The right-hand side of Figure 5.16 shows the resulting model after the transformation to the SDFK.

Following transformations can be observed:

- Blocks are mapped to nodes.
- Inputs and parameters without modification binding are mapped to node input arguments, e.g., lines 2,4,5 $\mapsto$ 35–37.
- Outputs are mapped to node return values, e.g., line 3 $\mapsto$ 38.
- There is a lexicographic order relation imposed on node input and output arguments.
- Delays and the start values of their arguments are mapped to initialized delays (e.g., "`previous(x)`" $\mapsto$ "`0 fby x`", if "`x(start=0)`", compare lines 6,8 $\mapsto$ 40).
- Parameters with modification bindings are mapped to local variables and equations, e.g., line 16 $\mapsto$ 46,49.
- Component modifications and component dot accesses are mapped to function application like node instance calls, e.g., lines 18,21,22 $\mapsto$ 52,53.
- Note that instance wrapper blocks such as `_Inst_A` in lines 25–34 are not treated in a special way—they are transformed by exactly the same rules as the other blocks.

5.9.3. Applicability to the Complete ACG-Modelica Language

The foregoing discussion is based on a subset of the ACG-Modelica language. This allows to keep the scope of the translation within manageable bounds and allows to focus on showing the close correspondence of ACG-Modelica's distinctive synchronous characteristics to well established and understood synchronous data-flow languages.

Normalized `PI` block.

```
block PI
  input Real u;
  output Real y;
  parameter Real kd;
  parameter Real Td;
  Real x(start=0);
equation
  x = previous(x) + u/Td;
  y = kd*(x + u);
end PI;
```

Normalized `A` block.

```
block A
  input Real ua;
  output Real ya;
  parameter Real k;
  parameter Real Td;
  parameter Real _pi_kd=_pi_Td*2;
  parameter Real _pi_Td=Td;
  PI pi(kd=_pi_kd, Td=_pi_Td);
  Real _t1;
equation
  pi.u = ua*k;
  _t1 = pi.y;
  ya = _t1;
end A;
```

Normalized **instance** block instantiating `A`.

```
block _Inst_A
  input Real ua;
  output Real ya;
  parameter Real _a_k;
  parameter Real _a_Td=0.2;
  A a(k=_a_k, Td=_a_Td);
equation
  a.ua = ua;
  ya = a.ya;
end _Inst_A;
```

⟼

SDFK `PI` node.

```
node PI (Td:float,
         kd:float,
         u:float) =
y:float with
var x: float in
    x     = 0 fby x + u/Td
and y     = kd*(x + u)
```

SDFK `A` node.

```
node A (Td:float,
        k:float,
        ua:float) =
(ya:float) with
var _pi_kd:float,
    _pi_Td:float,
    _t1:float in
    _pi_kd = _pi_Td*2
and _pi_Td = Td
and ya = _t1
and _t1 = PI(_pi_Td,
      _pi_kd, ua*k)
```

SDFK `_Inst_A` node.

```
node _Inst_A (_a_k:float,
              ua:float) =
ya:float with
var _a_Td:float in
    ya = A(_a_k, _a_Td, ua)
and _a_Td = 0.2
```

Figure 5.16.: Translation example for a simple PI-controller model. The left side displays the source model (the result of the normalization shown in Figure 5.15), the right side displays the model after translation to the SDFK language.

The multilevel translation approach presented in Section B.3 allows to add new language elements in a way that keeps the lower level untouched. This is thus an attractive way to further

extend the considered language scope as long as a source-to-source transformation into a smaller language kernel can be given.

Hierarchical scoping using *packages* and data typing has been left out of the discussion. Data typing is similar to general purpose languages and is considered to be out of scope of this work. Supporting Modelica **package** declarations to allow hierarchical scoping should be feasible by introducing an additional step in the multilevel translation approach.

A more distinctive omission is the absence of clock conversion operators — hence, the considered subset of ACG-Modelica does not allow to model multirate systems. In the footnote to operators **subSample** and **noClock** in Table 5.3, page 48 it was already argued that it is not strictly necessary to support these elements in the Modelica subset targeted for automatic code generation. So, from a practical viewpoint it seems quite reasonable to omit them. However, from a theoretical perspective it is a bit dissatisfying. Therefore, Section 5.9.4 shows how to extend the presented translation so that it also supports the **subSample** and **noClock** operators in order to allow for multirate models.

Proposed extensions to the Modelica language like extending the supported primitive data types (Section 5.5) and the support for manual block scheduling (Section 5.7) are not part of the presented translation. A formal treatment of data types is orthogonal to the synchronous semantics and is outside of the scope of this work. Language level support for manual block scheduling does not fit well in the declarative synchronous data-flow framework and would increase the language complexity considerably. Because of that, Section 5.7.5 discusses an alternative solution that does not require to extend the language, but still should be applicable to many cases that otherwise would require manual scheduling support at the language level.

5.9.4. *Normalized* mACG-Modelica with clock operators

The *normalized* mACG-Modelica language, as defined in B.5, is extended with the clock operators "**subSample**" and "**noClock**". In order to specify the translation to SDFK all expressions are annotated with clock information. The symbol "a" stands for an expression "e" annotated with its clock "ck" ("$a ::= e^{ck}$"). The fastest clock is denoted as "**base**" clock. All other clocks are related to the base clock with a subsampling relation. A clock "$\downarrow v(ck)$" is a clock that is subsampled from clock "ck" by the (constant) integer factor "v". The clock annotations are inferred by applying a clock calculus to an unannotated model (the modeler does not explicitly specify the clock annotations). Hence, the purpose of the clock calculus is to infer the clock annotations and to reject models that can not be executed synchronously.

The table below gives an example for annotated streams and expressions. Particularly, it shows the effect of operators "**subSample**" and "**noClock**" applied to clock annotated streams.

Stream/Expression	Stream values					
$x^{\mathbf{base}}$	x_0	x_1	x_2	x_3	x_4	...
$y^{\mathbf{base}}$	y_0	y_1	y_2	y_3	y_4	...
$z^{\downarrow 2(\mathbf{base})} = \mathbf{subSample}(x^{\mathbf{base}}, 2)^{\downarrow 2(\mathbf{base})}$	x_0		x_2		x_4	...
$(\mathbf{noClock}(z^{\downarrow 2(\mathbf{base})})^{\mathbf{base}} + y^{\mathbf{base}})^{\mathbf{base}}$	x_0+y_0	x_0+y_1	x_2+y_2	x_2+y_3	x_4+y_4	...

Expressions that only consist of parameters and literals are always accessible, i.e., they can be

used in any clock context. The clock "**const**" is introduced in order to give parameters a clock annotation that marks them as being accessible in any clock context.

The grammar of the *normalized* mACG-Modelica language with clock operators and clock annotations is given below (see Section B.2 for an explanation of the used syntactic categories).

$$
\begin{aligned}
td &::= \textbf{type}\ t; \\
d &::= \textbf{block}\ id\ p\ \textbf{equation}\ D\ \textbf{end}\ id; \\
c &::= \textbf{input} \mid \textbf{output} \\
p &::= p\ p \mid t\ x; \mid t\ x\ mo; \mid c\ t\ x; \mid c\ t\ x\ mo; \mid \textbf{parameter}\ t\ x \mid \textbf{parameter}\ t\ x\ =\ e; \\
mo &::= (ar\ ,\ldots,\ ar) \\
ar &::= id\ =\ e \\
D &::= D\ D \mid eq; \\
a &::= e^{ck} \qquad \text{(clock annotated expression)} \\
e &::= v \mid x \mid op(a,\ldots,a) \mid \textbf{previous}(x) \mid \textbf{if}\ a\ \textbf{then}\ a\ \textbf{else}\ a \\
&\quad \mid \textbf{subSample}(a,v) \mid \textbf{noClock}(a) \\
eq &::= x\ =\ a \mid x\ .\ x\ =\ a \mid x\ =\ x\ .\ x \\
x &::= id \\
ck &::= \textbf{base} \mid \downarrow v(ck) \mid \textbf{const} \\
v &= \text{value} \\
id &= \text{identifier}
\end{aligned}
$$

Similarly to [19, 35] the clock calculus is defined as a type inference system. Figure 5.17 shows the inference rules. A judgement $H \vdash e : ct$ states that expression e has a well-clocked typed ct under the clock environment H. The statement $H \vdash D$ means that D is a set of well-clocked equations under the clock environment H. An environment H is described using the form $[x_1 : ck_1, \ldots, x_n : ck_n]$ where $x_i \neq x_j$ for $i \neq j$.

Noteworthy are rules (C-DPAR1) and (C-DPAR2) that annotate parameters in the clock environment with the clock "**const**". The rule (C-PAR) ensures that the clock of a parameter used within an expression is always inferred from the context, similar to rule (C-VAL) for literal values. Hence, the same parameter can be used in contexts with different clock types ck. This is different to rule (C-VAR) which requires that a unique clock type can be inferred for normal variables. Rule (C-DOT) for component dot accesses requires that all inputs and outputs of a component are on the same clock[24].

Rule (C-SUBSAMPLE) allows to convert a clocked stream to a slower stream. The second argument of "**subSample**" needs to be a literal value. This is more restrictive than the restriction formulated in the Modelica specification [73, Section 16.5.2], which requires that this argument needs to be a *parameter expression*[25]. The idea of a parameter expression is that it can be

[24] See Section 5.6.2 for the motivation behind that restriction.

[25] The term *parameter expression* is defined in [73, Section 16.2.3] as: "The meaning is that the input argument when calling the operator must have parameter variability, that is the argument must depend directly or indirectly only on parameters, constants or literals".

$$H \vdash v : ck \text{ (C-VAL)}$$

$$H, x : \textbf{const} \vdash x : ck \text{ (C-PAR)}$$

$$H, x : ck \wedge ck \neq \textbf{const} \vdash x : ck \text{ (C-VAR)}$$

$$H, c : ck \vdash c.x : ck \text{ (C-DOT)}$$

$$\frac{H \vdash e : ck}{H \vdash e^{ck} : ck} \text{ (C-ANNOTATE)}$$

$$\frac{H \vdash a_1 : ck \ \ldots \ H \vdash a_n : ck}{H \vdash op(a_1, \ldots, a_n) : ck} \text{ (C-OP)}$$

$$\frac{H \vdash a_1 : ck \quad a_2 : ck \quad a_3 : ck}{H \vdash \textbf{if } a_1 \textbf{ then } a_2 \textbf{ else } a_3 : ck} \text{ (C-IF)}$$

$$\frac{H \vdash x : ck}{H \vdash \textbf{previous}(x) : ck} \text{ (C-PREVIOUS)}$$

$$\frac{H \vdash a : ck \quad v \in \mathbb{N}^+}{H \vdash \textbf{subSample}(a, v) : \downarrow v(ck)} \text{ (C-SUBSAMPLE)}$$

$$\frac{H \vdash a : ck_1}{H \vdash \textbf{noClock}(a) : ck_2} \text{ (C-NOCLOCK)}$$

$$\frac{H \vdash x : ck \quad H \vdash a : ck}{H \vdash x = a} \text{ (C-EQ)}$$

$$\frac{H \vdash c.x : ck \quad H \vdash a : ck}{H \vdash c.x = a} \text{ (C-EQDOT1)} \qquad \frac{H \vdash x : ck \quad H \vdash c.x : ck}{H \vdash x = c.x} \text{ (C-EQDOT2)}$$

$$\frac{H \vdash D_1 \ \ldots \ H \vdash D_n}{H \vdash D_1; \ \ldots; \ D_n} \text{ (C-EQNS)}$$

$$\frac{\vdash_{\text{decl}} p : H_p \quad H_p \vdash D}{\vdash \textbf{block } id \ p \textbf{ equation } D \textbf{ end } id;} \text{ (C-BLOCK)}$$

$$\frac{\vdash_{\text{decl}} p_1 : H_1 \ \ldots \ \vdash_{\text{decl}} p_n : H_n}{H_1, \ldots, H_n \vdash p_1; \ \ldots : \ p_n} \text{ (C-DECL)}$$

$$\vdash_{\text{decl}} t \ x; : [x : ck] \text{ (C-DVAR1)} \qquad \vdash_{\text{decl}} t \ x \ mo; : [x : ck] \text{ (C-DVAR2)}$$

$$\vdash_{\text{decl}} c \ t \ x; : [x : \textbf{base}] \text{ (C-DIO1)} \qquad \vdash_{\text{decl}} c \ t \ x \ mo; : [x : \textbf{base}] \text{ (C-DIO2)}$$

$$\vdash_{\text{decl}} \textbf{parameter } t \ x; : [x : \textbf{const}] \text{ (C-DPAR1)} \qquad \vdash_{\text{decl}} \textbf{parameter } t \ x \ mo; : [x : \textbf{const}] \text{ (C-DPAR2)}$$

Figure 5.17.: Clock constraints.

evaluated during translation, in order that clock analysis can be performed during translation. However, changing the value of a parameter may therefore require a fresh translation of the whole model. This impedes modular code generation as requested in Requirement 10. Hence, the more restrictive formulation is needed in the presented framework to allow the translation to modular code.

The rule (C-NOCLOCK) covers the operator "**noClock**". The rule decouples the clock type of its argument from the clock type of its surrounding expressions or equation. Hence, it allows to combine expressions with arbitrary clocks as long as it is possible to infer clocks "ck_1" and "ck_2" by the remaining inference rules[26].

After all expressions are annotated by their clocks the translation to SDFK (described in Section B.6) can be extended to support the **subSample** and **noClock** operators. This will be briefly sketched in the remaining part of this section.

In order to simplify the translation rules it is assumed that **subSample** and **noClock**(a) appear in a normalized form—i.e., that occurrences of **subSample** and **noClock**(a) are extracted from expressions and are replaced by a fresh variable x and an additional equation of the form

$$x \;=\; \textbf{subSample}(a,v) \quad \text{or, respectively} \quad x \;=\; \textbf{noClock}(a).$$

Hence, the normalization step of Section B.4 needs to be adapted to extract **subSample** and **noClock**(a) occurrences from expressions. The details are omitted here, since the normalization is similar to the normalization of RHS component dot accesses in equations (refer to function NBq in Figure B.8 on page 137).

Using the SDFK language a counter that counts modulo n can be defined as:

```
node sample(n:int) = (o:int) with
var cpt:int,
    o:int in
    cpt = if c then 0 else ((0 fby cpt) + 1)
and c = True fby (cpt = n - 1)
and o = (cpt = 0)
```

In the following it is assumed that this auxiliary node definition is available in the context of the SDFK program. Additionally, the translation requires the auxiliary functions BaseFactor defined in Figure 5.18. This function computes the factor v that is required to project a nested clock annotation ck, e.g., $ck = \downarrow 2(\downarrow 3(\textbf{base}))$, to an *equivalent* clock without nesting, e.g., $ck = \downarrow v(\textbf{base})$ where $v = 6$.

Another auxiliary function "cn(v)" is introduced to introduce a variable name for Boolean streams that represent clock activation relative to the base clock. The function returns a fresh identifier for any *new* value of v, i.e., if it is called again with a previously used value of v, it returns the same identifier as before:

$$\text{cn}(v_i) = \text{cn}(v_j) \iff v_i = v_j.$$

The translation rules of Figure B.12, Section B.6 are replaced by the rules given in Figure 5.19 (the utilized notation is described in Section B.4.1 and B.6). The remaining rules in Section B.6

[26] In particular this can be used for combining a previously "subSampled" stream with a faster stream, returning a stream with the clock of the faster stream.

$$
\begin{aligned}
&\text{BaseFactor}(\textbf{base}) && = 1\\
&\text{BaseFactor}(\textbf{const}) && = 1\\
&\text{BaseFactor}(\downarrow v(ck)) && = v \cdot \text{BaseFactor}(ck)
\end{aligned}
$$

Figure 5.18.: Function BaseFactor—Given a clock annotation ck of a clock annotated expression e^{ck}, return the v that allows to replace ck by $\downarrow v(\textbf{base})$.

can be reused with marginally modifications: clock annotations play no role in functions CId and CEq and just need to be preserved (i.e., x can be replaced by x^{ck} and e by e^{ck}), the operators **noClock** and **subSample** are subsumed by the logic given for the n-ary operator $op(a,\ldots,a)$.

Function TEqList from Figure B.12 is modified in Figure 5.19 to include a preceding application of the newly introduced function PEqList. PEqList iterates over the list of equations q and translates them to SDFK equations that are accumulated in the set P_q. Function PEq provides the logic to translate occurrences of operators "**subSample**" and "**noClock**" to SDFK equations. The strategy for doing this is:

1. Upsampling of the operator argument e^{ck_2} to a stream e_b running on the base clock (i.e., $e_b^{\textbf{base}}$) where the last value of e^{ck_2} is kept constant at clock ticks at which e^{ck_2} is not defined.

2. Downsampling of the stream e_b to a stream x running on clock ck_1.

To allow the upsampling and downsampling operations Boolean streams are needed that encode whether a clock is active ("**True**") or not active ("**False**"). This streams are created using the previously defined auxiliary functions "cn" and "`sample`". Note that because P_q is a set, it is ensured that for any i, there is at most one equation of the form $\{\text{cn}(v_i) = \texttt{sample}\ v_i\}$ in the set of P_q (e.g., $P_q \cup \{\text{cn}(v_i) = \texttt{sample}\ v_i\} = P_q \cup \{\text{cn}(v_i) = \texttt{sample}\ v_i\} \cup \{\text{cn}(v_i) = \texttt{sample}\ v_i\}$).

5.9.5. Translator Implementation

A prototypical translator was implemented for the presented translational semantics using the Scala programming language [83] and the Kiama language processing library [96]. The extension with clock operators, as described in Section 5.9.4, requires that a clock analysis phase precedes the translation (which was also implemented).

The translation implementation (without the clock operator extension) is briefly described by Thiele et al. [108] and the following paragraph is a direct citation from that manuscript.

The translator is structured in three main components:

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

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

$$
\begin{aligned}
&\mathrm{PEq}_{(d,s,P_q)} \\
&\quad \left(x = \textbf{noClock}(e^{ck_2})^{ck_1}\right) && = \big(d, s, P_q \cup \{\mathrm{cn}(v_1) = \texttt{sample}\ v_1\} \\
&&& \quad \cup \{\mathrm{cn}(v_2) = \texttt{sample}\ v_2\} \\
&&& \quad \cup \{e_b = \textbf{merge}\ \mathrm{cn}(v_2) \\
&&& \qquad (\textbf{True} \to \mathrm{TE}_{(d,s)}(e)) \\
&&& \qquad (\textbf{False} \to (\textbf{pre}\ e_b)\ \textbf{when False}(\mathrm{cn}(v_2)))\} \\
&&& \quad \cup \{x = e_b\ \textbf{when True}(\mathrm{cn}(v_1))\}\big)\ \text{where} \\
&&& \quad v_1 = \mathrm{BaseFactor}(ck_1), \quad v_2 = \mathrm{BaseFactor}(ck_2) \\
&\mathrm{PEq}_{(d,s,P_q)} && = \mathrm{PEq}_{(d,s,P_q)}\left(x = \textbf{noClock}(e^{ck_2})^{ck_1}\right) \\
&\quad \left(x = \textbf{subSample}(e^{ck_2}, v)^{ck_1}\right) \\
&\mathrm{PEq}_{(d,s,P_q)}(x = e^{ck}) && = (d, s, P_q \cup \{x = \mathrm{TE}_{(d,s)}(e)\}) \\
&\mathrm{PEqList}_{(d,s,P_q)}(q) && = \mathrm{PEq}_{(d,s,P_q)}(q) \\
&\mathrm{PEqList}_{(d,s,P_q)}(q, qs) && = \mathrm{PEqList}_{\mathrm{PEq}_{(d,s,P_q)}(q)}(qs)
\end{aligned}
$$

Translate list of equations into SDFK equations:

$$
\begin{aligned}
&\mathrm{TEqList}_{(d,s)}(q) && = \text{let } (\_, \_, P_q) = \mathrm{PEqList}_{(d,s,\{\})}(q) \text{ in} \\
&&& \quad x_1 = e_1\ \textbf{and}\ \ldots\ \textbf{and}\ x_n = e_n \\
&&& \quad \text{where } \{x_1 = e_1, \ldots, x_n = e_n\} = P_q
\end{aligned}
$$

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

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

Figure 5.19.: Translation from the normalized mACG-Modelica language with clock operators into the SDFK language. The translation functions from Figure B.12, page 140, are modified to support the clock operators "**subSample**" and "**noClock**".

Emitter Emitters to the SDFK language defined in Section B.1 and to Lustre code. The emitter component utilizes the functional pretty printing combinators provided by the Kiama library.

The SDFK output is not executable and needs to be checked statically (currently by manual inspection). The Lustre output allows taking advantage of the software infrastructure that is available around Lustre. Particularly, using Lustre as intermediate presentation allows to generate executable C-code utilizing Verimag's Lustre V4 Toolbox[27]. This C-code can be used for dynamic testing. The translator supports that by allowing to generate appropriate Modelica code adapters ("*C code wrapper blocks*") that provide an interface from Modelica to the generated C code (using Modelica's external function interface). These wrappers can be directly loaded into Modelica simulation environments, enabling convenient back-to-back testing.

Section 6.4 demonstrates the use of the wrapper block feature for SIL validation in the context of a servo system example.

5.10. Further Extensions to ACG-Modelica

This section comments briefly on further extensions to ACG-Modelica that are likely to be of interest to developers.

5.10.1. Inheritance

Inheritance is a powerful concept in object-oriented programming and it is supported in Modelica, but excluded in ACG-Modelica. The Modelica language provides rather intricate ways to combine (multiple) inheritance with structural modifications that highly increase language complexity. Keeping language complexity to a minimum is the reason why inheritance is not supported in the presented ACG-Modelica language. However, extending ACG-Modelica to support a suitably restricted form of inheritance seems quite feasible.

5.10.2. State Machines

While data-flow parts in control applications are naturally described with block diagrams, system logic parts are often described naturally using state machine formalisms. Modelica 3.3 added built-in language elements to support state machines [73, Chapter 17] with a comparable modeling power as Statecharts [53]. Extending synchronous data-flow with state machines has been described in pioneering work of Maraninchi and Rémond on *Mode-Automata* [71, 72] and also by Colaço et al. [34]. Colaço et al. describe a synchronous data-flow kernel language which is extended with language elements for Mode-Automata and provide a translational semantics from that extended language set to the basic synchronous data-flow kernel language. Extending ACG-Modelica with support for state machines in a similar way is desirable, but it is a substantial challenge on its own.

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

6. Servo System Example

This chapter is an extended version of the example that I published in [108].

6.1. Model of a Simple Electric Drive System

A typical problem of control engineering is the design of controllers for electric drive systems. While in general this task can be very complex and comprehensive, this section aims to present a simplified, yet still informative, discussion. The electric drive considered in this example is a permanent magnet DC machine which can be modeled as illustrated in Figure 6.1.

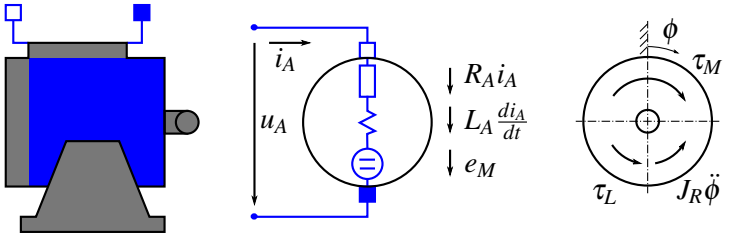


Figure 6.1.: Basic model of a permanent magnet DC machine.

R_A is the armature resistance, L_A the armature inductance, and e_M the induced armature voltage. Using Kirchhoff's voltage law gives

$$u_A = R_A i_A + L_A \frac{d\,i_A}{d\,t} + e_M. \tag{6.1}$$

The back emf, e_M, is proportional to the angular velocity while the motor torque, τ_m, is proportional to the armature current

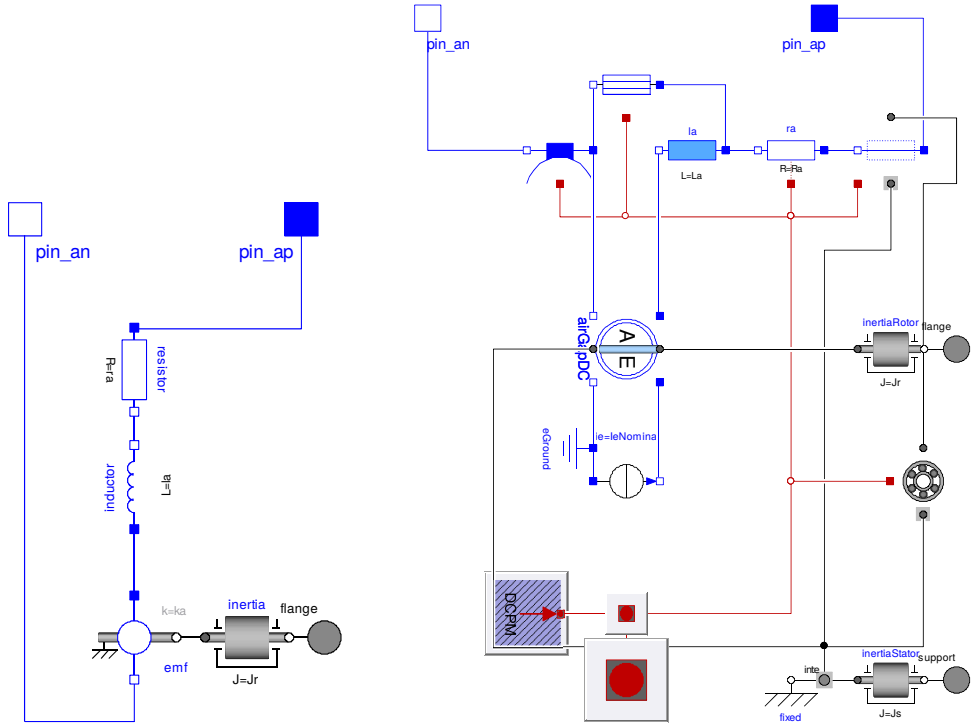
$$e_M = k_e \dot{\phi}, \qquad \tau_M = k_t i, \tag{6.2}$$

where k_e is the electromotive force constant while k_t is the motor torque constant. k_t is equal to k_e. Applying Newton's second law gives the equation

$$J_R \ddot{\phi} = \tau_m - \tau_L, \tag{6.3}$$

where J_R is the moment of inertia of the rotor and τ_L is the torque due to an external load.

The model is quickly assembled by using basic electrical components from the Modelica Standard Library (MSL). Alternatively, one may just choose a complete DC machine from the MSL and parametrize it as needed. The MSL provides much more detailed models of electrical



(a) Basic permanent magnet DC machine model.

(b) More detailed permanent magnet DC machine model from the Modelica Standard Library.

Figure 6.2.: Two models of a permanent magnet DC machine.

machines that also allow to take various loss effects into account. Figure 6.2 contrasts the component view of the basic Modelica model of the permanent magnet DC machine (Figure 6.2a) with its more detailed counterpart from the MSL (Figure 6.2b).

Rearranging Equation (6.1) gives

$$\frac{L_A}{R_A}\frac{di_A}{dt} + i_A = \frac{1}{R_A}(u_A - e_M) \tag{6.4}$$

which is a first order lag element (PT_1 element) with the input $(u_A - e_M)$, the (electrical) time constant $\frac{L_A}{R_A}$ and the steady-state gain $\frac{1}{R_A}$. Hence, the raise of the armature current is delayed due to the armature inductance.

Combining (6.1)–(6.3) allows to relate angular velocity $\omega = \dot{\phi}$ and input voltage resulting in

$$u_A = R_A \frac{J_R \dot{\omega}}{k_t} + L_A \frac{J_L \ddot{\omega}}{k_t} + k_e \omega. \tag{6.5}$$

This equation can be transformed into the Laplace domain (using zero initial conditions) and can be rearranged to yield the transfer function

$$\tilde{G}_M(s) = \frac{\omega(s)}{u_A(s)} = \frac{k_t}{(L_A s + R_A) J_R s + k_t k_e}. \tag{6.6}$$

Figure 6.3 uses the motor model displayed in Figure 6.2a as component `dcpmMotor` in a simulation experiment setup. A voltage step function is applied to the motor which is connected

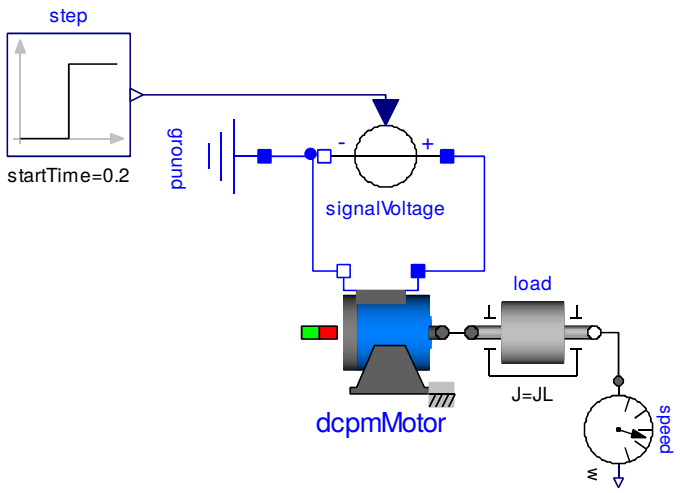


Figure 6.3.: Voltage (step function) applied to a permanent magnet DC machine which is driving an external load.

by a mechanical linkage to an external load inertia J_L.

The simulation is performed using realistic parameters of a motor with a power dissipation of 78 W (see Table 6.1).

Table 6.1.: Permanent magnet DC machine parameters (source: manufacturer's data sheet for the Pittman 9232S003 motor).

J_R	$= 1.9 \times 10^{-6}\,\mathrm{kg\,m^2}$	Rotor's moment of inertia
R_A	$= 7.38\,\Omega$	Armature resistance
L_A	$= 4.64 \times 10^{-3}\,\mathrm{H}$	Armature inductance
k_e	$= 3.11 \times 10^{-2}\,\mathrm{V\,s}$	Electromagnetic force constant
k_t	$= 3.11 \times 10^{-2}\,\mathrm{N\,m/A}$	Motor torque constant
$U_{A_{nom}}$	$= 24\,\mathrm{V}$	Nominal armature voltage

The model is simulated with a voltage step of 24 V at $t = 0\,\mathrm{s}$ and the load inertia J_L set to zero. Figure 6.4 shows the step response of the motor.

6.2. Control Design with Reduced-Order Model

The shape of the plot suggests that the high-order dynamics of the model can be neglected in many applications and that the dynamic behavior may be approximated by a first-order transfer function (PT_1 element) with appropriate parametrization. Suitable parameters for the PT_1 element can be identified from the plotting results in a graphical manner.

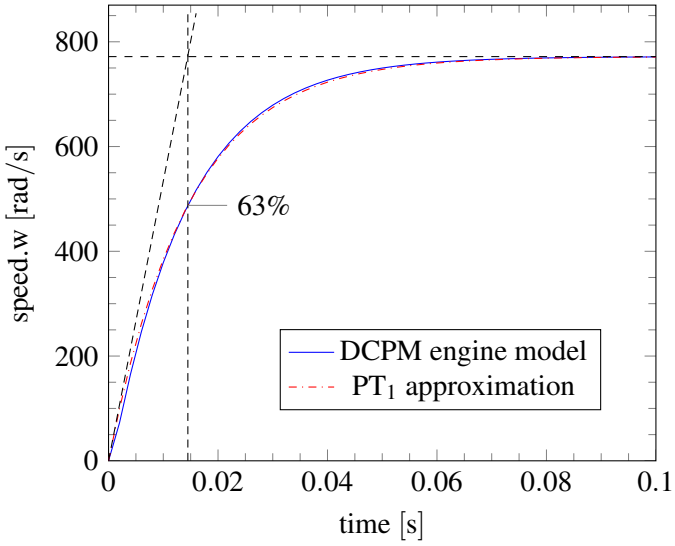


Figure 6.4.: Step response of the DCPM engine model without load ($J_L = 0$) to a 24 V step input. The plot suggests that the dynamics can be approximated by a first order transfer function (PT_1 element). The time constant τ_M of the PT_1 element can be determined by the standard approach of reading of the time at the point where the response value reaches about 63% of the steady-state (final) value. The step response of the corresponding PT_1 approximation shows hardly any deviation to the DCPM engine model's response.

The electrical time constant of the motor (Equation 6.4) is very small compared to the impact of the mechanical inertia. So, another way to yield a reduced order model is to neglect the electrical time constant by replacing the PT_1 element in Equation 6.4 by a proportional element (by setting $L_A = 0$). Hence, transfer function $\tilde{G}_M(s)$ (Equation (6.6)) is reduced to

$$G_M(s) = \frac{k_t}{R_A J_R s + k_t k_e} = \frac{\overbrace{\frac{1}{k_e}}^{k_M}}{\underbrace{\frac{R_A J_R}{k_t k_e}}_{\tau_M} s + 1}. \tag{6.7}$$

The system dynamics are therefore described by a PT_1 element. It suggests itself to compensate that dynamics by choosing a suitable control transfer function, e.g.,

$$G_C(s) = \frac{u_A(s)}{e_\omega(s)} = k_C \cdot \frac{\tau_C s + 1}{\tau_C s} = k_C \left(1 + \frac{1}{\tau_C s}\right). \tag{6.8}$$

Equation (6.8) describes a PI controller with *proportional gain* k_C and *integration time* τ_C. The block diagram in Figure 6.5 depicts the closed-loop configuration of the compensator and the approximated motor dynamics.

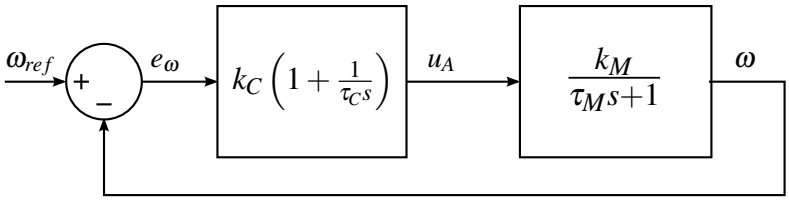


Figure 6.5.: Closed-loop configuration of the compensator and the approximated motor dynamics. Taking $\tau_C = \tau_M$ allows to compensate the effect of time constant τ_M on the system dynamics.

Taking $\tau_C = \tau_M$ the open-loop transfer function becomes

$$G_O(s) = \frac{\omega(s)}{e_\omega(s)} = G_C(s)G_M(s) = \frac{k_C k_M}{\tau_M s} \tag{6.9}$$

and the closed-loop transfer function follows to be

$$G_{CL}(s) = \frac{\omega(s)}{\omega_{ref}(s)} = \frac{G_O(s)}{1+G_O(s)} = \frac{1}{\frac{\tau_M}{k_C k_M}s+1}. \tag{6.10}$$

Hence, the closed-loop transfer function is again a PT_1 element with time constant $\frac{\tau_M}{k_C k_M}$. By increasing the value of k_C it is possible to adjust the time constant to an arbitrarily small value. However, physical constraints of the system impose limitations on sensible values for k_C.

6.3. Digital Controller Implementation

6.3.1. "Textbook" Implementation

So far, the closed-loop system is regulated by a continuous-time controller given as transfer function $G_C(s)$ (Equation (6.8)). As a next step that continuous-time controller shall be approximated by an algorithm that is suitable to run on a digital control unit. A typical approach is to use the backward-difference approximation method[1] for transforming continuous-time transfer functions into a system of recurrence equations. Using backward-differences the controller from Equation (6.8) can be approximated by

$$x_i = x_{i-1} + \frac{h}{\tau_C}u_i \tag{6.11a}$$

$$y_i = k_C(u_i + x_i) \tag{6.11b}$$

where h is the sampling period, y is the system output (corresponding to u_A in (6.8)), u the system input (corresponding to e_ω in (6.8)) and the indices i denote discrete points in time ($x_i = x(h \cdot i) = x(t)$).

[1] The backward-difference method corresponds to the transformation rule $s \to \frac{z-1}{h \cdot z}$, when mapping '$s$'-functions to (discrete-time) 'z'-functions (where h is the sampling period).

A Modelica model of such a simple PI controller is used in Figure 5.15, page 82 as illustrative example for the translation to a synchronous data-flow kernel. However, practical implementations of PI(D) controllers are more elaborate than this.

Typical aspects that are considered in practical controllers may include [41, 117, 85]

- limitation of control signals (integrator wind-up),
- interaction within control loops (information about limitations need to be communicated between inner and outer control loops),
- bumpless mode changes (due to operational reasons control systems often have to support different modes and it is essential that switching transients are avoided if mode changes occur),
- bumpless parameter changes (online tuning of parameters may effect the output even if the current error is zero; this can be sometimes avoided by a simultaneous change of the system's state variables),
- signal quality information and error handling, etc.

6.3.2. Practical PID Controller Implementation

An example for a practical PID controller implementation in the C language is given by Åström and Wittenmark in [117, Listing 8.1]. Based on the principles described by Åström and Wittenmark a discrete-time Modelica implementation is briefly presented. The implementation is adapted so that it matches good modeling practice and naming conventions of the Modelica Standard Library. Particularly, the interface matches closely to that of the continuous-time PID controller `Modelica.Blocks.Continuous.LimPID`, which already incorporates practically relevant features like *limited output*, *anti-windup compensation* and *setpoint weighting*. Hence, the continuous-time `LimPid` controller can be conveniently replaced by the presented discrete-time variant in order to analyse discretization effects, or as a base model for code generation.

The basis for the controller is the continuous-time transfer function

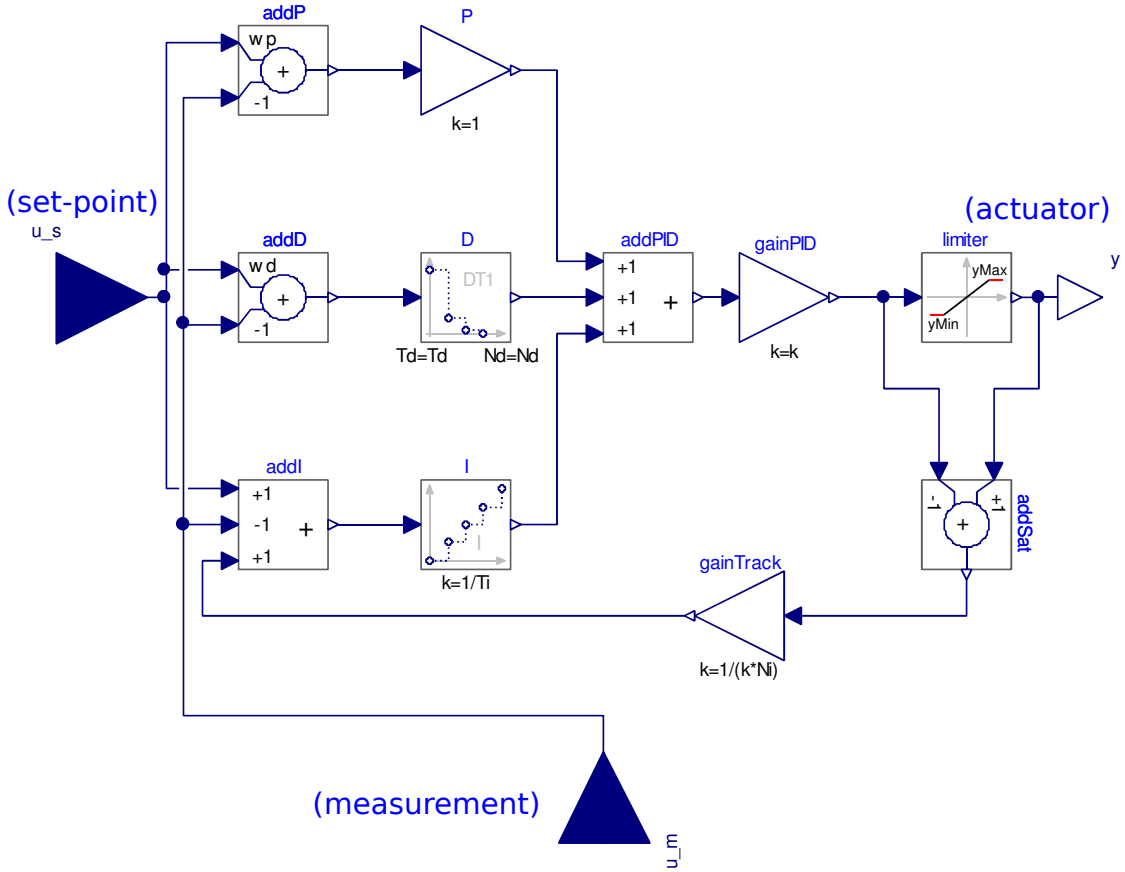
$$y(s) = k\left(w_p u_s(s) - u_m(s) + \frac{1}{sT_i}\left(u_s(s) - u_m(s)\right) + \frac{sT_d}{1 + sT_d/N_d}\left(w_d u_s(s) - u_m(s)\right)\right) \tag{6.12}$$

where u_s and u_m represent the set-point and the measurement input signal and y the controller output signal. The PID transfer function is given in an additive form so that the "P", "I", and "D" part is readily recognizable. T_i and T_d are the integral- and the derivative time constant. The "D" part is approximated by $sT_d \approx \frac{sT_d}{1+sT_d/N_d}$, where N_d limits the gain at high frequencies (typically: $3 \leq N_d \leq 20$). Set-point weighting is provided by parameters w_p and w_d and allows to weight the set-point in the proportional and derivative part independently from the measurement.

Following the example in [117, Listing 8.1] the discretization is performed by using *forward differences*[2] for the integral term and *backward differences*[3] for the derivative term. Figure 6.6 shows the resulting digital PID-controller model. Its structure (Figure 6.6a) is similar to the

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

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



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

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

(b) Parameters of the the digital PID controller.

Figure 6.6.: Digital PID controller with limited output, anti-windup compensation and set-point weighting.

continuous-time version provided in the Modelica Standard Library, the parameters (except for sampling period "`h`") are the same as in the continuous-time version so that parameters found for the continuous-time PID-controller can be directly reused in the digital version (Figure 6.6b).

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

Note that setting parameter `Td = 0` will give a controller with *PI controller characteristics*. Hence, the controller depicted in Figure 6.6a can be used as digital implementation of the compensator depicted in Figure 6.5. However, for implementation efficiency reasons it is desirable to use a dedicated PI control structure and remove all derivative related elements.

6.3.3. Graphical Representation

In Section 3.3 requirements for the graphical representation have been proposed with the motivation to allow *code reviews being done mostly on the graphical representation level*. The requirements have been picked up in Section 5.4.2 to devise concrete rules for the graphical representation of Modelica models. In the following, the graphical representation of the PID controller in Figure 6.6 shall be regarded in this context.

A graphical code review for the model requires that it is composed of components stemming from a "Library of Approved Classes with Graphical Representation" (Rule 11). Hence, either the utilized components are part of such a library, or it is required to further analyse the components by reviewing their underlying (textual) code. Furthermore, Rule 11 requires that a visual clue is present that allows to identify a class/block as being part of an approved library. Since such an approved library doesn't exist yet, there are also no corresponding visual clues in the model (i.e., no compliance to Rule 11).

Figure 6.6 is a block diagram with synchronous data-flow semantics and exclusive causal connectors — this complies to Rules 12 and 13. In Modelica blue connectors (and corresponding connection lines) are conventionally used for "**`Real`**" variables, i.e., floating-point variables with double precision (by using components from an approved library it can be ensured that one can rely on such graphical conventions).

Furthermore, the model has no "hidden" additional equations on the text layer (compliant to Rule 15). This can be quickly checked by a reviewer, but it would also lend itself for a development tool to provide an appropriate indication based on a statical code check for the model under review.

In summery, the model in Figure 6.6 follows most of the rules established in Section 5.4.2 (with the exception that its utilized components are not part of an approved library, since such a library does not exist yet). Hence, the model gives an idea of how high-level application development (developer role), as well as the model reviewing (reviewer role), could be done entirely at the graphical level.

6.4. Code Generation and SIL Validation

Figure 6.7 shows a Software-in-the-Loop (SIL) configuration in which the PID controller from Figure 6.6 was translated to C code (using the tool chain described in Section 5.9.5) and imported back into Modelica by using Modelica's external C function interface. The generated C

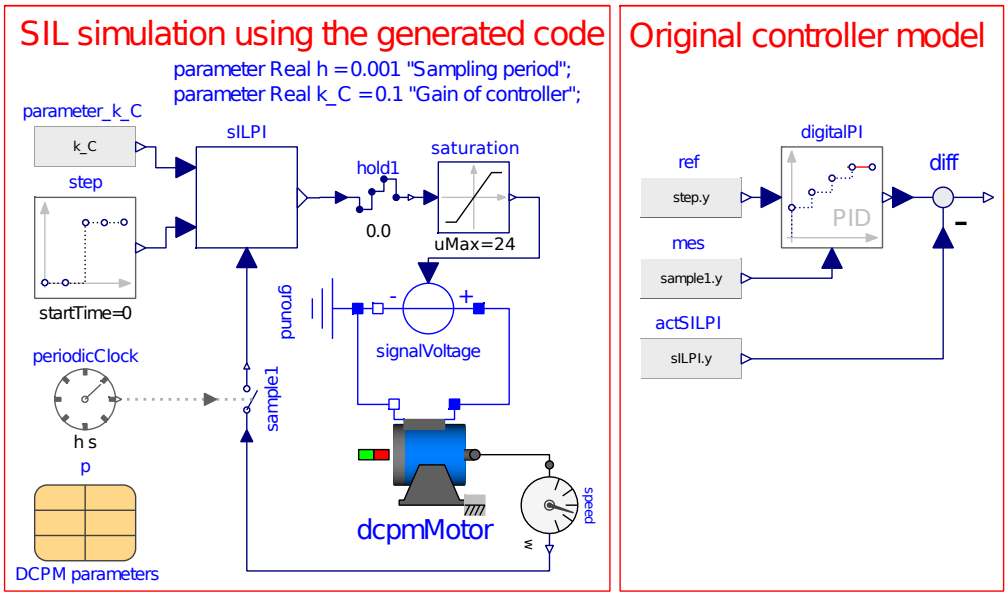
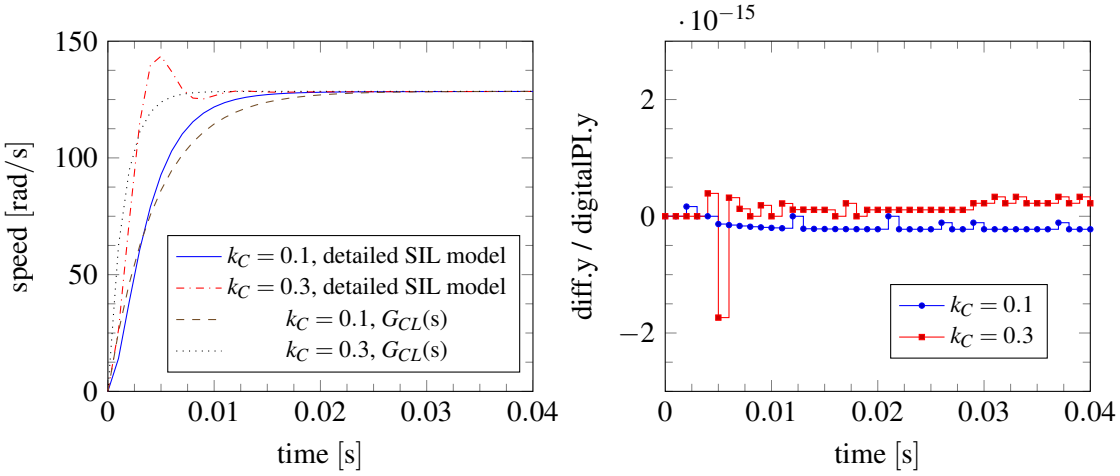


Figure 6.7.: PI(D) controller and plant in a SIL configuration. The generated code is interfaced to the model by using a "wrapper" block ("`sILPI`"). For direct comparison, the original digital PID model (see Figure 6.6) is included as component "`digitalPI`".

code is encapsulated in the "wrapper" block "`sILPI`". The digital PID block from Figure 6.6 is shown on the right side. It is fed with the same inputs as the "`sILPI`" block and is included in the model to allow a direct comparison of "model" vs "software". The PID block is configured as PI controller (`Td = 0`), and the time constant of the integrator block is set equal to the value derived for the compensator in Figure 6.5 (`Ti` $= \tau_C$). The controller gain (`k` $= k_C$) is provided as an input to the "software wrapper" block (the translation in Section 5.9 maps Modelica parameters to SDFK node inputs). Besides the top-level parameters "`k_C`" (controller gain) and "`h`" (sampling period for "`periodicClock`") the model has a parameter record component "`p`" which contains the motor parameters listed in Table 6.1. Block "`saturation`" limits the actuator output to ± 24 V. Furthermore, the model uses components "`sample1`", "`hold1`" and "`periodicClock`" from the *Modelica_Synchronous* library (see Section 4.3) in order to form a hybrid model.

Figure 6.8 compares the step response of the SIL configuration (Figure 6.7) and the step response of the idealized closed-loop configuration used for the compensator design (Figure 6.5, equation (6.10)) for two gain values k_C. The step response in Figure 6.8a shows clearly that the control performance cannot be improved arbitrarily by increasing the controller gain k_C. For low



(a) Step response of SIL model (with DCPM engine model and actuator saturation) and idealized closed-loop model $G_{CL}(s)$ (see equation (6.10)) for different controller gain values k_C.

(b) Relative error between the output of the "controller software" (i.e., code using the tool chain described in Section 5.9.5) and the "controller model".

Figure 6.8.: Step response of SIL model (Figure 6.7), idealized closed-loop model $G_{CL}(s)$ (equation (6.10), Figure 6.5), and SIL vs MIL output comparison.

values of k_C simulation results match the results predicted by the idealized closed-loop transfer function model $G_{CL}(s)$), see (6.10), that was used for the control law synthesis. For higher values of k_C the higher-order nature of the more detailed DCPM engine model and the effects of actuator saturation begin to deteriorate the closed-loop system performance.

Figure 6.8b shows a plot of the relative error between the output of the "controller software" encapsulated in the "wrapper" block "`sILPI.y`" and the output of the digital PID model "`digitalPI.y`". The difference between the "software" and the "model" stays within reasonable bounds (in the order of 1e-15). For $k_C = 0.3$ the actuator output is at its limits at the start and the anti-windup compensation with its associated equations is active. At $t = 0.005$ s the actuator output is close to zero and changes its sign for one tick, hence the sign change in the relative error.

7. Discussion and Conclusions

7.1. Discussion

This thesis had the goal to devise a development methodology that is capable of integrating two worlds: State-of-the-art modeling for *(safety-related) digital control functions* and state-of-the-art modeling for *multi-domain physical systems*.

Due to different communities state-of-the-art modeling languages for software functions have evolved very differently from modeling languages targeted for physical systems. However, modern products rely increasingly on the *tight integration of computation and physical processes*, where computation in networked embedded computers affects physical processes and vice versa. These kinds of systems have been termed cyber-physical systems. The development of such systems requires more integrated development approaches that reconcile state-of-the-art modeling methodology for software functions and multi-domain physical systems.

This thesis used the modeling language Modelica as a base to address the challenge of integrated software and physical modeling. Modelica's modeling capabilities for multi-domain physical models can be considered as the state-of-the-art in this field and therefore satisfy the requirements in this respect. However, despite some efforts, it is so far not practicable to use Modelica for safety-related digital control functions. Hence, the main effort of this thesis was to extend the state-of-the-art by establishing foundations that allow to use Modelica for safety-related digital control functions.

Chapter 1 and 2 motivate the research effort and recapitulate the most relevant background information.

Chapter 3 establishes a set of requirements (including their rationale) for model-based function development from the authors perspective. These requirements form the base for identifying current deficiencies in the Modelica language and to propose respective remedies.

Chapter 4 describes the synchronous language elements extension, which is a recent extension to the Modelica language that particularly addresses improved modeling of discrete-time control systems. The synchronous language elements, to which the author contributed, is the base for language extensions, restrictions and associated methods that are elaborated on in the remaining chapters. The extensions add the notion of *clock activation* of discrete-time equations. The model of computation within such partitions of discrete-time equations (*clocked partitions*) is based on the same synchronous paradigm as synchronous data-flow languages like Lustre or SCADE. The chapter contrasts this new approach to previous Modelica support for sampled-data systems and explains why the new approach is superior.

Despite that recent improvements, important requirements for model-based function development have still not been met. Identifying this deficiencies and developing remedies is the purpose of Chapter 5.

The chapter proposes language restrictions and extensions deemed suitable to enable Modelica to be used for safety-related (control) function development. The resulting language sub- and superset of Modelica is termed as *ACG-Modelica*. It is a central goal of the thesis to facilitate the development of a *qualifiable* automatic code generator (ACG) for this language set. It is important to understand that such an ACG may share little or no code with a typical Modelica compiler used for compiling simulation binaries (see Section 5.2).

The feasibility of qualifying a code generator is directly related to the complexity of the input language. Modelica is a very complex language, impeding attempts to implement a qualifiable code generator. However, it is possible to reduce language complexity significantly for *code generation models* (i.e., the discrete-time "software" part of the model for which many language elements for physical modeling are not needed). The proposed restrictions therefore represent a compromise that carefully strives to balance language expressivity (to suit the developers) and simplicity (to suit tool qualifiers). This compromise is inevitably subjective, however much effort has been spent not to make it arbitrarily. Proposed restrictions are justified and traced back to the requirements established in Chapter 3 (see Sections 5.4 and 5.8). As a result, the language complexity of ACG-Modelica is significantly reduced.

Besides language restrictions, Chapter 5 also proposes several language extensions. These extensions are motivated by perceived limitations of the current language standard. These limitations are addressed and a remediation proposal is worked out.

Section 5.5 proposes an extension to allow more fine-grained control over the data types used on an (embedded) target platform. For efficiency reasons, it is also proposed to introduce bitwise operations as a built-in language feature.

Section 5.6 adds the notion of "`atomic`" blocks in order to enable modular code generation techniques. Several advantages are identified for modular code generation: (i) it enables separate compilation, which has advantages in respect to scalability and protection of intellectual property, (ii) it improves the traceability from the model to the generated code, which is beneficial for high-integrity code generation, (iii) it allows improved code sharing, which reduces the size of code footprint, (iv) it facilitates the combination of generated code with handwritten code, and last but not least (v) it facilitates to generate code which is well-structured and adheres to good software engineering practice. The last point allows to pass a generated code artifact into a standard assurance process for handwritten code. This property is not strictly needed if the tool chain (including the ACG) has been qualified up to a suitable level within the intended application domain — however, this may not always be the case. In such cases at least some effort can be saved if code can be generated (instead of writing it by hand), even if the code must subsequently go through a standard assurance process (e.g., by performing code inspections on the generated code).

Section 5.7 explains why more manual control over the scheduling of blocks is desirable for some use cases. Besides a language extension proposal to enable manual block scheduling the section also proposes an alternative solution based on a modeling pattern. The modeling pattern based solution is not as powerful, but should be adequate for many use cases. Its important advantage is that it can be realized using existing language capabilities which make it the preferred solution for ACG-Modelica.

Section 5.8 discusses trade-offs between *acausal* and *causal* data-flow semantics. It demonstrates that Modelica's acausal data-flow can be problematic in safety-related applications. While

adequate safeguarding mechanisms should be able to mitigate perceived problems the simple solution proposed for ACG-Modelica is to use causal data-flow semantics exclusively. Note that any valid ACG-Modelica program based on causal data-flow is also a valid (acausal data-flow) Modelica program. The inverse is not true. Besides, causal data-flow is the standard semantics used in control engineering so that this restriction should be well acceptable for a control oriented subset of the Modelica language.

In the context of this thesis, Section 5.9 is of particular importance: it provides a formalized argument that it is possible to develop a qualifiable automatic code generator for an appropriate control-oriented subset of Modelica. This argument is obtained by (i) introducing a synchronous data-flow kernel language (SDFK) considered suitable to allow the proposition that a qualifiable automatic code generation can be implemented for it, (ii) definition of a representative Modelica language kernel (mACG-Modelica), and (iii) definition of a set of translation equations from mACG-Modelica to SDFK (i.e., *translational semantics* of mACG-Modelica in terms of SDFK). In addition to the formal translation equations the section briefly reports on a concrete prototypical translator that was implemented as part of the research efforts.

The translation provides two novelties: (i) a theoretical justification for the feasibility of developing a qualifiable automatic code-generator for Modelica and (ii) a base that can be used to develop a gateway from Modelica to existing qualifiable code generators based on the synchronous data-flow paradigm, e.g., SCADE/KCG.

Chapter 6 provides an elaborated example of using Modelica as a framework for model-based function development. The example targets a typical control engineering task: design of a digital controller for an electric drive system (servo control system). It illustrates that a typical development process requires models of varying fidelity that suffice for different purposes, e.g., reduced-order continuous-time plant models in order to apply established control law synthesis methods, discrete-time controller models that approximate continuous-time control design to enable digital controller implementations, high-fidelity hybrid models in order to evaluate and fine-tune system performance.

Section 6.3 seizes aspects of practical controller implementations that are often omitted in more theoretical oriented control design "textbooks" and presents a practical-oriented PID controller model for automatic code generation. The controller is modeled using the ACG-Modelica language subset and the prototypical translator for that subset is used to generate code. The generated code is interfaced to the simulation tool for software-in-the-loop simulation. Results of software- and model-in-the-loop simulation are compared against each other. In summary, the servo control system example illustrates how important aspects of model-based development can be handled in a compelling integrated and seamless fashion using the proposed Modelica framework.

7.2. Future Work

This thesis proposes a path to a qualifiable automatic code generator for Modelica and provides theoretical as well as practical evidence for the feasibility of such an undertaking. An obvious point for the future is to use these results as a base and develop and qualify an industry-strength code generator — an undertaking that is expected to require a (semi-) commercial effort, since

it can hardly be achieved in a purely academic setting.

Naturally, there are many options and directions for extending the language subset and supporting additional features. Any such extensions will need to renegotiate the balance between tool qualification efforts and language expressiveness. Particularly, the support of inheritance and built-in language support for state machines modeling as discussed in Section 5.10 are candidates for future extensions.

In the context of modern Modelica compiler technology, further more visionary extensions suggest themselves. Section 2.4 reported about high-level prototypical work using Modelica technology for advanced model-based controller designs. These works leverage the powerful symbolic equation manipulation capabilities available in some Modelica tools to synthesize nonlinear inverse models for control or enable nonlinear model predictive control approaches. It is an interesting research question how such capabilities can be reconciled with safety-related development objectives. A promising approach could be to present critical symbolic transformations by the development tool in a human-checkable form that allows to validate them manually. Novel methods for debugging equation-based languages already keep track of symbolic transformations during the translation process [86]. Whether this methods could also be leveraged to validate critical transformations occurring during code generation for advanced model-based controllers is a topic for future research.

7.3. Conclusion

The motivation for this thesis effort was the lack of a development methodology that is capable of integrating two worlds: state-of-the-art modeling for multi-domain physical systems and state-of-the-art modeling for (safety-related) digital control functions. The thesis addressed the gap that so far prevented using Modelica for the later, i.e., *adequate language expressiveness for discrete-time controller functions* and *production quality automatic code generation*. Both aspects have been addressed by restricting and extending Modelica to a language set termed *ACG-Modelica* and investigating a formal translation of this language to an established *synchronous data-flow kernel* language for which production quality code-generation techniques are known and implemented in industrial code generators for safety-relevant embedded software. The semantic distance between ACG-Modelica and the synchronous data-flow kernel has shown to be wider than initially expected which is reflected in the complexity of the translation equations that have been deduced. The concluding servo system example illustrates the interplay between physical modeling and controller modeling within the envisioned Modelica based function development framework.

A. Introduction to Modelica

Modelica can be called an *equation-based object-oriented* (EOO) language. That term, coined by Broman [22], 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.

A.1. Object-orientation

The term object-oriented is not to be understood in exactly the same way like in common object-oriented programming (OOP) languages like Smalltalk, C++ or Java. In OOP languages objects are instances of classes that comprise *data + methods*. Objects *behave* by methods that are called from other objects or by sending or receiving messages.

In contrast to method calls or message passing, behavior in EOO languages is described by equations. Therefore EOO language objects are instances of classes that basically comprise *data + equations*.

Similar to common OOP languages Modelica also supports mechanisms like inheritance and subtyping-polymorphism. Note that in Modelica *objects* are termed *components*, while *classes* are often referred to as *models*.

Modelica provides *annotations* that can be used to define a graphical component representation. Professional Modelica development environments typically provide a graphical diagram view in which models can be composed from such annotated components in a drag-and-drop style. Figure A.1 shows diagram views of a complete vehicle system model assembled from Modelica components spanning several domains such as powertrain, thermo, and multi-body dynamics.

A.2. Equation-based Behavior Modeling

Equations have been used as adequate notation for capturing system dynamic behavior over many centuries. The key characteristic of EOO languages is to directly use declarative mathematical equations for describing system dynamic behavior.

Modelica currently provides support for models described by

- *ordinary differential equations* (ODE),
- *algebraic equations*,

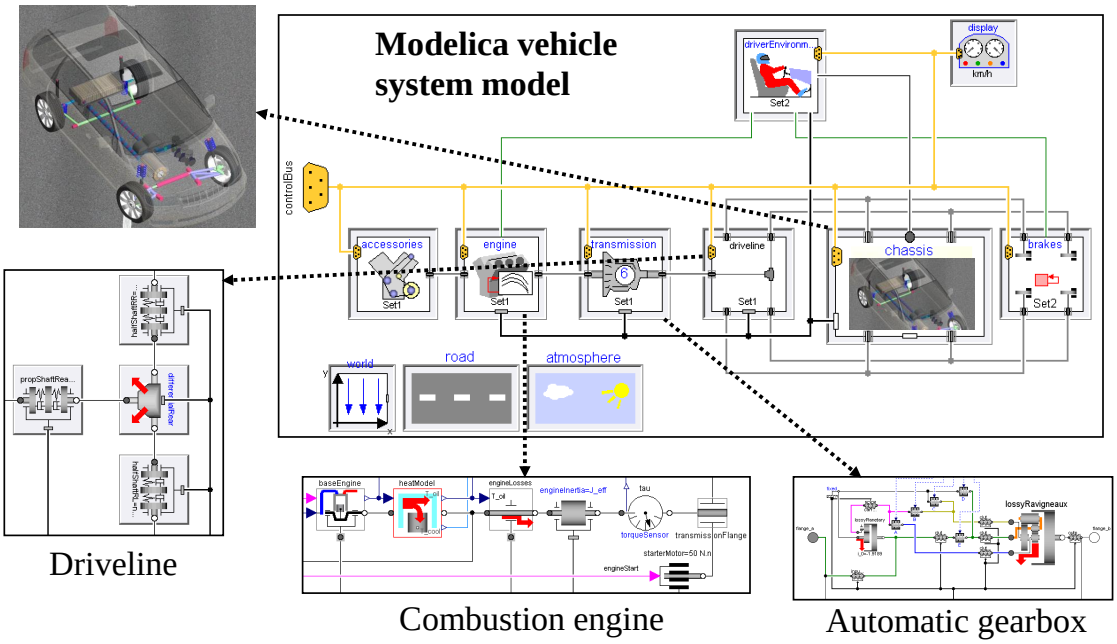


Figure A.1.: Diagram view of a complete vehicle system model spanning domains such as powertrain, thermo, and multi-body dynamics. Courtesy DLR-SR.

- *event handling* and *recurrence relations* (sampled control).

Currently there is no direct support for describing *partial differential equations* (PDEs) although it is possible to work with suitable discretized PDE models (e.g., by importing the results of a *finite element method* (FEM) program, or by manual discretization of simple PDEs).

The crucial point is that equations are acausal (or non-causal), which means that the causality of how the equations have to be solved need not be specified at modeling time. As an example consider Ohm's law, given as (acausal) equation

$$i = u/R$$

where i is the current, v is the potential difference across the conductor, and R is the resistance. Depending which variable is unknown the equation can be rearranged to following (causal) assignments:

$$R := u/i$$
$$u := Ri$$
$$i := u/R.$$

In conventional programming, equations need to be rearranged into causal assignment statements. This is a severe drawback for reusability of physical models since such models are then only valid for a specific combination of known and unknown variables.

There are two different abstraction levels where acausality is present in Modelica:

1. at the *equations level*, and
2. at the *component connection level*.

A.2.1. Equation Level

Consider the spring-mass-damper system depicted in Figure A.2.

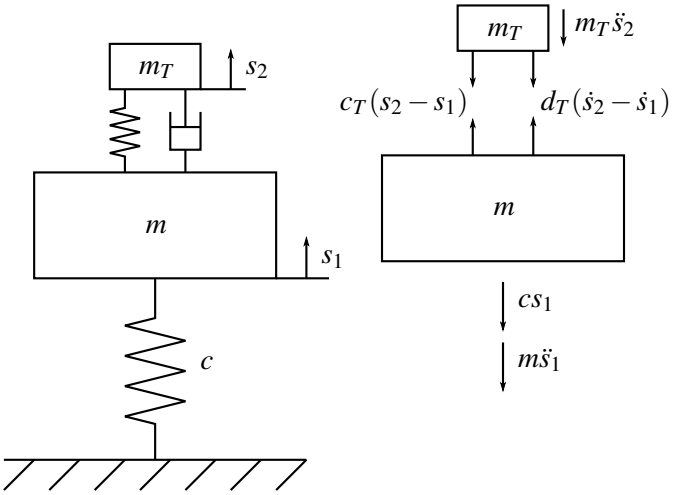


Figure A.2.: Spring-mass-damper system.

On the right-hand side the system is in its position of rest (spring forces equals weight of the masses). On the left-hand side the system is depicted slightly displaced with the corresponding cutting and inertial forces. Using d'Alembert's principle results in the following system of equations:

$$m\ddot{s}_1 + cs_1 = c_T(s_2 - s_1) + d_T(\dot{s}_2 - \dot{s}_1) \tag{A.1}$$

$$m_T\ddot{s}_2 + c_T(s_2 - s_1) + d_T(\dot{s}_2 - \dot{s}_1) = 0. \tag{A.2}$$

The two ODEs are coupled. If the natural frequency of the vibrating small mass m_T can be tuned (by choosing an appropriate combination of mass, stiffness and damping) to equal the natural frequency of the vibrating primary mass m an initial vibration of m can be damped significantly. This is the principle of a *tuned mass damper* (TMD).

Using the following numerical quantities the spring-mass-damper system shows tuned mass damper behavior.

m	$= 100\,\mathrm{kg}$	Primary mass
c	$= 500\,\mathrm{N/m}$	Stiffness
m_T	$= 5\,\mathrm{kg}$	TMD mass
c_T	$= 25\,\mathrm{N/m}$	TMD stiffness
d_T	$= 10\,\mathrm{Ns/m}$	TMD damping coefficient
s_{10}	$= 0.1\,\mathrm{m}$	Initial displacement of primary mass

The equations can almost directly be copied into a Modelica model.

```
1  model SpringMassDamper "Spring-mass-damper system"
2    parameter Real m(unit="kg") = 100 "Primary mass";
3    parameter Real c(unit="N/m") = 500 "Stiffness";
4    parameter Real m_T(unit="kg") = 5 "TMD mass";
5    parameter Real c_T(unit="N/m") = 25 "TMD stiffness";
6    parameter Real d_T(unit="N.s/m") = 10 "TMD damping coefficient";
7    parameter Real s_1_0(unit="m") = 0.1 "Initial displacement
8                                          of primary mass";
9  protected
10   Real s_1(unit="m", start=s_1_0, fixed=true) "Displacement of
11                                                primary mass";
12   Real v_1(unit="m/s", start=0, fixed=true) "Velocity of
13                                             primary mass";
14   Real s_2(unit="m", start=0, fixed=true) "Displacement of TMD mass";
15   Real v_2(unit="m/s", start=0, fixed=true) "Velocity of TMD mass";
16 equation
17   v_1 = der(s_1);
18   m*der(v_1) + c*s_1 = c_T*(s_2-s_1) + d_T*(v_2-v_1);
19   v_2 = der(s_2);
20   m_T*der(v_2) + c_T*(s_2-s_1) + d_T*(v_2-v_1) = 0;
21 end SpringMassDamper;
```

Figure A.3 shows the simulation result of the system with tuned mass damper device and compares it against the result which is obtained if the spring-damper element between the primary mass m and the TMD mass m_T is replaced by a rigid link. The damping effect of the tuned mass damper device is clearly visible.

Equation sections

Below the heading **`equation`** the governing equations are declared. The operator **`der`**`(x)` is the differentiation operator $\frac{dx}{dt}$. If an equation system contains higher order derivatives it needs to be transformed into a system with first-order derivatives by introducing appropriate substitutions. E.g., using the substitutions $v_1 = \dot{s}_1$ and $v_2 = \dot{s}_2$, Equations (A.1)–(A.2) can be rewritten to

$$v_1 = \dot{s}_1 \tag{A.3}$$

$$m\dot{v}_1 + cs_1 = c_T(s_2 - s_1) + d_T(v_2 - v_1) \tag{A.4}$$

$$v_2 = \dot{s}_2 \tag{A.5}$$

$$m_T\dot{v}_2 + c_T(s_2 - s_1) + d_T(v_2 - v_1) = 0, \tag{A.6}$$

which directly translates into the corresponding Modelica code (lines 17–20).

Note that it is not necessary to rearrange the equations into the explicit first order ODE form

$$\dot{x}(t) = f(t,x), \quad x \in \mathbb{R}^n \tag{A.7}$$

typically required by numerical ODE solvers. A Modelica tool will automatically sort equations and perform symbolic manipulations or rearrangements that allow to solve the equations

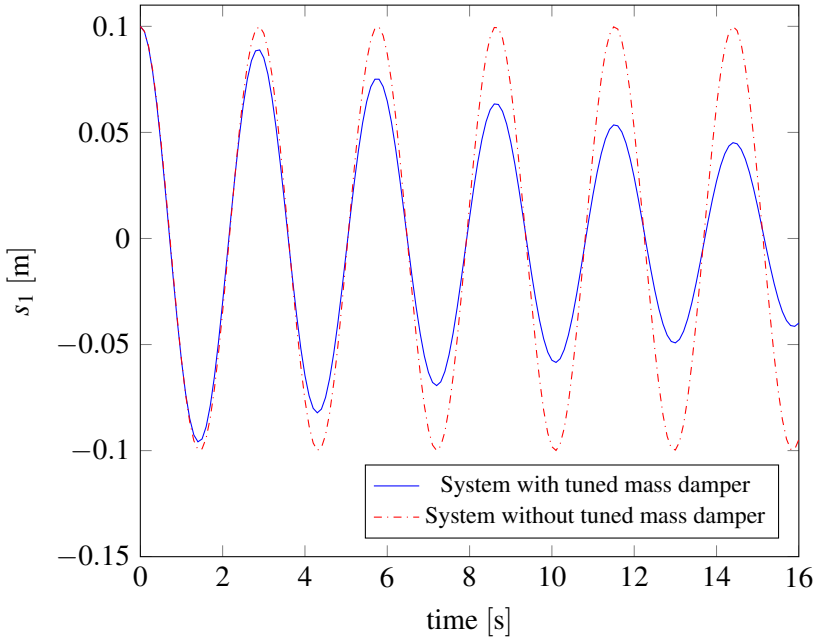


Figure A.3.: Simulation results of the mechanical system with tuned mass damper device from Figure A.2 compared against the system that results if the spring-damper element between the primary mass m and the TMD mass m_T is replaced by a rigid link.

numerically in an efficient manner[1] (see Section 5.2.2 for more details).

Remaining sections

Essentially everything in Modelica is a class. This includes predefined classes, e.g., **`Real`**, **`Integer`**, or **`Boolean`** as well as large *packages* such as the Modelica Standard Library (MSL). The first line in the example code starts the declaration of the model (**`model`** is a specialized kind of a class) `SpringMassDamper`. A short description string may optionally follow any class or instance/component[2] declaration.

In lines 2–7 the numerical quantities are defined as parameters of the model `SpringMassDamper`. Modelica supports the assignment of physical units. If units are assigned to quantities a Modelica tool can exploit that for improved dialog presentations, plotting purposes and for checking the unit compatibility of equations.

Lines 10–15 instantiates continuous-time variables used within the equations below. The heading **`protected`** restricts their visibility outside their enclosing class. The default visibility in Modelica is **`public`** (therefore, the parameters in lines 2–7 have public visibility). The attributes `start` and `fixed` define the initial condition for a variable. If the attribute `fixed=`**`true`** is set, the initial condition is required to hold during the initialization analysis of the Modelica tool. Otherwise the value of `start` is treated as a *guess* value and the tool may choose another

[1] It is a matter of quality of implementation how well a tool can do this.

[2] Note that in Modelica [73] the term *component* is used synonymously to *instance* or *object*.

initial value during initialization in order to start simulation with a *consistent set of initial values* for all model variables.

A.2.2. Component Connection Level

The natural mapping from mathematical equations to Modelica code as demonstrated above is convenient, however, it still requires to manually derive the governing equations. Reuse and modeling convenience is increased by capturing the behavior of fundamental technical components within libraries and providing a mechanism to connect the components. Figure A.4 shows the spring-mass-damper example built by using components available in the Modelica Standard Library. On the left hand side the textual representation is given, the right hand side shows the graphical representation created by drag-and-drop modeling within the Modelica tool Dymola. The graphical representation is stored in **`annotation`** elements within the textual Modelica code. For brevity and clarity this annotations are omitted in the displayed example code.

```
1  model SpringMassDamper2
2    import Modelica.Mechanics.
3      Translational.Components.*;
4    Fixed fixed;
5    Spring c(c=500);
6    Mass m(m=100, s(start=0.1, fixed=true));
7    Spring c_T(c=25);
8    Damper d_T(d=10);
9    Mass m_T(m=5);
10 equation
11   connect(fixed.flange, c.flange_a);
12   connect(c.flange_b, m.flange_a);
13   connect(m.flange_b, c_T.flange_a);
14   connect(m.flange_b, d_T.flange_a);
15   connect(d_T.flange_b, m_T.flange_a);
16   connect(c_T.flange_b, m_T.flange_a);
17 end SpringMassDamper2;
```

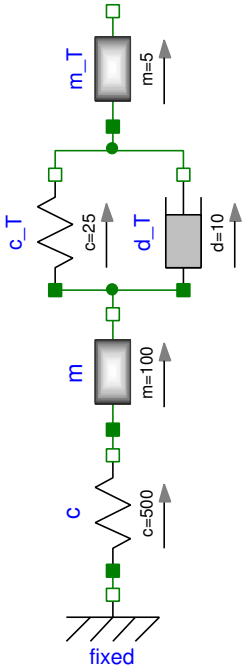


Figure A.4.: Spring-mass-damper system build using components from the Modelica Standard Library (MSL). *Modifiers* are used in the component declarations (line 5–9) to set parameter values and the initial displacement of the primary mass.

Components have *"connectors"* (also called *"ports"*) that constitute the interface of the component. Compatible ports can be coupled by *"connect"*-equations. When working at the graphical diagram level this is realized by drawing a connection line between respective ports.

Physical ports, like electrical pins or mechanical flanges, are modeled by an adequate choice of *potential* and *flow* variables (also called *across* and *through* variables) that are available in the port.

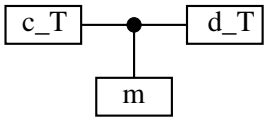
The connection of two or more components implicitly introduces a set of coupling equations for the involved potential and flow variables. All corresponding potential variables within connected ports are set equal, all the corresponding flow variables introduce an equation which sets the sum of the flow variables equal to zero. Therefore, the connections in the example do not denote data flow causality, they denote (acausal) physical constraints.

Connectors (Ports)

Connectors are a specialized kind of classes that may only have public sections (i.e., no equation or protected sections are allowed). They are enhanced to allow the use of **`connect(..)`** between their class instances. The MSL connector for translational mechanics (used in the example in Figure A.4) uses *distance* as potential variable and *force* as flow variable. A corresponding connector definition in Modelica is presented below.

```
1 connector Flange
2   Real s(unit="m") "Absolute position of flange";
3   flow Real f(unit="N") "Cut force directed into flange";
4 end Flange;
```

Consider the connection point of `m`, `c_T` and `d_T` from the spring-mass-damper example of Figure A.4.



The coupling leads to the following connect equations:

$$\begin{aligned} \text{m.flange\_b.s} &= \text{c\_T.flange\_a.s} \\ \text{m.flange\_b.s} &= \text{d\_T.flange\_a.s} \\ 0 &= \text{m.flange\_b.f} + \text{c\_T.flange\_a.f} + \text{d\_T.flange\_a.f} \end{aligned}$$

where the potential variable `s` ensures that relevant *boundary conditions* are fulfilled while the flow variable `f` generates the necessary *balance equation* at the connection point.

The choice of useful connector definitions is not obvious. Table A.1 provides examples for some other elementary connector definitions that are used in the MSL.

Physical Components

The physical components contain the governing equations in the context of their port interfaces. As an example consider the model of a translational spring component using the mechanical flange from above given in Listing A.1.

Line 3 and 4 declare the physical interaction ports. The equation in line 12 provides the well known relation that the force that a compressed or stretched spring exerts is proportional to its change in length. Line 9–11 provide the relation between the port variables and the auxiliary variables `s_rel` and `f` that are introduced in line 6 and 7 to improve readability. Similarly,

Listing A.1: A translational mechanics spring model.

```
model Spring "Linear 1D translational spring"
  parameter Real c(unit="N/m") "Spring constant";
  parameter Real s_rel0(unit="m") = 0 "Unstretched spring length";
  Flange flange_a;
  Flange flange_b;
  Real s_rel(unit="m", start=0) "Relative distance";
  Real f(unit="N") "Force between flanges";
equation
  s_rel = flange_b.s - flange_a.s;
  flange_b.f = f;
  flange_a.f = -f;
  f = c*(s_rel - s_rel0);
end Spring;
```

Listing A.2: A translational mechanics mass model.

```
model Mass "Sliding mass with inertia"
  parameter Real m(unit="kg") "Mass of the sliding mass";
  parameter Real L(unit="m") = 0
    "Length of component (= flange_b.s - flange_a.s)";
  Flange flange_a "Left flange of component";
  Flange flange_b "Right flange of component";
  Real s(unit="m") "Absolute position of center of component";
  Real v(unit="m/s") "Absolute velocity of component";
  Real a(unit="m/s2") "Absolute acceleration of component";
equation
  flange_a.s = s - L/2;
  flange_b.s = s + L/2;
  v = der(s);
  a = der(v);
  m*a = flange_a.f + flange_b.f;
end Mass;
```

Table A.1.: Excerpt of connector variables used in the MSL.

Physical Domain	Potential Variable	Flow Variable
Translational Mechanics	Distance	Force
Rotational Mechanics	Angle	Torque
Electrical Analog	Electrical Potential	Electrical Current
Thermal	Temperature	Heat Flow Rate
Magnetic	Magnetic Potential	Magnetic Flux

the model for a mass component using the mechanical flange interface from above is shown in Listing A.2.

Newton's second law of motion, $F = ma$, is declared in line 15. The net force acting on the mass is the sum of the forces acting on the mechanical flange interface. The absolute position of the center of the component is given in relation to the position of the flange interfaces in line 11 and 12. The acceleration is obtained by differentiating the position twice (line 13 and 14).

The other components of the example (`Damper`, and `Fixed`) can be modelled similarly.

Comparison with data-flow modeling

A straightforward data-flow (causal) model derived by solving (A.3)–(A.6) for $\dot{v}_1$ and $\dot{v}_2$ is shown in Figure A.5.

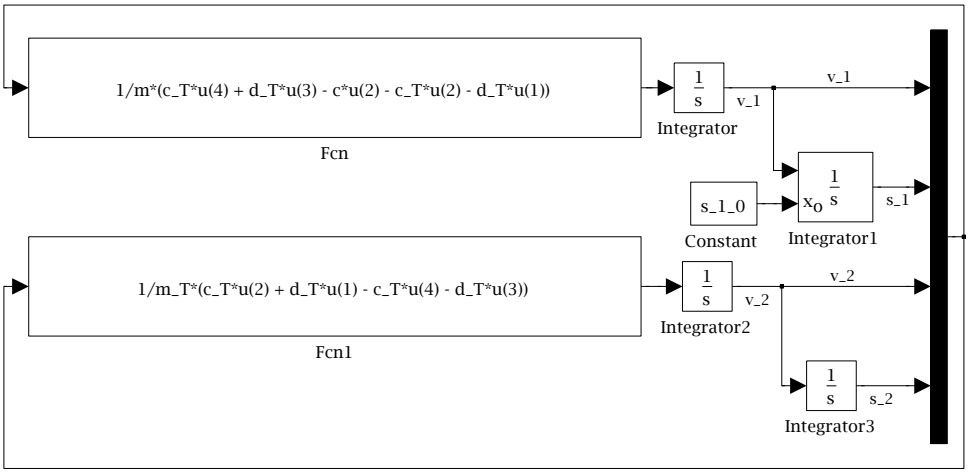


Figure A.5.: Straightforward data-flow model for the spring-mass-damper example. The black bar combines the four scalar input signals in one vector signal. Within the two function blocks (`Fcn` and `Fcn1`) the elements of the input vector signal `u` are accessed by indices (`u(1)`–`u(4)`). The output of `Fcn` and `Fcn1` is a scalar signal.

Compared to the (acausal) EOO model in Figure A.4 the topology of how the components would be connected in the physical world is lost. The diagram reveals aspects of the mathematical structure of the problem, but (without context information) the physical origin of the model can hardly be deduced. Furthermore, adding additional physical components will often require elaborate rearrangements in the data-flow diagram because of the dependence of data-flow on the causality of the underlying equation system.

A conceivable more modular model can be accomplished by utilizing hierarchical data-flow modeling as shown in Figure A.6.

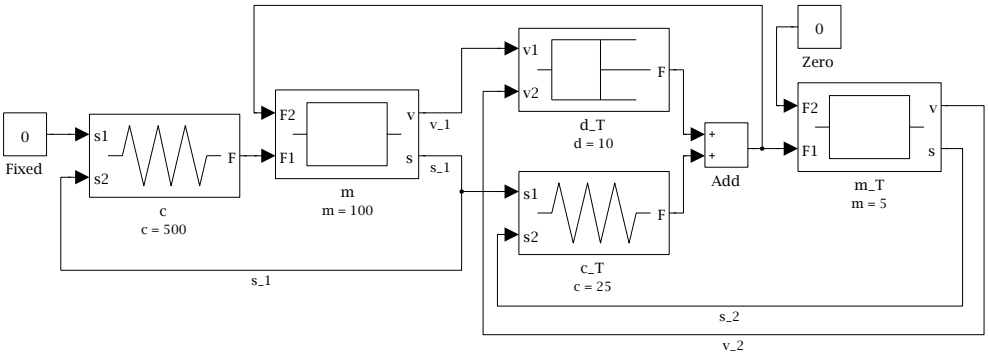


Figure A.6.: Hierarchical data-flow model for the spring-mass-damper example.

The base elements, spring, mass, and damper, are now described by their own signal-flow diagram (see Figure A.7).

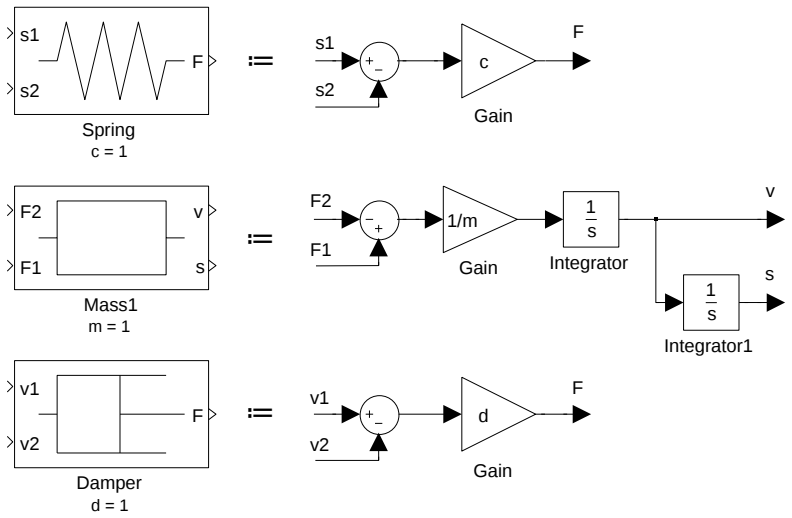


Figure A.7.: Data-flow diagram of the three base elements.

The structure of the underlying physical model is better recognizable in this improved model.

However, mainly because of the high number of required feedback loops readability and extensibility is still worse than in the EOO model displayed in Figure A.4.

In addition there persists a more subtle, but severe problem with the hierarchical data-flow modeling style depicted in Figure A.6 and A.7. Valid combinations of the modular building blocks in Figure A.7 are rather limited. For example, it is not possible to connect two mass-blocks directly without an intermediate element that provides an output force. Therefore, an additional version of the mass-block would be required to allow modeling of rigid couplings between several masses in a modular, object oriented manner. While this might not appear to be particularly disadvantageous in the depicted example context, it is a limitation which is of great relevance in practical physical modeling.

Now contrast this with EOO modeling. After removing the upper spring and damper in Figure A.4 the two masses can be directly connected as depicted in Figure A.8.

```
1  model SpringMassMass
2    import Modelica.Mechanics.
3      Translational.Components.*;
4    Fixed fixed;
5    Spring c(c=500);
6    Mass m(m=100, s(start=0.1, fixed=true));
7    Mass m_T(m=5);
8  equation
9    connect(fixed.flange, c.flange_a);
10   connect(c.flange_b, m.flange_a);
11   connect(m_T.flange_a, m.flange_b);
12 end SpringMassMass;
```

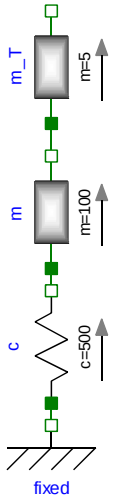


Figure A.8.: Spring-mass-mass system build using components from the Modelica Standard Library (MSL). *Modifiers* are used in the component declarations (line 5–9) to set parameter values and the initial displacement of the primary mass.

A Modelica tool will collapse the instance hierarchy and *flatten* the hierarchical composed model into one (large) system of equations (see Section 5.2.2). Listing A.3 shows the flattened equations of the spring-mass-mass example of Figure A.8 produced by the tool Dymola (stripped of some elements like variable declaration to allow for a more concise presentation).

The system of equations displayed in Listing A.3 is still rather large due to many trivial algebraic equality relations caused by the interface variables of the components and the connect equations relating them. In order to arrive at a more manageable representation the equations are transformed to the more concise form displayed in Figure A.9. This is achieved by eliminating trivial equalities using subscripts for variable distinction (format: $quantity_{object_{flange}}$), eliminating parameters with zero values, and using multiple differentiation of variables.

Formally differentiating (A.8) twice yields the equality $\ddot{s}_{m\_T} = \ddot{s}_m$ which allows to substitute

Listing A.3: Flattened system of equation for Spring-mass-mass system.

```
fixed.flange.s = fixed.s0;

// Component c
c.s_rel = c.flange_b.s-c.flange_a.s;
c.flange_b.f = c.f;
c.flange_a.f = -c.f;
c.f = c.c*(c.s_rel-c.s_rel0);

// Component m
m.flange_a.s = m.s-m.L/2;
m.flange_b.s = m.s+m.L/2;
m.v = der(m.s);
m.a = der(m.v);
m.m*m.a = m.flange_a.f+m.flange_b.f;

// Component m_T
m_T.flange_a.s = m_T.s-m_T.L/2;
m_T.flange_b.s = m_T.s+m_T.L/2;
m_T.v = der(m_T.s);
m_T.a = der(m_T.v);
m_T.m*m_T.a = m_T.flange_a.f+m_T.flange_b.f;
m_T.flange_b.f = 0.0;

// Component
c.flange_a.f+fixed.flange.f = 0.0;
fixed.flange.s = c.flange_a.s;
c.flange_b.f+m.flange_a.f = 0.0;
m.flange_a.s = c.flange_b.s;
m.flange_b.f+m_T.flange_a.f = 0.0;
m_T.flange_a.s = m.flange_b.s;
```

$$
\begin{aligned}
&\texttt{//Component c} \\
&f_{c_b} = c_c \cdot s_{c_b} \\
&\texttt{//Component m} \\
&s_m = s_{m_a} = s_{m_b} \\
&m_m \cdot \ddot{s}_m = f_{m_a} + f_{m_b} \\
&\texttt{//Component m\_T} \\
&s_{m\_T} = s_{m\_T_a} = s_{m\_T_b} \qquad \rightsquigarrow \qquad s_m = s_{m_T} \qquad \text{(A.8)} \\
&m_{m\_T} \cdot \ddot{s}_{m\_T} = f_{m\_T_a} \qquad m_m \cdot \ddot{s}_m = -c_c \cdot s_m - m_{m\_T} \cdot \ddot{s}_{m\_T} \qquad \text{(A.9)} \\
&\texttt{//Component} \\
&f_{c_b} + f_{m_a} = 0 \\
&s_{m_a} = s_{c_b} \\
&f_{m_b} + f_{m\_T_a} = 0 \\
&s_{m\_T_a} = s_{m_b}
\end{aligned}
$$

Figure A.9.: A more manageable presentation of the equations displayed in Listing A.3. Further simplification reduces the system to the two equations displayed on the right-hand side.

$\ddot{s}_{m\_T}$ by $\ddot{s}_m$ in (A.9) resulting in the ODE

$$(m_m + m_{m\_T}) \cdot \ddot{s}_m = -c_c \cdot s_m \tag{A.10}$$

which can be easily further rearranged into an explicit first order ODE form (see Equation (A.7)) for numerical integration.

The important point is that an EOO tool performs such symbolic manipulations *automatically*. Therefore compared to a (causal) data-flow modeling style an (acausal) EOO modeling style allows for improved *modularity* and *reusability* of physical models.

A.3. Inheritance and Redeclaration

For improving reusability, expressiveness, and maintenance of models Modelica supports concepts for inheritance along with redeclaration constructs. Redeclaration is a rather specific concept in Modelica that can be seen as a limited version of parametric polymorphism[3] [23].

Consider the basic models, spring, mass, and damper, used in the spring-mass-damper example of Figure A.4. They all share the property, that they contain two translational flanges. Furthermore, the spring and the damper model additionally share the property that they are massless components where the absolute value of the force on the left and the right side flange

[3]Examples for parametric polymorphism in mainstream languages are Java generics or C++ templates.

is the same. Therefore it suggests itself to introduce a common base class for the spring and damper model that captures these similarities.

```
1  model TwoFlange
2    "Superclass for model components with two flanges and no inertial effects"
3    Flange flange_a;
4    Flange flange_b;
5    Real s_rel(unit="m", start=0) "Relative distance";
6    Real f(unit="N") "Force between flanges";
7  equation
8    s_rel = flange_b.s - flange_a.s;
9    flange_b.f = f;
10   flange_a.f = -f;
11 end TwoFlange;
```

Using that superclass the class declarations for spring and damper can be shortened.

```
1  model Spring "Linear 1D translational spring"
2    extends TwoFlange;
3    parameter Real c(unit="N/m") "Spring constant";
4    parameter Real s_rel0(unit="m") = 0 "Unstretched spring length";
5  equation
6    f = c*(s_rel - s_rel0);
7  end Spring;
8
9  model Damper "Linear 1D translational damper"
10   extends TwoFlange;
11   parameter Real d(unit="N.s/m") "Damping constant";
12   Real v_rel(unit="m/s") "Relative velocity";
13 equation
14   v_rel = der(s_rel);
15   f = d*v_rel;
16 end Damper;
```

In the listing above the keyword **extends** denotes inheritance (line 2 and 10). Multiple inheritance is supported.

It is also possible to change the internal structure of a component when declaring it or when extending from it (Redeclaration). This can be achieved by using the **redeclare** language construct on components that have been explicitly declared as **replaceable**. It is required that the replacing component is *compatible*[4] to the component that shall be replaced.

Assume the existence of a combined spring-damper model "SpringDamper" (compatible to the model "Spring") and consider the following model declarations.

```
1  model A
2    replaceable Spring c(c=500)
3  end A;
4
5  model B
```

[4]Compatible in the sense of being a subtype of the type of the component to be replaced. Note that Modelica uses a *structural type system* and therefore an inheritance relation is no imperative requirement for a subtype relation.

```
6 extends A(redeclare SpringDamper c);
7 end B;
```

In model `B` the component `c` from model `A` is changed to be a `SpringDamper`. The parameter modification `c=500` is inherited (and therefore needs not to be reapplied to the new declaration).

A.4. Control Systems

Modelica also supports to model continuous and discrete control systems. Block diagrams with data-flow semantics and hierarchical state machines are typical conceptual models that are used in this domain. Chapter 5 will discuss that in detail and also propose solutions to currently perceived shortcomings in this area.

Data exchange in block diagrams is directed. Therefore a block can have input and output ports. Modelica provides the prefixes **`input`** and **`output`** that can be used to define *connectors* (ports) that can only be connected according to block diagram semantics.

```
connector RealInput = input Real "'input Real' as connector";

connector RealOutput = output Real "'output Real' as connector";
```

Figure A.10 shows the model of a simple controlled drive where the controller part is modeled as block diagram with causal data-flow while the physical plant component uses acausal connectors. An actuator and a sensor component are used to interface the block diagram domain with the rotational mechanics domain.

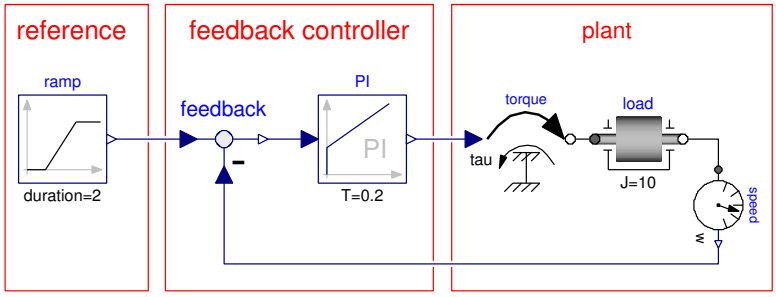


Figure A.10.: Simple controlled drive with continuous PI controller.

A.5. Summary

This brief introduction to Modelica is provided, so that readers who had no previous exposure to Modelica can understand and classify material presented in the following chapters. It only describes a small part of the language. For a complete coverage see the Modelica language specification [73] and corresponding text books and tutorials, e.g., [47].

B. Formal Translation Semantics

The content of this chapter has been published in

> 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. http://dx.doi.org/10.4173/mic.2015.1.3.

The following text is a citation from the above mentioned manuscript.

B.1. The Synchronous Data-Flow Kernel (SDFK)

The utilized synchronous data-flow kernel (SDFK) is described in [19], i.e.,

> Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. Clock-directed modular code generation for synchronous data-flow languages. SIGPLAN Not., 43(7):121–130, June 2008. http://doi.acm.org/10.1145/1379023.1375674.

In order to keep the following discussion more self-contained, the syntax and intuitive semantics[1] described in [19, Section 2] are briefly reproduced in the current section.

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

$$\begin{aligned} \textbf{if } x \textbf{ then } e_2 \textbf{ else } e_3 = \textbf{merge } x \\ (\textbf{True} \rightarrow e_2 \textbf{ when True}(x)) \\ (\textbf{False} \rightarrow e_3 \textbf{ when False}(x)) \end{aligned} \tag{B.1}$$

[1] The semantics of the synchronous data-flow kernel is given informally in [19]. The paper refers to [34] for the formal semantics of the clock calculus. Actually, the extended language presented in [34] is similar to the one used in [19]. In [34] this extended language is formally translated into a more "basic" data-flow kernel by a source-to-source transformation. For this "basic" data-flow kernel [34, Section 3.1] refers to [35] for a (formal) denotational Kahn semantics (except for the semantics of the modular reset operator "**every**" which is formally defined in [51]).

Table B.1.: Expressions in SDFK

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

Note that the clock annotations "ct" have no impact on the data-flow semantics of the language. Clocks do not have to be explicitly given in the SDFK language, e.g., instead of writing "$((v$ **fby** $x^{ck})^{ck} + y^{ck})^{ck}$" it suffices to write "$((v$ **fby** $x) + y)$". Clock annotations in the SDFK language are determined automatically by a clock calculus which is defined as a type inference system. This clock calculus precedes the code generation step (see [19, Section 2.2] for more details).

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

$$
\begin{aligned}
td &::= \textbf{type}\ bt \mid \textbf{type}\ bt = C + \ldots + C \\
d &::= \textbf{node}\ f(p) = p\ \textbf{with var}\ p\ \textbf{in}\ D \\
p &::= x : bt; \ldots; x : bt \\
D &::= pat = a \mid D\ \textbf{and}\ D \\
pat &::= x \mid (pat, \ldots, pat) \\
a &::= e^{ck}
\end{aligned}
$$

$$
\begin{aligned}
e &::= v \mid x \mid (a,\ldots,a) \mid v \textbf{ fby } a \mid op(a,\ldots,a) \\
&\quad \mid f(a,\ldots,a) \textbf{ every } a \mid a \textbf{ when } C(x) \\
&\quad \mid \textbf{merge } x\ (C \to a)\ldots(C \to a) \\
v &::= C \mid i \\
ct &::= ck \mid ct \times \ldots \times ct \\
ck &::= \textbf{base} \mid ck \textbf{ on } C(x)
\end{aligned}
$$

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

Some of the traditional operations supported by Lustre are related to SDFK through the following equivalences:

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

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

Table B.2.: Examples for applying the SDFK operators

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

B.2. mACG-Modelica

To allow a clear and brief presentation of the translational semantics the ACG-Modelica language is further reduced to a subset of representative elements. The resulting language kernel is denoted as mACG-Modelica.

A program is made of a list of global type ("*td*"), connector ("*cd*") and block ("*bd*") declarations. Only abstract types "*t*" are considered (nevertheless the provided examples will use concrete Modelica types, e.g., replacing "*t*" by "`Real`"). Connectors "*cd*" have either input or output causality (Rule 2). A block declaration "*d*" has the form "**block** *id p* **equation** *D* **end**;", where "*id*" is the name of the block, "*p*" contains the local component declarations and "*D*" the equation declarations. A component declaration "*p*" can be modified by a modification "*mo*" (support of simplified modifications according to Rule 4). Parameters can be declared with modification expression "**parameter** *t x* = *me*;", or without modification binding "**parameter** *t x*;"[2].

The use of component dot accesses for parameters is not supported in the presented translation in order to simplify the presentation. Allowing it would require to additionally introduce component dot access normalization (Figure B.8) and "dummy" equation generation (Figure B.9) in a slightly adapted form for parameters in the normalization step. For the actual translation step it would be necessary to translate all parameters not only to node input arguments, but also to node outputs arguments. This seems to make the translation harder to understand without adding much additional conceptional value[3].

Equations "*D*" are either connect equations "**connect**(cx, cx)" or equations of the form "$cx = e$", where "*cx*" is a single variable (the unknown of the equation, which is either accessed by a simple identifier "*x*", or by using a component dot access "*x.x*") and "*e*" is an expression (Rule 2). Similar to SDFK an abstract n-ary operator "$op(e, \ldots, e)$" is provided to simplify the presentation (nevertheless the provided examples will be presented using concrete Modelica operators).

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

$$
\begin{aligned}
td &::= \textbf{type}\ t; \\
bd &::= \textbf{block}\ id\ p\ \textbf{equation}\ D\ \textbf{end}; \\
cd &::= \textbf{connector}\ id\ =\ c\ t; \\
c &::= \textbf{input} \mid \textbf{output} \\
p &::= p\ p \mid t\ x; \mid t\ x\ mo; \mid c\ t\ x; \mid c\ t\ x\ mo; \mid \textbf{parameter}\ t\ x; \mid \textbf{parameter}\ t\ x\ =\ me; \\
mo &::= (ar\ ,\ldots,\ ar) \\
ar &::= id\ =\ me \\
D &::= D\ D \mid eq; \\
e &::= v \mid cx \mid op(e,\ldots,e) \mid \textbf{previous}(x) \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \\
me &::= v \mid x \mid op(me,\ldots,me) \mid \textbf{if}\ me\ \textbf{then}\ me\ \textbf{else}\ me \\
eq &::= cx\ =\ e \mid \textbf{connect}(cx, cx) \\
cx &::= x\ .\ x \mid x
\end{aligned}
$$

[2] Modelica semantics require that modification bindings for parameters have *parametric* or *constant variability* [73, Section 3.8]. This needs to be ensured by a statical check *before* the translation (for conciseness the description of that check is omitted).

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

$x ::= id$
$v =$ value
$id =$ identifier

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

From all the synchronous language elements that are listed in Table 5.3 on page 48 only "**previous**" appears in the grammar above. The clock conversion operators "**subSample**" and "**noClock**" are omitted — as described in Section 5.7.5 they are not necessarily indispensable elements in the Modelica subset targeted for automatic code generation. Nevertheless, Section 5.9.4 shows how the translational semantics can be extended to support these operators.

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

B.3. A Multilevel Translation Approach

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

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

Normalization and translation are defined as a system of mutually recursive functions. The normalization is needed in order to transform mACG-Modelica into a (simplified) *normalized* form which is the basis for the translation to SDFK. The syntax for the *normalized* mACG-Modelica language can be found in Section B.5.

Using a multilevel translation approach facilitates to include further language elements, as long as a source-to-source transformation into a smaller language kernel can be given. The

generation of connection equations is a good example for this: it eliminates the connector declarations and connect equations by replacing them with simple variable declarations (using the proper input/output causalities) and simple equations of the form "$x = e$". Hence, the multilevel translation approach provides a path to incrementally extend the mACG-Modelica subset until the complete ACG-Modelica language is supported (see also the discussion in Section 5.9.3).

B.4. Normalization

B.4.1. Notation

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

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

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

B.4.2. Generation of Connection Equations

The generation of connection equations is defined as a source-to-source transformation through a set of mutually recursive functions. G(D) at the bottom of Figure B.1 defines the translation of Modelica code that includes **`connect`**-equations into a Modelica code in which these **`connect`**-equations are replaced by simple connection equations of the form "$x = e$".

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

GTcd traverses the class declarations. It removes the connector declarations and utilizes function GTd (see Figure B.2) to replace **`connect`**-equations by simple equations of the form "$x = e$".

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

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

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

Figure B.1.: Generation of connection equations.

$$\mathrm{GTd}_{\left((I+[t:t_I],O,Bs),Bd\right)}(t\ x) = \textbf{input}\ t_I\ x$$

$$\mathrm{GTd}_{\left((I+[t:t_I],O,Bs),Bd\right)}(t\ x(\textbf{start}=v)) = \textbf{input}\ t_I\ x(\textbf{start}=v)$$

$$\mathrm{GTd}_{\left((I,O+[t:t_O],Bs),Bd\right)}(t\ x) = \textbf{output}\ t_O\ x$$

$$\mathrm{GTd}_{\left((I,O+[t:t_O],Bs),Bd\right)}(t\ x(\textbf{start}=v)) = \textbf{output}\ t_O\ x(\textbf{start}=v)$$

x – internal input, y – internal output:

$$\mathrm{GTd}_{\left((I+[t_1:\_],O+[t_2:\_],Bs),\ (b,d_b+[x:t_1]+[y:t_2])\right)}(\textbf{connect}(x,y)) = y = x$$

x – internal output, y – internal input:

$$\mathrm{GTd}_{\left((I+[t_2:\_],O+[t_1:\_],Bs),\ (b,d_b+[x:t_1]+[y:t_2])\right)}(\textbf{connect}(x,y)) = x = y$$

$c.x$ – external input, y – internal input:

$$\mathrm{GTd}_{\left((I+[t_1:\_]+[t_2:\_],O,Bs+[(t_c,d_c+[x:t_1])]),\ (b,t_b+[c:t_c]+[y:t_2]])\right)}(\textbf{connect}(c.x,y)) = c.x = y$$

$c.x$ – external output, y – internal output:

$$\mathrm{GTd}_{\left((I,O+[t_2:\_]+[t_1:\_],Bs+[(t_c,d_c+[x:t_1])]),\ (b,d_b+[c:t_c]+[y:t_2])\right)}(\textbf{connect}(c.x,y)) = y = c.x$$

$$\mathrm{GTd}_{(Cd,Bd)}(\textbf{connect}(y,c.x)) = \mathrm{GTd}_{(Cd,Bd)}(\textbf{connect}(c.x,y))$$

$c.x$ – external input, $c.y$ – external output:

$$\mathrm{GTd}_{\left((I+[t_1:\_],o+[t_2:\_],Bs+[(t_c,d_c+[x:t_1])]+[(t_m,d_m+[y:t_2])]),\ (b,d_b+[c:t_c]+[m:t_m])\right)}(\textbf{connect}(c.x,m.y)) = c.x = m.y$$

$c.x$ – external output, $c.y$ – external input:

$$\mathrm{GTd}_{\left((I+[t_2:\_],O+[t_1:\_],Bs+[(t_c,d_c+[x:t_1])]+[(t_m,d_m+[y:t_2])]),\ (b,d_b+[c:t_c]+[m:t_m])\right)}(\textbf{connect}(c.x,m.y)) = m.y = c.x$$

$$\mathrm{GTd}_{(Cd,Bd)}(a) = a;\text{, for the remaining forms of } a$$

$$\mathrm{GTd}_{(Cd,Bd)}(D_1;\ \ldots;\ D_n;) = \mathrm{GTd}_{(Cd,Bd)}(D_1);\ \ldots;\ \mathrm{GTd}_{(Cd,Bd)}(D_n);$$

Figure B.2.: Function GTd—Replace `connect`-equations by equations of the form "$x = e$".

B.4.3. Modification and Dot Access Normalization

This normalization step includes the stripping of parameter modifications, the normalizing of component modifications and the extraction of component dot accesses appearing in expressions. As previously, it is defined as a source-to-source transformation through a set of mutually recursive functions. The translation makes use of a couple of auxiliary functions defined in Figure B.3. A notable complication of the name decoration function $\mathrm{dn}_{(D,p)}(x)$ in Figure B.3 is that

Name decoration, combining p, x to _p_x:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure B.3.: Auxiliary functions for the normalization.

it may be used at places where it is required that the decorated name introduces a fresh variable — i.e., a variable of the same name must not already exist in its scope. To ensure that a fresh name is introduced one may provide a set of names D which are forbidden as a result of name decoration. The second equation in Figure B.3 handles cases in which the resulting name would be in D by prefixing an additional "_" to the name until the resulting name is not any longer in D.

Function $\mathrm{N}(D)$ at the bottom of Figure B.4 defines the translation from the mACG-Modelica language into the *normalized* mACG-Modelica language. This is achieved in three steps:

1. Creation of an auxiliary representation structure $C = \text{NCenv}_{[]}(D)$ (defined in Figure B.5) which encodes the mACG-Modelica block declarations "D" in a structure that facilitates the normalization transformation in the subsequent step,
2. application of the normalization transformation to the auxiliary structure, $C_N = \text{NormC}(C)$ (defined in Figure B.6 and discussed in more detail later), and finally
3. output of the auxiliary structure as *normalized* mACG-Modelica using function $\text{NdeclList}(C_N)$.

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

Figure B.4.: Translation from the mACG-Modelica language into the *normalized* mACD-Modelica language. The translation function $\text{N}(D)$ utilizes functions NCenv (Figure B.5) and NormC (Figure B.6) for the transformation and the remaining functions for transforming the auxiliary representation structure C_N to *normalized* mACG-Modelica.

The auxiliary representation structure $C = [(b_1, p_1, m_1, r_1, q_1);\ \ldots;\ (b_n, p_n, m_n, r_n, q_n)]$ is a list of block class declarations within the (global) environment. b_i is a class declaration name.

$p_i = [x_1 : t_1 : e_1;\ \ldots;\ x_n : t_n : e_n]$ is the list of parameter declarations of block b_i with their respective type t_i and an optionally bound expression e_i. The case that no expression is bound to a parameter e_i is denoted by the symbol $\perp$ (e.g., $x : t : \perp$).

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

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

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

Function NormC in Figure B.6 performs the actual normalization transformation on the auxiliary block representation structure C constructed in function NCenv (see Figure B.5). It applies functions NB and NbInst to each block. The translation differs depending on whether a block is used as instance in another block, or whether a block is used as the top-level instance. Although the final program will have only one top-level instance, function NormC always creates two versions of a block: One version that is used if the block is used as instance in another block (using function NB) and one version that is used if the block is used as top-level instance (using function NbInst which is described in more detail in Section B.4.4).

Function NB first strips all parameter modifications from a block using the auxiliary function NBstrip. This is necessary, because the parameters of a block will later be transformed to node inputs in the SDFK language which resemble function arguments. Therefore, the parameters of a block need to be set at the place there the block/node is instantiated and not within the block itself.

After that, function NB uses function NBm (defined in Figure B.7) to normalize component modifications for every component instantiated in the current block. NBm first extracts all parameters from the component's block definition and introduces them as new (decorated) parameters in the block enclosing the component declaration. During that process it needs to be ensured that component modifications applied in the enclosing block replace parameter modifications in the component's block definition.

As a next step function NB applies function NBq (defined in Figure B.8) to extract and normalize nested component dot accesses appearing in RHS expressions of equations. It traverses a RHS expressions e and searches for occurrences of component dot accesses $c.a$ in e. If it finds one, the component dot access is extracted and a *fresh* (decorated) variable $x_{ca} = \mathrm{dn}_{(D,c)}(a)$ is introduced and bound to $c.a$ in a new equation $x_{ca} = c.a$. The component dot access $c.a$ in e is then replaced by x_{ca}. In order to ensure that any component dot access $c.a$ is uniquely associated to *exactly one* equation $x_{ca} = c.a$ it is checked whether an equation association $x_{ca} = c.a$ already exists — if so, the variable x_{ca} is reused and no fresh variable is introduced.

And finally function NB applies function NBo (defined in Figure B.9) to add "dummy" equations for component outputs that are not accessed in in the instantiating block. This is necessary, since after normalization it is required that *all* outputs of a component actually have a binding equation in a block that instantiates the component. Hence, function NBo first extracts all RHS component dot access variables that appear in the block equations for the respective compo-

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

Figure B.5.: Function NCenv—Create an auxiliary representation structure for mACG-Modelica block declarations.

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

Figure B.6.: Function NormC—Normalization transformation on the auxiliary structure C. Applies NB and NbInst (see Section B.4.4) to each block. Function NB uses NBstrip to remove parameter modifications, NBm (defined in Figure B.7) to normalize component modifications, NBq (defined in Figure B.8) to extract and normalize dot accesses appearing in RHS expressions and NBo (defined in Figure B.9) to add "dummy" equations for component outputs that are not accessed.

$$
\begin{aligned}
&\mathrm{NBm}_{\left(C+[(t_c,[p_{t_1}:t_{t_1}:e_{t_1};\ \ldots;\ p_{t_k}:t_{t_k}:e_{t_k}],\_,\_,\_)],\right.}\\
&\quad\left.(b_B,p_B,m_B+[(c:t_c,\_)],r_B,q_B)\right)\\
&\qquad\left(c:t_c,[p_{c_1}=e_{c_1};\ \ldots;\ p_{c_n}=e_{c_n}]\right)
\end{aligned}
$$

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

Figure B.7.: Function NBm—Normalize component modifications. All parameters from the modified component are extracted from the component's block definition and are introduced as fresh (decorated) parameters in the block enclosing the component declaration. During that introduction it is ensured that the components modification replace the parameter modifications from the component's block definition.

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

Figure B.8.: Function NBq—Normalize RHS component dot access in equations. The RHS expression e is traversed by function NBe. An occurrence of a component dot accesses $c.a$ in e is extracted and a *fresh* (decorated) variable $x_{ca}=\mathrm{dn}_{(D,c)}(a)$ is introduced and bound to $c.a$ in a new equation $x_{ca}=c.a$. The component dot access $c.a$ in e is then replaced by x_{ca}. It is also checked whether $c.a$ already *is associated* to an equation $x_{ca}=c.a$ — in that case *no* fresh variable is introduced and x_{ca} is reused. This ensures that any component dot access $c.a$ is uniquely associated to exactly one equation $x_{ca}=c.a$.

nent. Since component dot accesses have been previously normalized to $x = c.a$ by function NBq, the set Y_{B_o} of all RHS component dot access variables of component c is obtained by $Y_{B_o} = \{o \mid [\_ = c.o] \in q_B\}$. The remaining relations check whether there are outputs in c which have no binding in the current block context and create "dummy" variables and equations if such outputs exist.

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

Figure B.9.: Function NBo—Add "dummy" equations for component outputs that are not accessed in the instantiating block. Normalization requires that *all* outputs of a component have bindings to equations in the block that instantiates the component.

B.4.4. Top-Level Block Instantiation

For any block declaration with name m in mACG-Modelica, function NbInst, defined in Figure B.10, creates a fresh block with its name decorated by the string "*Inst*". The fresh block duplicates inputs and outputs of the instantiated block m and adds equations to connect its inputs and outputs to the inputs and outputs of m.

As a last step it calls function NBm (see Figure B.7) on the freshly constructed block. This call ensures that parameter modifications in the block definition of m are applied at the created instance of m and that parameters in m that have no binding modification are introduced as parameters (likewise with no binding modifications) in the fresh block. Note that parameters that have no binding modification will be turned to additional node inputs in the subsequent translation to the SDFK language (see Section B.6).

B.5. *Normalized* mACG-Modelica

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

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

Figure B.10.: Function NbInst—Create block instance wrapper block.

grammar:

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

B.6. Translation

This section defines the translational semantics for the *normalized* mACG-Modelica language. Hence, semantics of modeling constructs in *normalized* mACG-Modelica are expressed in terms of constructs from the SDFK language. The translation is defined as a set of mutually recursive functions reusing the notation introduced in Section B.4.1, page 128. Additionally it relies on a suitable lexicographical order relation "$<_L$" to allow an unambiguous ordering of node input and output arguments (see Figure B.11).

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

Figure B.11.: Lexicographical order relation and function SortList—Auxiliary relation and function for arranging node input or output arguments in alphabetical order. The elements of the list *xs* have the structure $name : type$. They are sorted by a suitable lexicographical order relation on *name*.

Function $T(.)$ in Figure B.12 defines the translation of a block from the *normalized* mACG-Modelica language to the SDFK language. Note that the translation is performed block-by-block, without requiring a context of global block declarations as it was required during normalization.

Translate list of equations into SDFK equations:

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

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

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

Figure B.12.: Translation from the *normalized* mACG-Modelica language into the SDFK language. The translation function $\text{T}(.)$ utilizes functions CId (defined in Figure B.13) and CEq (defined in Figure B.14) to create auxiliary structures d, i, o, l, s, j, q which are further used in the translation functions TEqList and TCList and their supporting functions TE (defined in Figure B.15) and TC (defined in Figure B.16).

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

$d = [x_1 : t_1; \ldots; x_n : t_n]$ stands for a list of declarations within the scope, $i = [x_1 : t_1; \ldots; x_n : t_n]$

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

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

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

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

Figure B.13.: Function CId—Create auxiliary structure for instance declarations.

Most of function CId (defined in Figure B.13) is rather straightforward. It seems worth mentioning that a parameter with a bound expression is added as equation to q while a param-

eter without a bound expression is added as input to i. The parameter modifications ($m_1 = e_1, \ldots, m_n = e_n$) in component declarations are mapped to input variables with their respective bound expression in the block components environment j ($[m_1 \mapsto e_1; \ \ldots; m_n \mapsto e_n]$).

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

Figure B.14.: Function CEq—Create auxiliary structure for equation declarations.

Function CEq (defined in Figure B.14) adds equations from a block to q, unless the equation includes a component dot access. In that case the block components environment j is modified and the equation is either mapped to the inputs ci or outputs co bindings of that component.

Once CEq returns the auxiliary representation structure d, i, o, l, s, j, q, the elements i, o, and l translate (after applying SortList to i and o) directly to an SDFK node signature and its local variable declarations. Equations gathered in q are translated to SDFK equations by function TEqList, which in turn applies TE to each RHS expression in q. Function TE (defined in Figure B.15) translates mACG-Modelica expressions into SDFK expressions.

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

Figure B.15.: Function TE—Translate mACG-Modelica expression into SDFK expression

Finally j, in which information of block component usage is collected, is translated to SDFK equations by function TCList, which in turn applies function TC to each element of j. Function

TC (defined in Figure B.16) translates collected component dot accesses to an SDFK equation with an RHS node instance. For example, a *normalized* mACG-Modelica code containing the dot accesses `c.u1=x1;`, `c.u2=x2;` `x3=c.y1;` `x4=c.y2;` for a component `c` would finally be translated into the SDFK equation `(x3,x4)=c(x1,x2).`

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

Figure B.16.: Function TC—Translate component dot accesses to an SDFK equation with an RHS SDFK node instance.

Bibliography

[1] W. B. Ackerman. Data flow languages. *Computer*, 15(2):15–25, 1982.

[2] J. Åkesson, U. Nordström, and H. Elmqvist. Dymola and Modelica_EmbeddedSystems in Teaching–Experiences from a Project Course. In 7th *Int. Modelica Conference*, pages 603–611, 2009.

[3] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, February 1995.

[4] R. Alur, C. Courcoubetis, T. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. *Hybrid systems*, 736:209–229, 1993.

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

[6] C. V. Anoop and C. Betta. Comparative Study of Fixed-Point and Floating-Point Code for a Fixed-Point Micro. In *dSPACE User Conference*, Bangalore, India, Sept 2012.

[7] P. J. Ashenden, G. D. Peterson, and D. A. Teegarden. *The System Designer's Guide to VHDL-AMS*. Morgan Kaufmann Publishers, San Francisco, September 2002.

[8] AUTOSAR. Specification of Interaction with Behavioral Models. Part of Release 4.1, Jan., 18 2013. http://www.autosar.org/.

[9] N. Bajçinca, R. Cortesão, and M. Hauschild. Robust control for steer-by-wire vehicles. *Autonomous Robots*, 19:193 – 214, 08 2005.

[10] P. Barnard. Software development principles applied to graphical model development. In *AIAA Modeling and Simulation Technologies Conference and Exhibit, San Francisco*, 2005.

[11] J. Batteh and M. Tiller. A modular model architecture in Modelica for rapid virtual prototyping of conventional and hybrid ground vehicles. In *Ground Vehicle Systems Engineering and Technology Symposium (GVSETS)*, pages 493–502, Bielefeld, Germany, March 2009.

[12] K. Bauer and K. Schneider. From synchronous programs to symbolic representations of hybrid systems. In *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*, HSCC '10, pages 41–50, New York, NY, USA, 2010. ACM.

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

[14] A. Benveniste, B. Caillaud, and M. Pouzet. The fundamentals of hybrid systems modelers. In *49th IEEE International Conference on Decision and Control (CDC'10)*, pages 4180–4185, Atlanta, Georgia, USA, December 15-17 2010.

[15] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: specification and distributed code generation. *Information and Computation*, 163:125–171, 2000.

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

[17] M. C. Berg, N. Amit, and J. D. Powell. Multirate digital control system design. *Automatic Control, IEEE Transactions on*, 33(12):1139–1150, 1988.

[18] D. Bhatt, B. Hall, S. Dajani-Brown, S. Hickman, and M. Paulitsch. Model-based development and the implications to design assurance and certification. In *Digital Avionics Systems Conference, 2005. DASC 2005. The 24th*, volume 2, 2005.

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

[20] M. Bonvini, F. Donida, and A. Leva. Modelica as a design tool for hardware-in-the-loop simulation. In 7th *Int. Modelica Conference*, pages 378–385, Como, Italy, Sep. 2009.

[21] F. Boussinot and R. De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.

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

[23] D. Broman, P. Fritzson, and S. Furic. Types in the Modelica Language. In 5th *Int. Modelica Conference*, pages 303–315, Vienna, Austria, September 2006.

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

[25] C. Buckl. *Model-Based Development of Fault-Tolerant Real-Time Systems*. PhD thesis, Lehrstuhl für Echtzeitsysteme und Robotik, Technische Universität München, 2008.

[26] C. Buckl, A. Knoll, and G. Schrott. Model-based development of fault-tolerant embedded software. In *Proceedings of Second International Symposium on Leveraging Applications of Fomal Methods, Verification and Validation (IEEE-ISoLA)*, 2006.

[27] C. Buckl, M. Regensburger, A. Knoll, and G. Schrott. Models for automatic generation of safety-critical real-time systems. In *Second International Conference on Availability, Reliability and Security (ARES)*, 2007.

[28] T. Bünte, J. Brembeck, and L. M. Ho. Human machine interface concept for interactive motion control of a highly maneuverable robotic vehicle. In *The Intelligent Vehicles Symposium*, Baden-Baden, Germany, June 2011.

[29] T. Bünte, A. Sahin, and N. Bajcinca. Inversion of Vehicle Steering Dynamics with Modelica/Dymola. In G. Schmitz, editor, 4th *Int. Modelica Conference*, pages 319–328, Hamburg, Germany, March 2005.

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

[31] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, LCTES '03, pages 153–162, New York, NY, USA, 2003. ACM.

[32] E. Chrisofakis, A. Junghanns, C. Kehrer, and A. Rink. Simulation-based development of automotive control software with Modelica. In 8th *Int. Modelica Conference*, pages 1–7, Dresden, Germany, March 2011.

[33] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2010.

[34] J.-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous dataflow with state machines. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 173–182, New York, NY, USA, 2005. ACM.

[35] J.-L. Colaço and M. Pouzet. Clocks as First Class Abstract Types. In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, October 2003.

[36] A. Deuring, J. Gerl, and H. Wilhelm. Multi-Domain Vehicle Dynamics Simulation in Dymola. In 8th *Int. Modelica Conference*, pages 13–17, Dresden, Germany, March 2011.

[37] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42:42–52, April 2009.

[38] M. Eder and A. Knoll. Design of an Experimental Platform for an X-by-wire Car with Four-wheel Steering. In *Proc. IEEE Conf. Automation Science and Engineering (CASE)*, pages 656–661, 2010.

[39] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, J. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127 – 144, jan. 2003.

[40] H. Elmqvist. *A Structured Model Language for Large Continuous Systems*. PhD thesis, Lund Institute of Technology, Lund, Sweden, 1978.

[41] H. Elmqvist. An Object and Data-Flow Based Visual Language for Process Control. In *ISA/92-Canada Conference & Exhibition*, pages 181–192, Toronto, Canada, April 1992. Instrument Society of America.

[42] H. Elmqvist, M. Otter, D. Henriksson, B. Thiele, and S. E. Mattsson. Modelica for Embedded Systems. In 7th *Int. Modelica Conference*, pages 354–363, Como, Italy, September 2009.

[43] H. Elmqvist, M. Otter, and S. E. Mattsson. Fundamentals of Synchronous Control in Modelica. In M. Otter and D. Zimmer, editors, 9th *Int. Modelica Conference*, pages 15–26, Munich, Germany, September 2012.

[44] 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.

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

[46] R. Franke, B. Babji, M. Antoine, and A. Isaksson. Model-based online applications in the ABB Dynamic Optimization framework. In B. Bachmann, editor, 6th *Int. Modelica Conference*, pages 279–285, Bielefeld, 2008.

[47] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, 2003.

[48] D. Glasson. Development and applications of multirate digital control. *Control Systems Magazine, IEEE*, 3(4):2–8, 1983.

[49] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[50] N. Halbwachs. A Synchronous Language at Work: the Story of Lustre. In *Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design. MEMOCODE '05.*, pages 3–11, July 2005.

[51] G. Hamon and M. Pouzet. Modular Resetting of Synchronous Data-flow Programs. In *ACM International conference on Principles of Declarative Programming (PPDP'00)*, Montreal, Canada, September 2000.

[52] 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.

[53] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[54] T. Henzinger. The theory of hybrid automata. In *11th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 278–292. IEEE, 1996.

[55] T. Henzinger and J. Sifakis. The discipline of embedded systems design. *Computer*, 40(10):32 –40, oct. 2007.

[56] 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.

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

[58] ISO 26262: Road vehicles – Functional safety, Parts 1 through 10, 2011-2012.

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

[60] ISO 26262-8:2011, Road vehicles – Functional safety – Part 8: Supporting processes, November 2011.

[61] M. Krenn and T. Richter. Active steering-BMW's approach to modern steering technology. In *Braking 2004: Vehicle Braking and Chassis Control*, page 3. Wiley, 2004.

[62] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[63] E. A. Lee. Cyber-physical systems - are computing foundations adequate? In *NSF Workshop on Cyber-Physical Systems*, October 2006.

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

[65] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. http://LeeSeshia.org, ISBN 978-0-557-70857-4, 2011.

[66] E. A. Lee and H. Zheng. Operational semantics of hybrid systems. In M. Morari and L. Thiele, editors, *Hybrid Systems: Computation and Control*, volume 3414 of *Lecture Notes in Computer Science*, pages 25–53. Springer Berlin / Heidelberg, 2005.

[67] J. W. Lloyd. Practical Advantages of Declarative Programming. In M. Alpuente, R. Barbuti, and I. Ramos, editors, *Joint Conference on Declarative Programming, GULP-PRODE'94*, volume 1, pages 18–33, Peñiscola, Spain, September 19–22 1994.

[68] G. Looye. Design of Robust Autopilot Control Laws with Nonlinear Dynamic Inversion. *at - Automatisierungstechnik*, 49(12):523–531, December 2001.

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

[70] P. Löw, R. Pabst, and E. Petry. *Funktionale Sicherheit in der Praxis*. dpunkt.verlag, 2010.

[71] F. Maraninchi and Y. Rémond. Mode-Automata: About Modes and States for Reactive Systems. In *European Symposium On Programming*, Lisbon, Portugal, March 1998. Springer-Verlag.

[72] F. Maraninchi and Y. Rémond. Mode-Automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46:219–254, 2003.

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

[74] P. Mosterman and J. Ciolfi. Using interleaved execution to resolve cyclic dependencies in time-based block diagrams. In *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, volume 4, pages 4057–4062 Vol.4, Dec 2004.

[75] M. Najafi, S. Furic, and R. Nikoukhah. Scicos: a general purpose modeling and simulation environment. In 4th *Int. Modelica Conference*, pages 367–374, Hamburg-Harburg, Germany, March 2005.

[76] M. Najafi and R. Nikoukhah. Implementation of Hybrid Automata in Scicos. In *Control Applications, 2007. CCA 2007. IEEE International Conference on*, pages 819–824, Singapore, October 2007. IEEE.

[77] O. Niggemann, U. Eisemann, M. Beine, and U. Kiffmeier. Behavior Modeling Tools in an Architecture-Driven Development Process - From Function Models to AUTOSAR. In *SAE 2007 World Congress & Exhibition*, Detroit, Michigan, United States, April 2007. SAE Technical Paper 2007-01-0507.

[78] O. Niggemann and J. Stroop. Models for model's sake: why explicit system models are also an end to themselves. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 561–570, New York, NY, USA, 2008. ACM.

[79] R. Nikoukhah and S. Furic. Towards a full integration of Modelica models in the scicos environment. In *7th Modelica Conference*, pages 641–645, Como, Italy, September 2009.

[80] Object Management Group. OMG Systems Modeling Language (OMG SysML) v1.2. Standard Specification, November 2010. OMG document number: formal/2010-06-01, available at http://www.omgsysml.org.

[81] Object Management Group. OMG Unified Modelling Language Specification (OMG UML) v2.4.1. Standard Specification, August 2011.

[82] L. A. Ochel and B. Bachmann. Initialization of Equation-Based Hybrid Models within OpenModelica. In 5th *International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pages 97–103, Nottingham, UK, April 2013.

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

[84] M. Otter, B. Thiele, and H. Elmqvist. A Library for Synchronous Control Systems in Modelica. In M. Otter and D. Zimmer, editors, 9th *Int. Modelica Conference*, pages 27–36, Munich, Germany, September 2012.

[85] L. Parnebo and B. Hansson. Plug and play in control loop design. In *Preprints Reglermöte*, Linköping, Sweden, May 2002.

[86] A. Pop, M. Sjölund, A. Ashgar, P. Fritzson, and F. Casella. Integrated Debugging of Modelica Models. *Modeling, Identification and Control*, 35(2):93–107, 2014.

[87] M. Pouzet. *Lucid Synchrone Tutorial and Reference Manual*, 2006.

[88] M. Pouzet and P. Raymond. Modular Static Scheduling of Synchronous Data-flow Networks: An efficient symbolic representation. *Journal of Design Automation for Embedded Systems*, 3(14):165–192, 2010.

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

[90] A. Robinson. *Non Standard Analysis*. Princeton Landmarks in Mathematics, 1996.

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

[92] W. Schamai, P. Frizson, C. Paredis, and A. Pop. Towards Unified System Modeling and Simulation with ModelicaML: Modeling of Executable Behavior Using Graphical Notations. In 7th *Int. Modelica Conference*, pages 612–621, Como, Italy, September 2009.

[93] K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.

[94] S.-A. Schneider, T. Lovric, and P. R. Mai. The Validation Suite Approach to Safety Qualification of Tools. In *SAE World Congress*, Detroit, MI, USA, April 2009. SAE International.

[95] J. Schäuffele and T. Zurawka. *Automotive Software Engineering – Principles, Processes, Methods and Tools*. SAE International, 2005.

[96] A. M. Sloane. Lightweight Language Processing in Kiama. In J. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 408–425. Springer Berlin Heidelberg, 2011.

[97] P. Smith, S. Prabhu, and J. Friedman. Best Practices for Establishing a Model-Based Design Culture. In *SAE 2007 World Congress & Exhibition*, Detroit, Michigan, United States, April 2007. SAE Technical Paper 2007-01-0777.

[98] T. Stahl and M. Völter. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

[99] I. Stürmer. *Systematic Testing of Code Generation Tools*. PhD thesis, Technische Universität Berlin, 2006.

[100] I. Stürmer, M. Conrad, I. Fey, and H. 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.

[101] I. Stürmer, D. Weinberg, and M. 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.

[102] E. D. Tate, M. Sasena, J. Gohl, and M. Tiller. Model embedded control: A method to rapidly synthesize controllers in a modeling environment. In 6th *Int. Modelica Conference*, 2008.

[103] A. Tewari. *Modern Control Design with Matlab and Simulink*. John Wiley & Sons, Ltd., 2002.

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

[105] The Motor Industry Software Reliability Association. MISRA-C:1998 - Guidelines for the use of the C language in vehicle based systems, 1998. (superseeded by MISRA C:2012) http://www.misra.org.uk.

[106] 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.

[107] 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.

[108] B. Thiele, A. Knoll, and P. Fritzson. Towards Qualifiable Code Generation from a Clocked Synchronous Subset of Modelica. *Modeling, Identification and Control*, 36(1):23–52, 2015.

[109] B. Thiele, S.-A. Schneider, and P. R. Mai. A Modelica Sub-and Superset for Safety-Relevant Control Applications. In M. Otter and D. Zimmer, editors, 9th *Int. Modelica Conference*, pages 455–476, Munich, Germany, September 2012.

[110] T. Thomsen. MISRA C und seine Anwendbarkeit auf Seriencodegeneratoren. Technical report, dSPACE GmbH, 2003.

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

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

[113] L. Vitkin, S. Dong, and M. Chandrashekar. Selection of Fixed-Point or Floating-Point for Production Applications. In *dSPACE User Conference*, 2006.

[114] F. Wagner, L. Lui, and G. Frey. Simulation of Distributed Automation Systems in Modelica. In 6th *International Modelica Conference*, pages 113–122, Bielefeld, Germany, 2008.

[115] M. Whalen and M. 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.

[116] J. Åkesson, T. Ekman, and G. 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).

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

Curriculum Vitae

Persönliche Daten

Name:	Bernhard Amadeus Thiele
Geburtsdatum und -ort:	27.09.1980 in Garmisch-Partenkirchen
Familienstand:	ledig
Adresse:	Rydsvägen 84A lgh 1001 58431 Linköping, Schweden
E-Mail:	bernhard-thiele@web.de

Schul- und Berufsausbildung

2012 – heute **externer Doktorand**
Technische Universität München
Lehrstuhl für Echtzeitsysteme und Robotik
Titel: *Framework for Modelica Based Function Development*
Betreuer: Univ.-Prof. Dr. Alois Knoll

2001 – 2007 **Diplom-Ingenieur Maschinenwesen (Dipl.-Ing. Univ.)**
Technische Universität München
Titel: *Einsatz adaptiver Regelstrategien für aktive Komponenten in Werkzeug-Maschinen*

2004 – 2005 **Auslandssemester**
University of Birmingham, UK
Mechanical Engineering Department and School of Mathematics

2000 – 2001 **Wehrdienst**

1991 – 2000 **Abitur**
Werdenfels-Gymnasium Garmisch-Partenkirchen

Berufspraxis

2014 – heute **wissenschaftlicher Mitarbeiter**
Linköping University
Department of Computer and Information Science
581 83 Linköping, Sweden

2007 – 2014 **wissenschaftlicher Mitarbeiter**
Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR)
Institut für Systemdynamik und Regelungstechnik
Münchner Straße 20, 82234 Weßling