Proceedings of the

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools

Zurich, Switzerland, September 5, 2011

EOOLT 2011

Editors

François E. Cellier
David Broman
Peter Fritzson
Edward A. Lee

ISSN: 1650-3686

4th International Workshop on Equation-based Object-oriented Modeling Languages and Tools

Proceedings

Editors:

François E. Cellier, David Broman, Peter Fritzson, and Edward A. Lee

Copyright

Series: Linköping Electronic Conference Proceedings
Number 56
ISSN (print): 1650-3686
ISSN (online): 1650-3740
http://www.ep.liu.se/ecp_home/index.en.aspx?issue=056

Printed by ETH Zurich, 2011

TABLE OF CONTENTS

Real-time Oriented Modeling Languages and Tools

Preface

During the past decade, integrated model-based design of complex cyber-physical systems (which mix physical dynamics with software and networks) has gained significant attention. Hybrid modeling languages based on equations, supporting both continuous-time and event-based aspects (e.g. Modelica, SysML, VHDL-AMS, and Simulink/ Simscape) enable high-level reuse and integrated modeling capabilities of both the physically surrounding system and software for embedded systems. Using such equation-based object-oriented (EOO) modeling languages, it has become feasible to model complex systems covering multiple application domains at a high level of abstraction through reusable model components.

The interest in EOO languages and tools is rapidly growing in the industry because of their increasing importance in modeling, simulation, and specification of complex systems. There exist several different EOO language communities today that grew out of different application domains (multi-body system dynamics, electronic circuit simulation, chemical process engineering). The members of these disparate communities rarely talk to each other in spite of the similarities of their modeling and simulation needs.

The EOOLT workshop series aims at bringing these different communities together to discuss their common needs and goals as well as the algorithms and tools that best support them.

There was a good response to the Call for Papers. Ten papers were in the end accepted for full presentations whereas two work-in-progress papers were accepted for shorter presentations. One additional presentation was invited by the workshop organizers. All papers were subject to rather detailed reviewing by the program committee; on the average, four reviews were received per manuscript submitted.

On behalf of the program committee, the Program Chairs would like to thank all those who submitted papers to EOOLT'2011. Special thanks go to the program committee members for reviewing the papers on a strict time schedule.

Zurich, August 2011

François E. Cellier
David Broman
Peter Fritzson
Edward A. Lee

INTERNATIONAL PROGRAM COMMITTEE

François E. Cellier (chair)	ETH Zurich	Switzerland
David Broman (co-chair)	Linköping University	Sweden
Peter Fritzson (co-chair)	Linköping University	Sweden
Edward A. Lee (co-chair)	University of California, Berkeley	U.S.A.

Bernhard Bachmann	University of Applied Sciences, Bielefeld	Germany
Bert van Beek	Eindhoven University of Technology	The Netherlands
Jan Broenink	University of Twente	The Netherlands
Peter Bunus	Linköping University	Sweden
Francesco Casella	Polytechnical University of Milano	Italy
Christoph Clauß	Fraunhofer Institute for Integrated Circuits, Dresden	Germany
Olaf Enge-Rosenblatt	Fraunhofer Institute for Integrated Circuits, Dresden	Germany
Xenofon Floros	ETH Zurich	Switzerland
Sven-Erik Mattsson	Dynasim AB, Lund	Sweden
Jakob Mauß	QTronic GmbH, Berlin	Germany
Pieter J. Mostermann	MathWorks, Inc., Natick, Mass.	U.S.A.
Henrik Nilsson	University of Nottingham	United Kingdom
Dionisio de Niz Villaseñor	Carnegie Mellon University	U.S.A.
Martin Otter	DLR Oberpfaffenhofen	Germany
Chris Paredis	Georgia Institute of Technology	U.S.A.
Peter Pepper	Technical University Berlin	Germany
Adrian Pop	Linköping University	Sweden
Nicolas Rouquette	Jet Propulsion Laboratory, NASA	U.S.A.
Carl-Johan Sjöstedt	KTH, Stockholm	Sweden
Christian Sonntag	Technical University Dortmund	Germany
Alfonso Urquía	UNED, Madrid	Spain
Hans Vangheluwe	McGill University	Canada
and	University of Antwerp	Belgium
Dirk Zimmer	DLR Oberpfaffenhofen	Germany
Johan Åkesson	Lund University	Sweden

Languages and Tools For Model-based Design

Advancing Model-Based Design by Modeling Approximations of Computational Semantics

Pieter J. Mosterman[1,2] Justyna Zander[3]

[1]Design Automation Department, MathWorks, USA, pieter.mosterman@mathworks.com
[2]School of Computer Science, McGill University, Canada
[3]Harvard Humanitarian Initiative, Harvard University, USA, justyna.zander@googlemail.com

Abstract

Over the past decades, engineered systems have increasingly come to rely on embedded computation in order to include advanced and sophisticated features. The unparallelled flexibility of software has been a blessing for implementing functionality with a complexity that could not have been imagined heretofore. One important manifestation of this is in the use of software as the universal system integration mechanism. With the increasing use, however, has come a suite of difficulties in effectively employing software engineering practices because (i) C (the language of choice in embedded software implementation) is very close to the hardware implementation and (ii) software engineering methods typically only consider logical correctness, irrespective of critical characteristics for embedded computation (e.g., response time). To address these problems, Model-Based Design helps raise the level of abstraction while accounting for such critical characteristics. The corresponding models are designed using high-level formalisms such as block diagrams and state transition diagrams whose meaning is particularly intuitive because of their executable nature. The necessity to support increasingly complicated language elements, however, has caused the underlying execution engine to explode in complexity. As a result, the meaning of the high-level formalisms exists almost exclusively by merit of simulation. This paper attempts to present the challenges faced by the current state of Model-Based Design tools and outlines a solution approach by modeling the execution engine.

Keywords Model-Based Design, Cyber-Physical Systems, Modeling, Simulation, Computation, Numerical Integration, Hybrid Systems

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zürich, Switzerland.
 Proceedings available at:
http://www.ep.liu.se/ecp/056/
EOOLT 2011 website:
http://www.eoolt.org/2011/

1. Introduction

With the advent of solid state transitors, computation has been on a steady curve of making ever increasing computational power available. Because of the flexibility of software in the design of functionality this trend has been a boon for engineered system designers. Embedding powerful processors into most any engineered system has allowed including features that are close to being limited only by imagination.

However, while miniaturization has been driving an exponential increase in transistor count per surface area, design methods that enable exploitation of the corresponding computing power have been lagging [23]. This problem not only manifests in hardware but also in software. For example, because of integration issues in the avionics software, the F/A-22 fighter continuously struggled to meet its projected deployment dates [25].

The challenges in software producibility [8][1] are perhaps especially curious in the face of the great strides that have been made in programming and software engineering. However, approaches such as *structured programming*, *modularization*, *object-oriented programming*, etc. all concentrate on an *information* rather than a *dynamics* perspective. Consequently, these approaches do not sufficiently benefit the embedded systems or Cyber-Physical Systems (CPS) communities. For example, when it comes to the effects of *embedding* computation in a physical environment the response time of a sequence of computations is critical, yet software engineering approaches typically consider *time* a 'nonfunctional' characteristic (e.g., [10]) even though computational complexity is a standard consideration.

Abstraction is a natural approach to attempt to overcome these difficulties in designing embedded computational features. In CPS design, abstraction attempts to facilitate critical characteristics such as computation timing and data representation by providing high-level formalisms that align well with the problem space (control algorithms, signal processing algorithms, etc.), rather than the solution

[1] In the automotive industry, the extensive use of software has been responsible for distinct quality issues such as documented in "Sync sinks Ford's J.D. Power quality ratings" (http://money.cnn.com/2011/06/22/autos/ford_jd_power_initial_quality).

space (typically C code). The abstraction results in *models* of the underlying software and so the corresponding approach to design is called *Model-Based Design* (e.g., [20]).

Now, to enable Model-Based Design of CPS it is not only necessary to abstract the embedded computation but also the physical world to a representation that is most appropriate for solving the design problem. The use of the most appropriate formalism at the most appropriate level of abstraction constitutes the underlying premise of the field of Computer Automated Multiparadigm Modeling (CAMPaM), which highlights and studies the complications that arise from models at multiple time scales, with multiple levels of detail, and using a combination of multiple formalisms [15, 16, 17]. In this context, it is essential to concentrate on the formalization of the meaning of models that represent the physical world so as to correspond to formalizations typical for embedded computation.

In Section 2, Model-Based Design is introduced in more detail. Section 3 outlines the challenges that have emerged from the increasing use of Model-Based Design. An outline of a possible solution approach is communicated in Section 4. Section 5 concludes the considerations.

2. Model-Based Design

Design of engineered systems is a process in multiple stages where each stage has a set of requirements as input and a specification as output (e.g., [7]). A produced specification then serves as requirements for the next design stage where additional information necessary to implement a specification is included. This information comprises additional implementation detail as well as functionality. For example, a control law may be specified by a declarative block diagram (e.g., [1]). This specification serves as the requirements for the embedded software of the control law. The block diagram specification then requires additional detail such as data types of its signals so that the specification can be implemented in embedded software. Furthermore, additional functionality such as a scheduler to execute the blocks in the block diagram specification is added.

The rise of Information Technology (IT) has supported digital documents such as created with rich text, presentation, and spreadsheet applications. Likewise, graphical models of dynamic systems such as block diagram models have become available in digital form. Such digital models are at the core of Model-Based Design. This is depicted in Figure 1 where a system specification is represented by a triangle. The computer application that is used to create the digital form of a specification is represented by the rectangle at the base of the triangle. This computer application executes on a host technology, as depicted by the rectangle at the bottom.

Because a digital form of the specification exists, any models that it contains can now be processed automatically. Model-Based Design has exploited this substantially by enabling the (i) execution of dynamic models, (ii) transformation of models to include implementation detail (or even complete C code), and (iii) integration of models with further design artifacts such as requirements and tests. In turn, the availability of all these specific features has formed the catalyst to developing an entire test and verification ecosystem around the digital models through the various design stages.

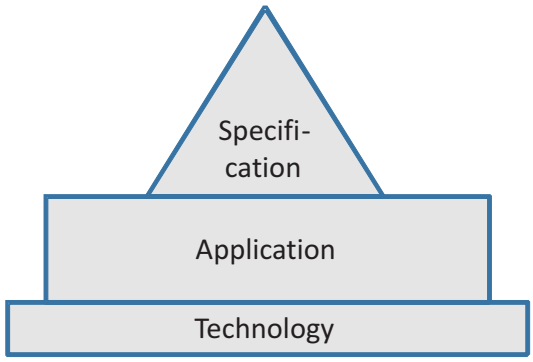


Figure 1. Digital representation of models.

3. The Challenge

By making the digital representation of a model executable, Model-Based Design introduces approximations necessary for simulation algorithms. For example, the behavior of a model as a differential equation system is typically approximated by numerical integration algorithms that are implemented in software modules (e.g., [21]). In case such a continuous-time model includes discrete mode changes, a *hybrid dynamic system* [3, 12] emerges that comprises further approximations (e.g., because of the root-finding algorithms that detect the point in time when a discrete mode change occurs [13] and the reinitialization algorithms that may follow such a mode change [14]). These approximations are depicted in Figure 2 by representing the original system specification triangle of Figure 1 as a discretized triangle. Often the approximations are specific to the particular Model-Based Design (MBD) application that is used (e.g., Simulink® [24]).

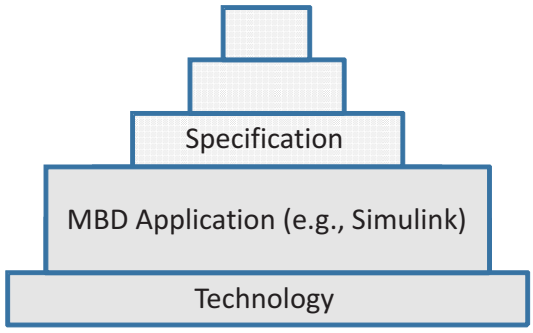


Figure 2. Computational approximation of executable models.

In spite of these approximations, Model-Based Design has shown to be very successful [9, 11]. To a large extent this is precisely because an executable model exists. For example, a computational model of a physical system can be fit to constitute a precise representation by automatic system identification methods that rely on a multitude of simulations to search for the optimal underlying model. As the

digital models have become prime design deliverables between design stages, the approximations are carried consistently through to the eventual implementation. As a result, the computational models can be successfully exploited, though at the expense of a necessity for extensive testing throughout. Moreover, any analysis other than simulation based is prohibitively complex, as it requires analysis of the entire execution engine code base.

To further mature and advance Model-Based Design, the challenge that developers of Model-Based Design applications face is then one of defining the computational approximations. Specifically, the computational approximations are best *modeled* at a declarative level, as illustrated in Figure 3. Such a model of the execution engine enables analyses of the models that the execution engine executes and that are used in design, while accounting for the inherent computational approximations. Moreover, the declarative representation enables synthesis methods based on computational approaches such as *model checking* (e.g., [19]).

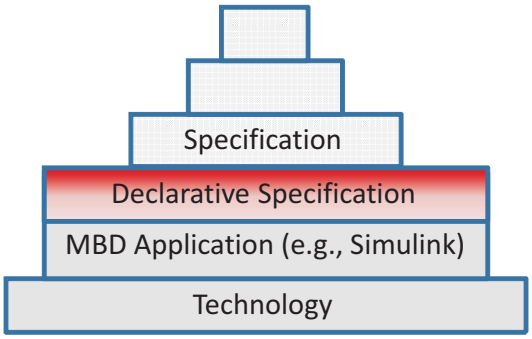


Figure 3. Declarative specification of computational approximations.

4. A Solution Approach

An approach to tackle the challenge of defining the computational approximations must be declarative and sufficiently formal so as to serve as a specification or reference implementation. The approach must support analysis and synthesis while being sufficiently powerful to apply to a broad range of models of computation.

In previous work, Haskell was employed as the definition language [5] which satisfies the declarative requirement. Moreover, the functional nature supported a stream-based approach (i.e., 'state-less behavior' of functions) which aids greatly in analyzability (e.g., [2]). Furthermore, work on synchronous languages [4, 6] has demonstrated the breadth of applicability of such a stream-based declarative approach while enabling synthesis of an implementation in terms of a state machine or software code.

More recently, the approximations of a variable-step numerical integration algorithm have been formally modeled in a declarative sense by a strict subset of Simulink blocks [18, 19]. This subset includes only 'delay' and 'latch' operations as so-called 'sequential' operations [22] that communicate between sample times. For example, Figure 4 shows a model of the *Euler* and Figure 5 of the *trapezoidal* integration algorithms as they are used by a Heun numerical integration scheme. Such a scheme employs two stages, where in the first stage a forward Euler approximation is computed while in the second stage the Euler estimate is used to compute a trapezoidal approximation.

The Euler and Heun approximations are then employed in a variable-step numerical integration algorithm that compares the two approximations and in case a tolerance is exceeded, the step size is reduced by a factor two. Because in this case the integration has to undo the original approximation as produced by the *aggregate series* facility, the Euler (Figure 4) and Trapezoidal approximation (Figure 5) include a *reject series* facility to selectively subtract the approximations of failed time steps.

Because of the declarative model of a variable-step solver, detailed analyses of the semantics in the face of zero-crossings are enabled [26]. Moreover, it allows a feedforward control to be synthesized for a stiff hybrid dynamic system by using computational model checking methods [19].

5. Conclusions

In this paper, a vision for continuous improvement of Model-Based Design has been outlined. It was argued that Information Technology ultimately enabled the development of Model-Based Design principles that can be applied for developing computable and executable graphical models. Such models introduce the necessity to properly understand the semantics of the computation because the power of computational execution amplifies numerical approximations. This highlights the challenge that Model-Based Design is facing today. Whenever a specification of a system is considered, its computational meaning comprises the approximation of the computation method used for simulation. In this manner, the semantics of a system design permeates down to the implementation detail of the simulation engine (in some cases, it is even affected by host technology specifics). This, in turn, renders activities such as system analysis, design refinement, and synthesis very complex and prone to human mistakes or errors, even (or especially) in case automatic processing is exploited.

To mitigate the complexity, a separate conceptual and simultaneously technical layer is introduced to the existing Model-Based Design artifacts. This layer is a formal declarative specification that is applicable to a broad range of models of computation. Its functional nature is supported by a stream-based approach, which lends itself well to analysis. As an example, a declarative stream-based implemention is outlined for a variable-step numerical integration algorithm (based on a forward Euler and Heun approximation).

The discussion above applies in particular to Cyber-Physical Systems (CPS) where both embedded computation and the dynamics of the physical world must be represented in a unified computational manner to allow for comprehensive analyses. Neglecting the computational approximation and execution semantic for such a combination is prone to improper results and unpredictable conclusions.

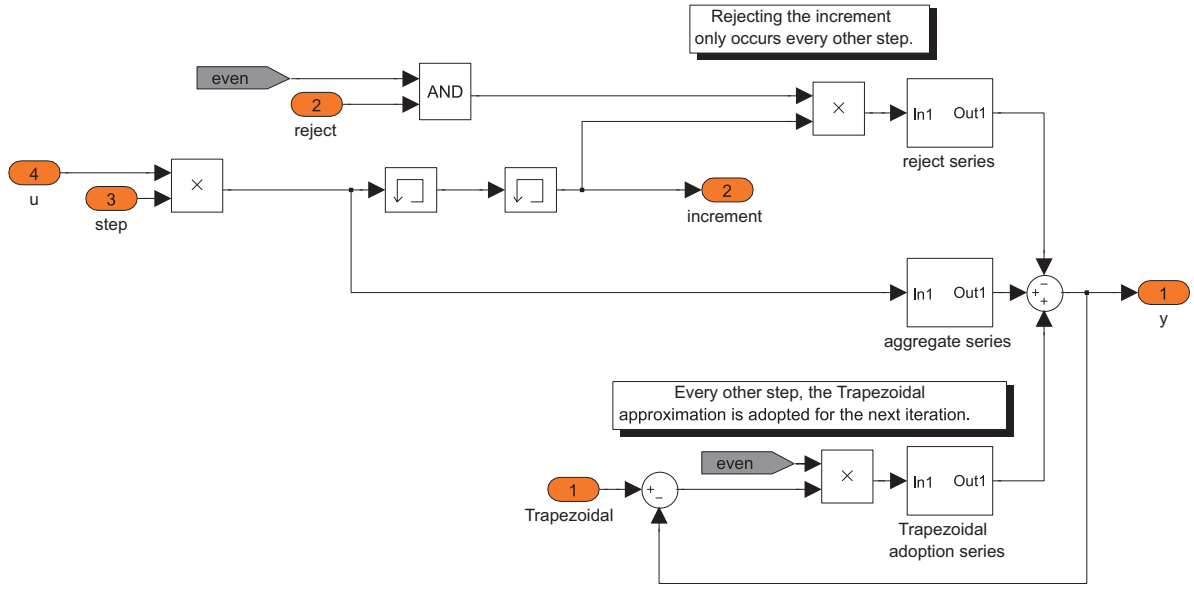


Figure 4. Euler approximation of the Heun numerical integration algorithm.

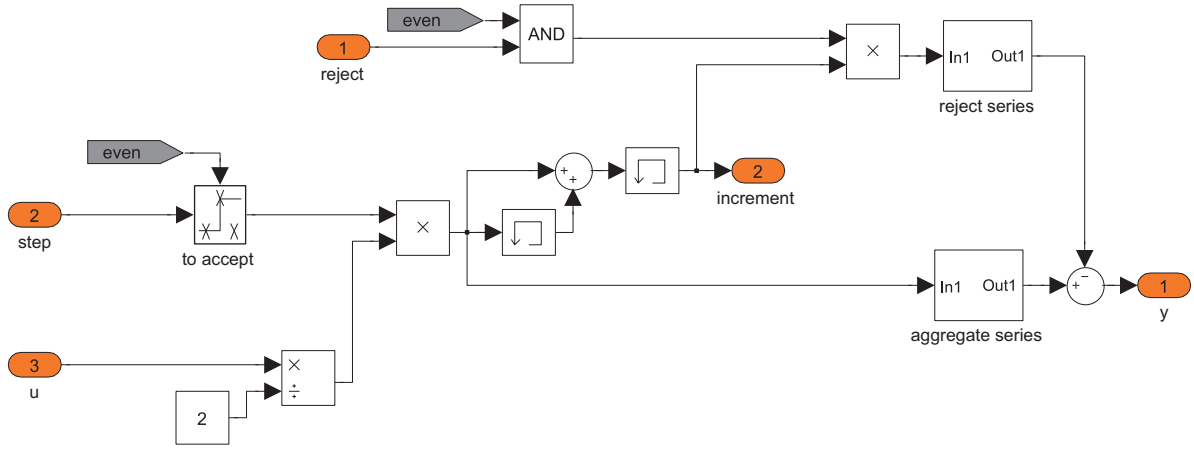


Figure 5. Trapezoidal approximation of the Heun numerical integration algorithm.

In the context of future directions, the formal model of the execution engine represented by the declarative layer proposed in this paper introduces the following benefits over the current approach. The analysis and specific design of a CPS component can now be performed at a high abstraction, without involving an imperative specification or even implementation. Because the computation is described in an abstract manner, it is easier for engineers to understand. In turn, this illustrates how Model-Based Design is not only of value for designing engineered systems, but the principles also apply to Model-Based Design tools themselves, thereby unlocking the potential for an autocatalytic trend

References

[1] Karl J. Åström and Björn Wittenmark. *Computer Controlled Systems: Theory and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[2] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[3] Maria Domenica Di Benedetto and Alberto L. Sangiovanni-Vincentelli, editors. *Hybrid Systems: Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*. Springer-Verlag, March 2001.

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

[5] Ben Denckla and Pieter J. Mosterman. Stream- and state-based semantics of hierarchy in block diagrams. In *Proceedings of the 17th IFAC World Congress*, pages 7955–7960, Seoul, Korea, July 2008.

[6] Nicolas Halbwachs, Pascal Raymond, and Christophe Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau, Germany, August 1991.

[7] Derek J. Hatley and Imtiaz Pirbhai. *Strategies for Real-Time Systems Specification*. Dorset House Publishing Co., New York, New York, 1988.

[8] High Confidence Software and Systems Coordinating Group. High-confidence medical devices: Cyber-physical systems for the 21st century health care. Technical report, Networking and Information Technology Research and Development Program, feb 2009.

[9] Jerry Krasner. Comparing embedded design outcomes with and without model-based design. Technical report, Embedded Market Forecasters, Framingham, MA, October 2010.

[10] Edward A. Lee. What's ahead for embedded software. *Computer*, 33(9):18–26, September 2000.

[11] Joy Lin. Measuring return on investment of model-based design. *EE Times Design*, May 2011.

[12] Nancy Lynch and Bruce Krogh, editors. *Hybrid Systems: Computation and Control*, volume 1790 of *Lecture Notes in Computer Science*. Springer-Verlag, March 2000.

[13] Cleve Moler. Are we there yet? zero crossing and event handling for differential equations. *EE Times*, pages 16–17, 1997. Simulink 2 Special Edition.

[14] Pieter J. Mosterman. HYBRSIM—a modeling and simulation environment for hybrid bond graphs. *Journal of Systems and Control Engineering*, 216(1):35–46, 2002.

[15] Pieter J. Mosterman, Janos Sztipanovits, and Sebastian Engell. Computer automated multi-paradigm modeling in control systems technology. *IEEE Transactions on Control System Technology*, 12(2):223–234, March 2004.

[16] Pieter J. Mosterman and Hans Vangheluwe. Guest editorial: Special issue on computer automated multi-paradigm modeling. *ACM Transactions on Modeling and Computer Simulation*, 12(4):249–255, 2002.

[17] Pieter J. Mosterman and Hans Vangheluwe. Computer automated multi-paradigm modeling: An introduction. *SIMULATION: Transactions of The Society for Modeling and Simulation International*, 80(9):433–450, September 2004.

[18] Pieter J. Mosterman, Justyna Zander, Gregoire Hamon, and Ben Denckla. Towards computational hybrid system semantics for time-based block diagrams. In *Proceedings of the 3rd IFAC Conference on Analysis and Design of Hybrid Systems*, pages 376–385, Zaragoza, Spain, September 2009. plenary paper.

[19] Pieter J. Mosterman, Justyna Zander, Gregoire Hamon, and Ben Denckla. A computational model of time for stiff hybrid systems applied to control synthesis. *Control Engineering Practice*, 19, 2011.

[20] Gabriela Nicolescu and Pieter J. Mosterman, editors. *Model-Based Design for Embedded Systems*. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC Press, Boca Raton, FL, 2009. ISBN: 9781420067842.

[21] Linda R. Petzold. A description of DASSL: A differential/algebraic system solver. Technical Report SAND82-8637, Sandia National Laboratories, Livermore, CA, 1982.

[22] Ingo Sander. *System Modeling and Design Refinement in ForSyDe*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, April 2003.

[23] Semiconductor Industry Association. International technology roadmap for semiconductors: 1999 edition—design. Technical report, Sematech, Austin, TX, 1999.

[24] Simulink®. *Using Simulink®*. MathWorks®, Natick, MA, March 2011.

[25] Michael Sullivan. TACTICAL AIRCRAFT–F/A-22 and JSF acquisition plans and implications for tactical aircraft modernization. Technical Report GAO-05-519T, United States Government Accountability Office, April 2005.

[26] Justyna Zander, Pieter J. Mosterman, Gréegoire Hamon, and Ben Denckla. On the structure of time in computational semantics of a variable-step solver for hybrid behavior analysis. In *Proceedings of the 18th IFAC World Congress*, Milan, Italy, September 2011.

A Model Driven Approach for Requirements Engineering of Industrial Automation Systems

Hongchao Ji[1] Oliver Lenord[1] Dieter Schramm[2]

[1]Bosch Rexroth AG, Germany
{hongchao.ji, oliver.lenord}@boschrexroth.de
[2]Institute for Mechatronics and System Dynamics , University of Duisburg-Essen, Germany
dieter.schramm@uni-due.de

Abstract

Model driven requirements engineering (MDRE) is proposed to deal with the ever-increasing complexity of technical systems in the sense of providing requirement specifications as formal models that are correct, complete, consistent, unambiguous and easy to read and easy to maintain. A critical issue in this area is the lack of a universal and standardized modeling language which covers the whole requirements engineering process from requirement specification, allocation to verification. SysML is being proposed to meet these requirements. In this paper a model driven requirements engineering process for industrial applications in the field of automation systems is described in order to reveal shortcomings in recent modeling tools and modeling languages. Special focus is layed on the requirements definition and the automated verification of the design against the requirements using executable models. Based on the analysis a new profile of the Unified Modeling Language (UML) called Model Driven Requirements Engineering for Bosch Rexroth (MDRE4BR) is presented which aims to contribute to latest investigations in this field. An application example of a hydrostatic press system is given to illustrate the approach.

Keywords Model Driven Requirements Engineering, Industrial Automation Systems, SysML, Modelica

1. Introduction

Model driven engineering (MDE) has been proven to be capable to cope with complexity in the field of software engineering. A trend in modeling and simulation is to apply the MDE paradigms to technical systems consisting of components in hardware and software. The increasing complexity of technical systems has raised many challenges such as keeping the the design consistent and approving the correctness with respect to the customer requirements. Studies at the Bosch Group have shown that over 50% of field problems were due to insufficient requirements engineering (RE) [6]. The RE accompanies the whole product development process, in which various engineering disciplines are involved. Therefore, a universal and standardized modeling language is required which shares the understanding among engineers from different disciplines. This common language shall enable the building of requirements models, system design models, traceability models as well as verification models containing domain-specific details.

The Systems Modeling Language (SysML) [15] is being proposed by the Object Management Group (OMG) [8] to meet these requirements and has already been evaluated by several researchers [4, 5]. The main drawbacks coincide with the results of the analysis of the engineering process of automation systems in section 2 which can be concluded as follows: first the requirements constructs in SysML are not sufficient for real industrial applications, and second the SysML is not capable to describe continuous-time dynamic models.

In order to contribute to the solution of these shortcomings, a UML profile MDRE4BR is proposed in this paper. It reuses and extends the current requirements constructs in SysML targeting the first issue. Recent works on the integration of SysML and Modelica [7] like ModelicaML [12] have already addressed the second issue and proved its effectiveness [13]. Reusing these improvements the proposed extensions of the MDRE4BR are linked with the ModelicaML to transform the later introduced analytical models into executable Modelica models.

This paper is organized in 6 sections. Section 2 motivates the use of the model driven requirements engineering approach in the field of industrial automation systems. Requirements deriving from the application of this approach to the systems engineering of automation systems are discussed. Section 3 gives a short introduction to the related work on the modeling languages SysML, Modelica and ModelicaML. Section 4 describes the implementation of the MDRE4BR profile in detail. The capabilities of the proposed methodology and the MDRE4BR profile are demonstrated on behalf of an industrial application in section

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zürich, Switzerland.
 Proceedings available at:
http://www.ep.liu.se/ecp/056/
EOOLT 2011 website:
http://www.eoolt.org/2011/

5.The paper is closed with conclusions and an outlook to future work.

2. Requirements Engineering in the Field of Industrial Automation Systems

2.1 Scope

This work is seen from the systems engineer's perspective in the field of industrial automation. In terms of building a solid foundation of the later derived requirements on the modeling languages and the tool support, the engineering process is described at first.

Industrial automation systems are characterized by their ability to process a material or work piece according to a defined procedure to achieve the output of a desired product. The quality of the product shall remain stable even though the boundary conditions as climate, disturbances and material properties may differ in a given range. The process is managed in an automated way in order to meet the quality goals in a reproducible, efficient and reliable way.

The systems engineer is now challenged to design a machine that is capable to run the process in a deterministic way. This task is typically performed within a specific design domain that refers to a field of technical expertise such as: the treatment technology, the mechanical design, the drive system and the control. The treatment is usually considered as the core competence of the original equipment manufacturer (OEM). The same is in general true for the closely related mechanical design.

The drive and control technology is typically provided by suppliers. This is due to the fact that power supply, actuators and controls are available on the market as cost efficient standard components in a high quality. Nevertheless the drive and control system is also strongly coupled to other parts of the automation systems. The proper selection of the components and their integration into the overall structure strongly influence function, performance, robustness and reliability. Leading edge technology is determined by the competence of interdisciplinary system design.

Due to the fact that the requirements specification is the subject matter of contract between customer and contractor the above described context implies that the requirements engineering has to be seen not only from the technical perspective but also needs to consider the contractual situation along the supplier chain. With respect to that it is self-evident that the requirements shall be defined and structured not only according to technical aspects but also according to the contractual situation. The definition of levels of abstraction is an appropriate way to meet these needs. The depicted levels of abstraction in Figure 1 reflect the described supplier chain and major technologies involved and therefore are a reasonable choice on behalf of the automated press system considered in section 5.

In order to deal with the complexity of large systems the design objects are clustered in a system break down structure. The requirements derived on the different levels of abstraction can be referenced in requirement specifications in order to provide the contractual views on subsets of requirements.

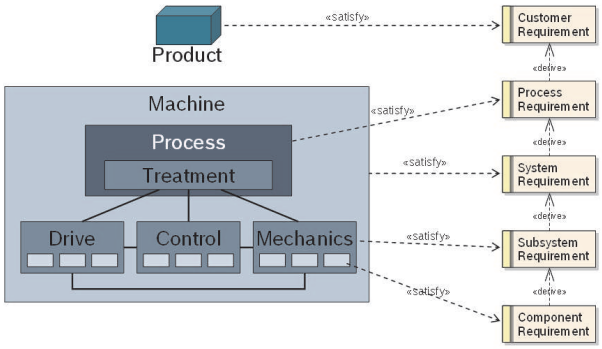


Figure 1. Levels of Abstraction in Requirements and System Design

2.2 The Model Driven Requirements Engineering Approach

Dealing with the above mentioned challenges of managing requirements the MDRE is a very promising and well supported approach. In addition to the UML the SysML defines requirements and several relations between requirements and between requirements and design objects such as: derive, refine and copy. Together with an appropriate package structure the proposed levels of abstraction can be reflected in a SysML requirements model. A classification of requirements can be easily established through stereotypes.

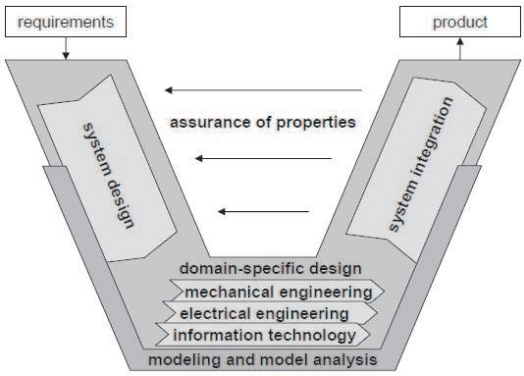


Figure 2. V Model According to VDI 2206 [16]

Furthermore the SysML supports the system design through structural and behavioral diagrams. Relations like trace and satisfy support the traceability between design objects and requirements.

All of this covers a wide range of what is needed in order to refine abstract customer requirements towards a detailed design. The descriptive character of the models are a very appropriate representation of the system design since it allows a certain fuzziness that is unavoidable in the early stage of the system design. Referring to the well known V Model according to the VDI 2206 standard [16] as depicted in Figure 2, it is concluded that the system design phase is well supported by the SysML.

In the domain-specific design, the vertical lower part of the V Model, real world examples, i.e. the hydraulic schematic of a press system, show that the expressiveness of the SysML and its descriptive models are too weak to precisely describe a design on a physical level. Standardized domain-specific schematics, like i.e. the ISO1219, have a much stronger semantics. In practice they have been proven to precisely describe a design such that there is no doubt on how to actually manufacture, assemble and commission the specified pieces. The challenge at this point is to close the gap between the descriptive design model and the physical design model. The SysML concept of blocks and ports is reflected by object-oriented drawing tools for hydraulic and electric circuits like D&C Scheme Editor [1] and ePLAN [9]. The SysML supports domain-specific graphical representations of blocks. The block attributes can be used to define specific technical attributes. Still it is very inconvenient to use common UML/SysML tools for the purpose of drawing domain-specific schematics. Mainly this is due to the fact that no libraries of standard components are available including a large number of symbols and related attributes for all kinds of components and variants. Furthermore common drawing features like title blocks or generating parts lists are not supported. A future engineering tool aiming to support the whole engineering process will have to cope with these requirements.

In the iterative process of refining the descriptive model it is desirable to frequently check the design. According to the V Model (Figure 2) this refers to the depicted iterative step of ”assurance of properties”. Subject matter of this task is to verify the integrated system model against the requirements. Up to now this kind of system verification and validation is a manual procedure that relies on the expertise of the systems engineer. In the sense of a model driven development process it would be beneficial to run this procedure in an automated fashion. Not for all types of requirements this is applicable. In case of analytically verifiable requirements (introduced in 4.2) this could be performed based on a system simulation. To achieve this goal, modeling languages and modeling tools are challenged to answer the following questions:

1. How to achieve an executable analytical model that is capable to approve functional requirements?
2. How to link the analytical model with the design objects of the descriptive model in order to keep it consistent with the evolving system design?
3. How to establish a link to the requirements such that the execution of the analytical model reveals directly which requirements are violated or satisfied?

Because of the interdisciplinary character of automation systems, domain-specific simulation tools like SIMULINK, ADAMS or PSPICE are applicable only in case of requirements that can be evaluated in the related domains separately. In general an integrated interdisciplinary simulation model is required. In the field of multi-physics simulations several modeling methods and tools are available to provide a satisfying integrated system simulation that applies to question number one. A well established and standardized multi-physics modeling language is Modelica which is considered in the following for further investigation.

The other two questions have been addressed by several work on modeling languages that are discussed in the following section.

3. Background and Related Work

3.1 Introduction to SysML and Modelica

SysML is a general purpose language used in the field of systems engineering. It is defined as a UML profile which reuses subsets of UML constructs and extends them with some additional modeling elements. The SysML is capable to capture the textual requirements and to allocate them with the design models and test cases. However due to loosely defined executable semantics SysML is not capable to model physical systems in an executable way. In contrast to that, Modelica is an object oriented and equation based modeling language for multi-domain physical systems. Graphical modeling is supported by the object diagram which offers an intuitive way to describe power transmission through acausal connections as well as directed signal flows. The strong semantics allow the generation of executable models of continuous as well as discrete systems. Object oriented language constructs enable the efficient reuse of models and the design of comprehensive and easy to use model libraries. A language which integrates the descriptive modeling power of SysML and the formal executable simulation power of Modelica seems to be a promising approach for the requirements engineering in industrial automation systems. An overview on latest research activities in this field is given in the following.

3.2 Requirements Specification

Several researchers have already used and extended the SysML as requirement specification language. In Dubois et. al. [2] a requirement meta model to enforce the traceability concept in SysML in the automotive domain is presented, in which a traceability model connects three independent flows (requirement model, solution model and verification and validation (V&V) model). However the traceability of the V& V model to the other models is not clearly defined.

The vVDR methodology [14] addresses the virtual verification of systems requirements. The other contribution of this work is an approach to formalize requirements such that they can be quantified and tested effectively. The vVDR approach is considered in the later case study.

3.3 Integration of SysML and Modelica

Main target of the ModelicaML language by Schamai et. al. [12, 13] is to provide an executable graphical language for hybrid modeling and simulation while enabling an effective way to create, read and maintain Modelica models. In contrast to the original ModelicaML [11], the new ModelicaML is implemented as a pure UML profile with no direct dependency on the SysML. Instead, the new ModelicaML uses a subset of UML, extends the UML meta model (us-

ing the UML profiling mechanism) with new constructs in order to introduce missing Modelica concepts and to reuse several SysML concepts like the requirement constructs. ModelicaML is defined as a graphical notation that facilitates different views (e.g. composition, inheritance and behavior) on Modelica models. Those graphical notations can be translated into Modelica code and simulated with any Modelica tool.

The SysML-Modelica Transformation Specification is another research activity on the integration of SysML and Modelica. The main target of this work is to specify a standardized bi-directional transformation as foundation for later implementations that support the unambiguous, efficient and automatic transfer of model information between SysML and Modelica. In order to define a formal transformation between SysML and Modelica, an extension to SysML called SysML4Modelica profile, that represents most common Modelica constructs, is developed. In this profile, each Modelica construct will be checked whether there is already an equivalent construct in SysML from a semantic point of view. When the corresponding construct does not exist, a stereotype will be created to extend the SysML language accordingly. A mapping between the SysML4Modelica profile and the Modelica meta model is specified which enables a round-trip transformation from SysML to Modelica and vice versa [10].

The SysML-Modelica Transformation addresses the need of SysML users to define analytical relations in a mathematical form in addition to the existing descriptive constructs. The SysML4Modelica profile is simply applied to a selected sub part of the system model which is constraint through mathematical relations to be analytically resolved. This modeling and simulation approach is pretty straight forward in case of rather simple relations and measures between parameters, referred to as parametrics. In case of advanced simulation models this approach loses its practical relevance. The expressiveness of the Modelica object diagram in conjunction with the large number of easily reusable models in Modelica libraries is indispensable for even small system models.

The general approach of separating the descriptive model from the analytical model is useful. The design process is an incremental procedure which implies that not all parts are defined on the same level of detail at a time. Furthermore it is common that different parts of the model shall be simulated under different circumstances. This aspect is not addressed in the ModelicaML profile which considers the design and the simulation model as one consistent instance of the system.

4. The MDRE4BR Profile

The MDRE4BR profile is structured in three parts, namely the requirements definition package which classifies different types of requirements; the requirements traceability package, which details different traceability links related to requirements allocation; the requirements verification package, which covers the verification of the design against the requirements by means of an executable model. These packages are defined in detail right after the description of the underlying concept.

4.1 Overall Concept

The *requirements model* contains all requirements according to the classification below.

The *design model* is a descriptive model to describe structure and behavior of the system. Every design object is uniquely defined in order to describe a complete and consistent model of the system.

The *analytical model* is an executable model. This requires a much more formal description of the behavior. This is performed through the mathematical expressiveness of Modelica. The analytical model consists of one or multiple mathematical models. Each mathematical model describes one aspect of the overall context. In contrast to the design model this may lead to the case that the very same component is represented through different mathematical models. Referring to the press system in section 5 this would refer to the case that i.e. the pressure supply is considered under two different aspects. First in the analysis of the stability of the control circuit, second to approve the proper layout of the suction pipe of the hydraulic aggregate. In the first case a simplified representation as ideal constant pressure source is adequate, while in the other case the dimensions of pump, suction pipe and tank have to be considered in the mathematical description.

If the simulation models are decoupled from each other redundant data is produced. Sooner or later this will lead to inconsistencies that are hard to detect and have a strong impact on the quality of the design. To prevent this situation a relation between the mathematical models and the design objects is required. In order to realize meaningful relations the design object has to reflect the attributes of all related models. In the profile this is easily considered by referencing the attributes of the design objects from the Modelica parameters and constants of the analytical model. In the case study these relations have been established manually which is time consuming and error prone. At this point an improved tooling is needed that supports the binding of related objects. A promising approach would be to provide a component library of predefined objects that contain the relevant attributes of the design object as well as references to a number of meaningful mathematical models of different level of detail.

The mathematical model can be transformed into an executable model. This transformation is performed through compilation of the Modelica model and binding with the simulation run-time. The simulation run-time instantiates and runs the algorithm that is needed to solve the differential-algebraic equations. This procedure requires a definition of the simulation settings such as start time, end time, tolerances and output step size. Furthermore the initial state of the equation system needs to be defined. In the MDRE4BR profile these informations are described through the so-called test scenario. The test scenario defines all these boundary conditions under which a mathematical model shall be executed.

In order to use an executable model for the purpose of requirements verification the test scenario can be extended with so-called test cases. The test case is a mathematical model that evaluates a specific measure by comparing the simulation results with a desired output. Further details are given in section 4.2.

4.2 Requirements Definition

A general definition of the MDRE4BR requirement (see Figure 3) extends the standard requirement definition in SysML with additional attributes which are described as follows:

- The *id* property is the unique identifier of the requirement (as defined in SysML);
- The *text* property is the textual description of the requirement (as defined in SysML);
- The *abstrationLevel* property defines the different abstractions level of requirements;
- The *priority* property defines the importance of the requirement such as mandatory or optional;
- The *version* property records the change history of the requirement;
- The *satisfyState* identifies whether one requirement has been fulfilled by a model object or not;
- The *verifyState* property identifies the verification state of the requirement which can be pending, passed, failed or not verifiable according to the verification results in the test cases.

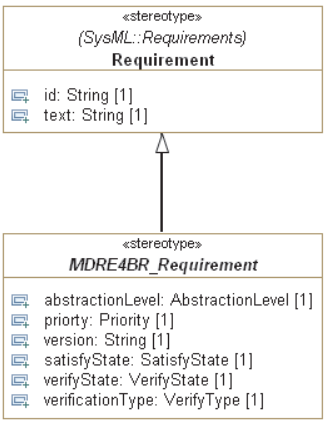


Figure 3. General Requirement Definition

The requirements classification is another additional concept in the requirements definition. The requirements can be classified as analytically-verifiable and not-analytically-verifiable requirements. Analytically verifiable refers to requirements which can be verified through evaluation of the system model against formally described criteria. In the context of this paper this is limited to the case that the behavior of the system is described by an executable simulation model which is verified through test scenarios and test cases. In contrast to that not-analytically-verifiable requirements refer to requirements which require additional knowledge or judgment to decide whether or not the design satisfies the requirement. In this case all related aspects of the design need to be easily accessible to the decision making system engineer. For this purpose the trace or satisfy relation between requirement and design object shall be used.

The following classification, based on the taxonomy proposed in [3], is supported by the MDRE4BR profile through stereotypes.

- A *functional requirement* is a requirement that should produce an expected reaction to a given stimuli.
- A *performance* is a requirement to check whether a system variable such as timing, speed, volume or throughput is in a desired range.
- A *structural requirement* is a requirement which describes the structural demand of the stakeholder.
- All the other types of not-analytically-verifiable requirements can be modeled with the *other requirement* stereotype.

This classification allows distinguishing requirements according to how they shall be processed in the verification phase. Requirements that shall be verified automatically are identified and described through additional attributes such that they can be processed in the desired fashion.

4.3 Requirements Traceability

Requirements traceability is used to specify the relationships from and to requirements. At first this defines the relations needed to express an appropriate requirements break down structure considering different levels of abstraction and other dependencies among the requirements. Furthermore relations are defined that are needed to trace the requirements from and to the objects of the design model and/or the analytical model.

The current available traceability links in SysML are defined as follows:

- The copy, containment and deriveReqt are defined to model the traceability among requirements;
- Traceability between requirements and design objects are supported by satisfy, trace and refine;
- The relationship between requirements and test cases are defined with verify.

These traceability links defined in the SysML cover the relationships needed in the considered industrial field of application. The traceability between requirements and design objects supports the systems engineer to systematically complete the system model. This is done by incrementally building up the design model through additional design objects that are needed to satisfy a particular set of requirements. Finally all design objects have been checked against all requirements of the related level of abstraction such that all applicable satisfy relations have been established. In this process it may apply that a design object can not be sufficiently specified through the existing requirements. This revealed incompleteness of the requirements is resolved by the iterative refinement of the requirements.

In the described context of the design phase the traceability links are very useful but the application for the desired automated verification is limited due to the lack of formal executable semantics and syntax. The proposed extended traceability link explicitly defines the information to be transferred via this traceability links in order to achieve an automated verification. This is done by allocating a performance requirement with a performance test case, using the MDRE4BRverify relation. The performance variable defined in the performance requirement is associated with the same in the test case as depicted in Figure 4.

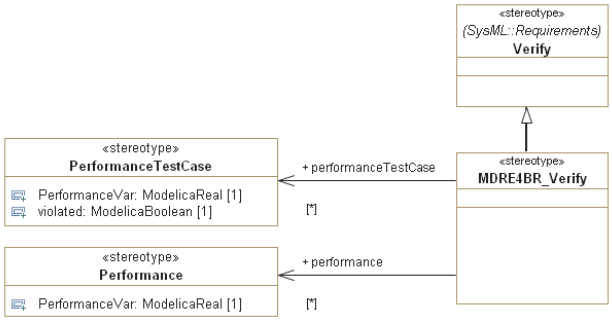


Figure 4. Verify Relation in MDRE4BR Profile

4.4 Requirements Verification

The goal of the requirements verification in the MDRE process is to verify the design against the requirements in an automated and reproduceable way. In the MDRE4BR profile this is achieved by combining the above described extended semantics of the verify relation with the concept of violation monitors and variable binding described in the vVDR methodology [14].

In the MDRE4BR profile, different types of test cases for different kinds of requirements are defined as stereotypes, such as, the performance test case and the functional requirement test case which are derived from the SysML meta class TestCase as shown in Figure 5. The violation monitor is modeled as part of the performance test case or functional test case in order to evaluate the requirement.

A new stereotype called test scenario is defined, which contains multiple test cases referring to at least one mathematical model. The requirements verification can be performed by executing the test scenarios and related test cases defined in the analytical model. The relations among these elements are depicted in Figure 6.

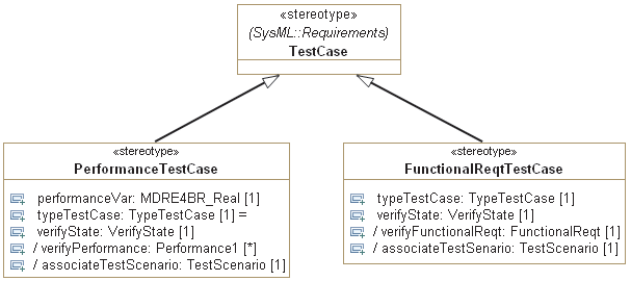


Figure 5. The Performance and Functional Test Cases

In the verification process, the user defines the different test scenarios by referencing the selected test cases and the mathematical models needed to produce the simulation results. The variable binding is used in the test scenario to establish the links between test cases and mathematical model as describe in the vVDR method [14] .

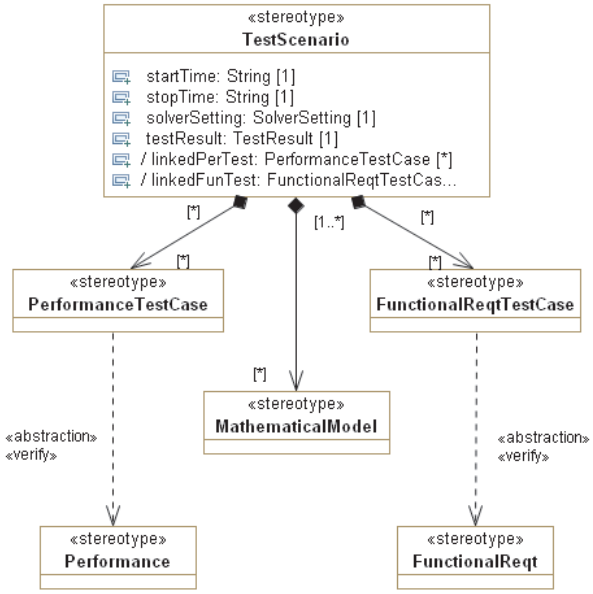


Figure 6. The Relations in Verification Package

5. Case Study: Hydrostatic Press System

In this section, a hydrostatic press system (Figure 7)is used to illustrate the model driven requirements engineering approach by using the MDRE4BR profile. The main steps of the procedure can be summarized as follows:

Figure 7. Hydrostatic Press

1. Capture the customer requirements as stereotyped requirements according to the proposed classification.
2. Derive requirements on a lower level of abstraction using the «deriveReq» relation to the higher level requirements.
3. Create descriptive structural and behavioral design models.

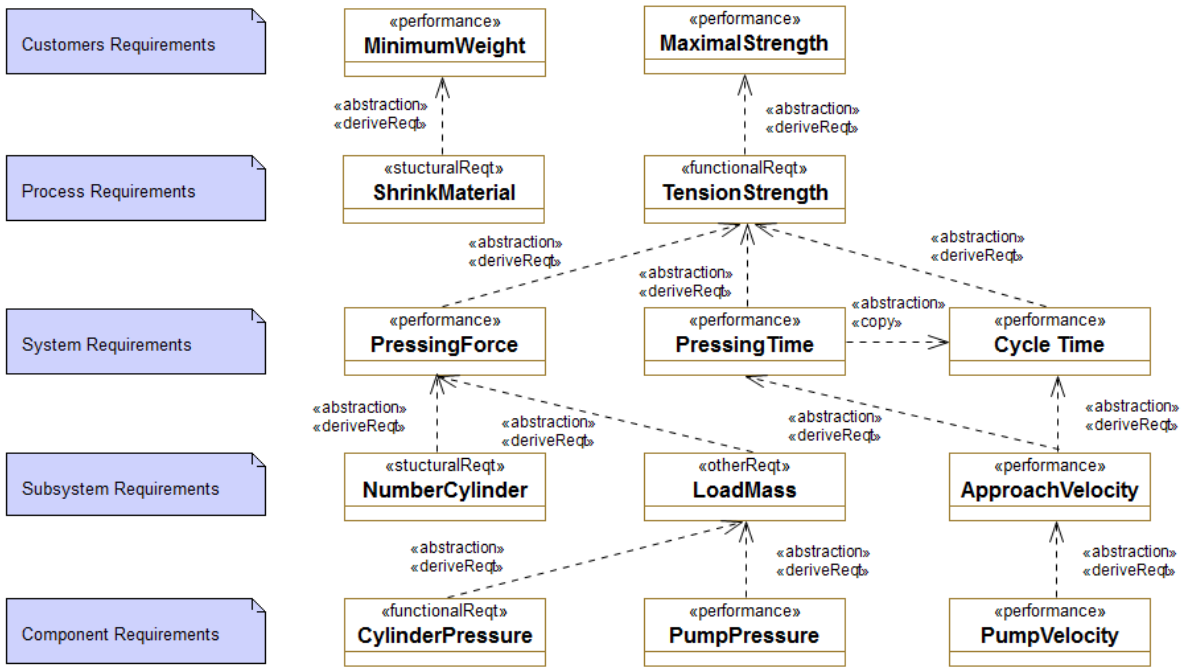


Figure 8. Derive Requirements from Different Abstraction Levels

4. Establish «satisfy» relations between requirements and design objects.
5. Define test cases for those design objects that need to satisfy verifiable requirements.
6. Establish «verify» relations in order to bind the result of the test case to the verifyState of the related requirement and to retrieve the measures from the requirement and apply them to the related test case variable.
7. Create a mathematical model of the related design objects on the considered level of abstraction.
8. Define test scenarios which execute the mathematical models under defined boundary conditions.
9. Integrate those test cases in the test scenarios which are applicable.
10. Run a model verification that executes all related test scenarios and updates the verifyStatus of all related verifiable requirements.

This procedure is performed in an incremental fashion until the design has reached a satisfying level of detail that is also reflected in the analytical models. After the final verification all verifiable requirements shall be approved through the related test scenarios.

In the case study a press system is considered which shall be used to demonstrate the procedure on behalf of a real industrial application.

5.1 Requirements Capture and Refinement

The hydrostatic press shall be considered in the context of the OEM-supplier relation as it applies to a typical Bosch Rexroth engineering project. In this case the high level requirements have already been refined to the subsystem level. Figure 8 shows some exemplary requirements reflecting different levels of abstraction.

Because of the limited space in the diagram, the descriptions of the requirements are to be found in Table 1 and 2. In the following the focus is on the the requirements from the abstraction level system requirements and below. As shown in Figure 8, these system requirements are derived from the customers and process requirements and can be classified according to the subtypes of the requirements definition in the MDRE4BR profile. The beginning letter of the ID of the requirement refers to the type of the requirement.

Name	Description
MinimumWeight	Steal structure shall have minimum weight.
MaximalStrength	Steal structure shall have maximum strength.
ShrinkMaterial	In-mold hardening shall not shrink the material below manufacturing tolerance of 0.1mm.
TensionStrength	In-mold hardening shall increase the tensile strength to $1000N/mm^2$

Table 1. The Customers and Process Requirements

5.2 Structural and Behavioral Design

In Figure 9 the overall design on the subsystem level is depicted using the SysML internal block diagram. This rough design refers to an early phase of the design process in which the subsystems and the interfaces are in the main focus.

The behavior of the system is described through the SysML activity diagram in Figure 10 according to the working process shown in Table 3. The conditional state-

ID	Description
P1	The pressing force of cylinder shall be $180000kN$.
P2	The pressing time shall shall be $8s$.
P3	The cycle time shall not exceed $27s$.
P4	The approach velocity of the load shall be $0.5m/s$.
P5	The pressure drop across the pump shall not exceed 340 bar.
P6	The driving speed of the pump is 3000 rpm in approach phase.
F1	The cylinder pressure can not exceed 120 bar.
S1	The number of driving cylinders is 2.
O1	The moving mass shall be 40t.

Table 2. The System Requirements List

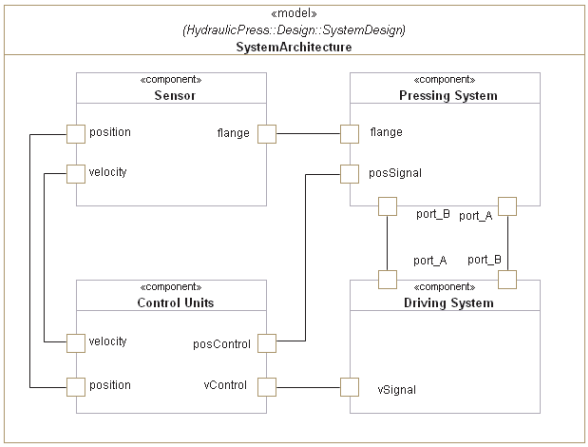


Figure 9. The System Architecture

ments precisely describe the transitions that refer to different controller modes. In a later step these subsystems need to be refined on the component level in order to specify how the system shall be physically built. In the opinion of the authors this kind of detailed design is done best based on a domain-specific language as i.e. the ISO1219 in case of hydraulic circuits. In consequence a future engineering tool should integrate an object library of standard components with the graphical representation according the ISO1219. Additional attributes as i.e. material number and position number are needed to reflect the functionality of drawing schematics and generating parts lists as known from engineering tools like D&C Scheme Editor or ePLAN.

State	Entry Condition	E-Drive Speed	Cylinder Velocity
Apporach	$x = 0$	3000	0.50
Brake	$x > 1200$	3000	0.015
Press	$x > 1240$	1500	0.007
Keep Position	$x = 1300$	controlled	0
Slow Return	$\Delta t = 8$	controlled	0.01
Return	$x < 1290$	3000	0.2

Table 3. Desired Working Process

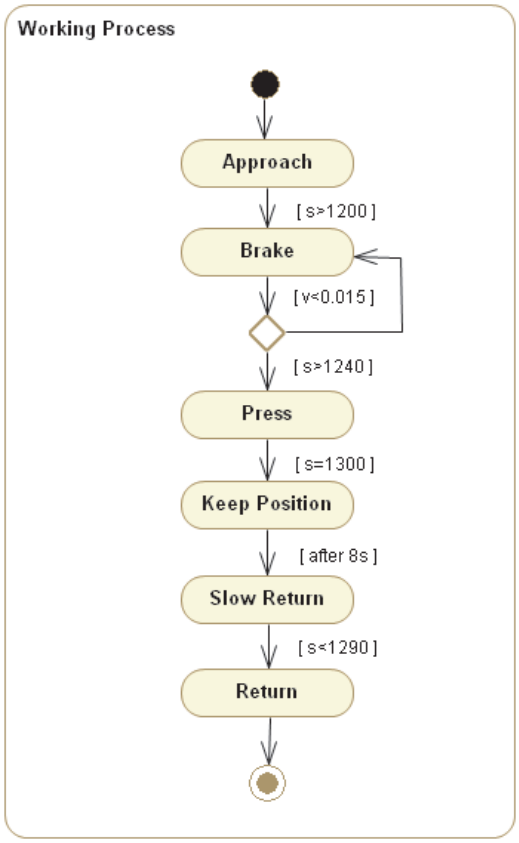


Figure 10. The Behavior of the System

Based on the captured requirements, shown in Figure 8, additional satisfy relationships from the design objects to the requirements have been established. In 11 this has been done exemplarily with the requirement P4: ApproachVelocity from Table 2. A much more expressive representation as relation matrix of design objects versus requirements is not shown due to the limited space. In the relation matrix, the SatisfyState of each requirement is checked to see whether all the requirements are considered in the design. With help of the relation matrix, the coverage of the requirements can be defined.

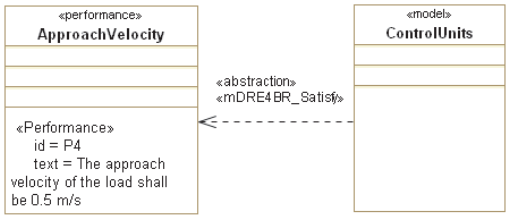


Figure 11. Building Satisfy Relation

5.3 Requirements Verification

The verification of the system behavior shall be performed by means of simulations based on the analytical model. This has been done exemplarily with the requirement P4: ApproachVelocity. This has been linked to a performance test case TestCase ApprochVelocity which is contained by the TestScenario1 as depicted in Figure 12. Moreover the

TestScenario1 contain a mathematical model which is used to stimulate the associated test cases. A performance test case is modeled further by a Modelica state machine as violation monitor to check the performance requirement ApproachVelocity (Figure 13).

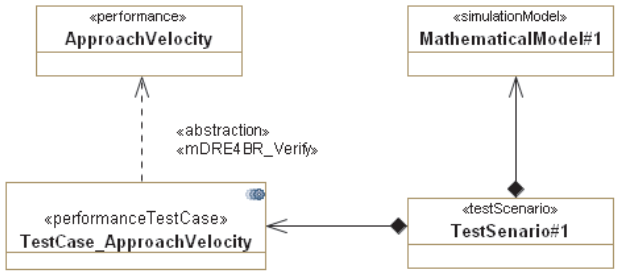


Figure 12. Verification Model

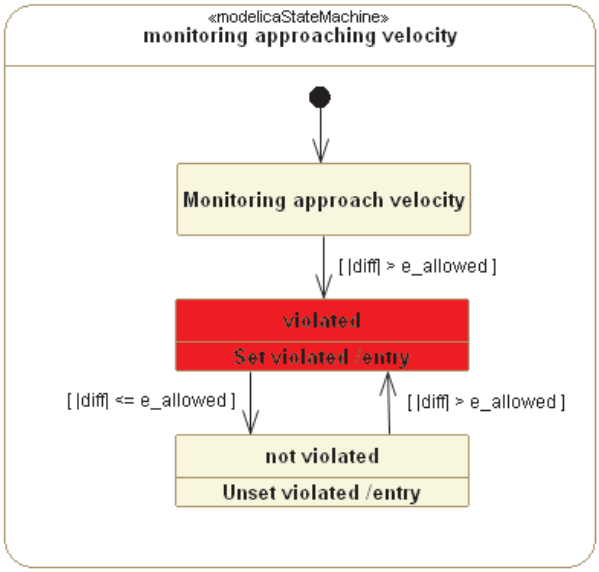


Figure 13. Modeling a Test Case

The mathematical model is defined by a Modelica model that has been build in Dymola. It consists of the four main parts: control unit, pressing system, driving system and sensor. The instances of the subsystem models are traced to the related objects of the design model. Since the focus of this paper is to present the model driven requirements engineering approach, the system simulation model will not be introduced in more detail here.

Currently, the binding of the corresponding variables between test cases and simulation models is very inconvenient. The future work shall address this issue in order to integrate the MDRE4BR profile with the ModelicaML code generator such that an automatic binding can be realized on the code level.

5.4 Verification Results

The simulation results referring to the requirement P4 is presented in Figure 15. It can be seen that, the approach velocity exceeds the desired velocity at the beginning. There-

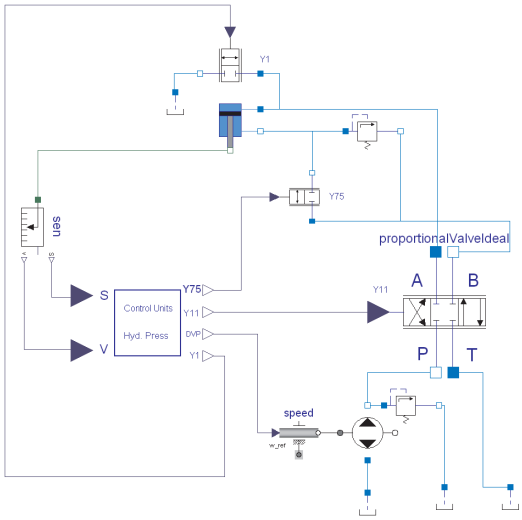


Figure 14. Object Diagram of Hydrostatic Press

fore this requirement is violated in this test scenario. The verifyState of the requirement is set to failed accordingly.

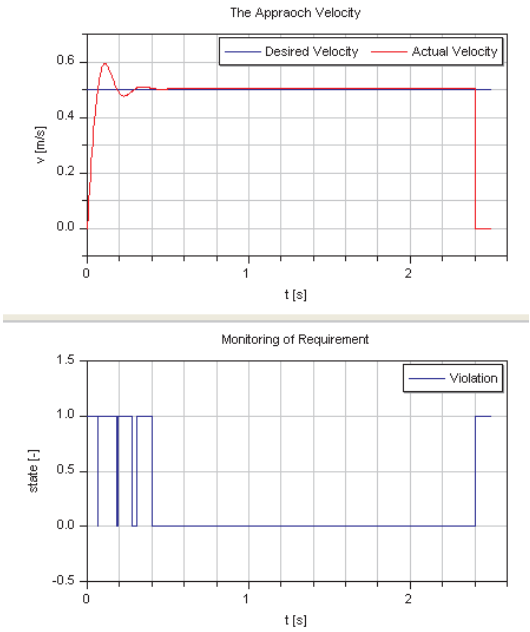


Figure 15. Verification of Requirement

In the same manner, all the other verifiable requirements can be verified. The verification results are summarized in the following figure (Figure 16).

6. Conclusion and Outlook

In this paper the model driven requirements engineering approach has been analyzed in the systems engineering context of industrial automation systems. Recent approaches integrating SysML and Modelica have been investigated according to their capability to describe a holistic model that covers the requirements engineering, the system design and the model verification and validation through simulation. Extensions and modifications have been presented as MDRE4BR profile. The concept has

ID	Description	?
P1	The maximum force of cylinder can not exceed180 kN.	
P2	The pressing time shall not exceed 8 s.	
P3	The cycle time shall not exceed 27s	
P4	The approaching velocity of the load shall be 0.5 m/s.	
P5	The pressure drop across the pump shall not exceed 340 bar.	
P6	The driving speed of the pump is 3000 rpm in approach phase.	
F1	The pressing pressure can not exceed 120 bar.	
S1	The number of driving cylinder is 2.	
O1	The moving mass shall be 40t.	

Figure 16. Summary of Verification Results

been demonstrated by a case study of a typical engineering project.

In the future work the tool support shall be improved. This refers to the establishing of the relations between design model and analytical model, as well as the relations between test case and mathematical model.

Acknowledgments

This work is funded by the Bosch Rexroth AG and the German Federal Ministry of Education and Research (BMBF) in the ITEA2 OPENPROD project.

References

[1] Bosch Rexroth AG. D&C Scheme Editor Manual. Technical report, Bosch Rexroth AG, 2010.

[2] Hubert Dubois, Marie-Agnès Peraldi-Frati, and Fadoi Lakhal. A Model for Requirements Traceability in a Heterogeneous Model-Based Design Process: Application to Automotive Embedded Systems. In *15th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2010, Oxford, United Kingdom, 22-26 March 2010*, pages 233–242, 2010.

[3] Martin Glinz. On Non-Functional Requirements. In *15th IEEE International Requirements Engineering Conference, RE 2007, October 15-19th, 2007, New Delhi, India*, pages 21–26, 2007.

[4] Eric Herzog and Asmus Pandikow. SysML - an Assessment. In *Proceedings of the 15th INCOSE International Symposium*, 2005.

[5] Marcos Vinicius Linhares, Alexandre Jose da Silva, and Romulo Silva de Oliveira. Empirical Evaluation of SysML through the Modeling of an Industrial Automation Unit. In *Proceedings of 11th IEEE International Conference on Emerging Technologies and Factory Automation*, 2006.

[6] Jürgen Lüttin. Bosch Requirements Engineering Framework - Overview. Technical report, Robert Bosch GmbH, 2010.

[7] `http://www.modelica.org`.

[8] `http://www.omg.org`.

[9] `http://www.eplan.de`.

[10] Christiaan J.J. Paredis, Yves Bernard, Roger M. Burkhart, Hans-Peter de Koning, Sanford Friedenthal, Peter Fritzson, Nicolas F. Rouquette, and Wladimir Schamai. An Overview of the SysML-Modelica Transformation Specification. In *2010 INCOSE International Symposium*, July 2010.

[11] Adrian Pop, David Akhvlediani, and Peter Fritzson. Towards Unified System Modeling with the ModelicaML UML Profile. In *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT'07)*, pages 13–24, 2007.

[12] Wladimir Schamai. Modelica Modeling Language (ModelicaML). Technical report, EADS Innovation Works, Germany, 2009.

[13] Wladimir Schamai, Peter Fritzson, Chris Paredis, and Adrian Pop. Towards Unified System Modeling and Simulation with ModelicaML: Modeling of Executable Behavior Using Graphical Notations. In *Proceedings of the 7th International Modelica Conference, Como, Italy, 20-22 September 2009*, number 43 in Linköping Electronic Conference Proceedings, pages 612–621. Linköping University Electronic Press, Linköpings universitet, December 2009.

[14] Wladimir Schamai, Philipp Helle, Peter Fritzson, and Christiaan J. J. Paredis. Virtual Verification of System Designs against System Requirements. In *Models in Software Engineering - Workshops and Symposia at MODELS 2010, Oslo, Norway, October 2-8, 2010, Reports and Revised Selected Papers*, pages 75–89, 2010.

[15] `http://www.omgsysml.org`.

[16] VDI. Design Methodology for Mechatronic Systems (VDI 2206). Technical report, VDI, 2004.

A Generic FMU Interface for Modelica

Wuzhu Chen[1] Michaela Huhn[1] Peter Fritzson[2]

[1]Department of Informatics, Clausthal University of Technology, Germany,
{wuzhu.chen,michaela.huhn}@tu-clausthal.de
[2]Department of Computer and Information Science, Linköping University,Sweden, peter.fritzson@liu.se

Abstract

This paper discusses technical issues and implementation of a generic interface to import a Functional Mock-up Unit (FMU) into Modelica simulators, specifically the OpenModelica environment. Whereas other approaches for importing the FMUs rely on functionality specific to the simulator environment, this approach tries to provide a generic Modelica interface for embedding an FMU to be imported into a Modelica model. In this way any FMU conforming to the Functional Mock-up Interface (FMI) for Model Exchange v1.0 Specification for model exchange from MODELISAR can be imported into *any* Modelica simulator. When importing an FMU into a model, the resulting Modelica model can be used just like any pure Modelica models. Hence, a better reusability and interoperability for both sides, namely the external models provided via FMI and the Modelica environment, are achieved.

Keywords OpenModelica, Functional Mock-up Interface(FMI), Functional Mock-up Unit (FMU) import, Modelica code generation

1. Introduction

1.1 Modelica Models and Modelica Simulators

The Modelica language is primarily designed for modeling and simulating complex hybrid dynamic systems, so its features intend to reflect the structure, the inherent behavior, the hierarchy and the heterogeneity of those systems. Briefly speaking, Modelica language is object-oriented, equation-based, component-based and capable of modeling multi-domain problems. As a result of the modularity concepts, models described by the Modelica language can be almost freely exchanged between different Modelica tools only with some minimal restrictions, for instance, differences in the version of Modelica language or the version of the Modelica Standard Library being used. All hierarchical information as well as the behavior of models will be explicitly transferred during such exchanges, since they are delineated in the standardized Modelica language.

However, simulating complex and heterogeneous systems bears several needs for integrating models that originate from other sources: For some subsystems elaborated or optimized models may have been implemented using other domain-specific modeling languages or a general-purpose programming language.

Another reason that applies even if Modelica is used for subsystems is the protection of intellectual property. In many situations it is a business case to enable using the model for simulation without externalizing the model's structure and behavior. Thus, tool support for integrating individual sub-models into a Modelica simulation environment as well as co-simulation that facilitates the coupling of various simulators is a basic requirement for real-world multi-domain modeling.

A Modelica simulator typically contains a compiler and a run-time module. The compiler translates Modelica models into an appropriate programming language (C, C++, etc.), which will be further translated into executables by their own standard compilers. The run-time module then executes the outcomes from the compiler, handles the configuration of particular simulation runs, solves the equation system, depicts and stores the results.

1.2 Functional Mock-up Interface (FMI)

The *FMI for Model Exchange v1.0* [3] has been specified and released by the MODELISAR[1] consortium. This specification provides a series of open, standardized interfaces, through which the instantiation, initialization, execution of an individual executable (or C code) model representation called *Functional Mock-up Unit* (FMU) can be performed within a simulation environment. In order to use the FMI from both sides, the FMU and the simulator have to implement the interface functions as defined in the FMI specification. The FMU contains a concrete mathematical model described by differential, algebraic and discrete equations with possible events of a dynamic physical system transformed into explicit form and represented as C code or machine code. The idea of the FMI standard is, that an FMU can be generated from any modeling environment and im-

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zürich, Switzerland.
 Proceedings available at:
http://www.ep.liu.se/ecp/056/
EOOLT 2011 website:
http://www.eoolt.org/2011/

[1] MODELISAR is a ITEA2 project focusing on the interoperability between Modelica and AUTOSAR to support Vehicle Functional Mock-up

ported into any simulation environment whenever the compliance with FMI specification for model exchange is fulfilled.

Technically, a generated FMU will be distributed as a compressed .fmu file basically consisting of two parts. The first part is the model interface functions according to the FMI specification. They can be given either in C source code or in binary forms (e.g. in the form of .dll or .so files) to protect the model developers' intellectual property. The second part is the model description XML file, conforming to the schema defined in the FMI specification. The XML file contains all information about the model variables and some other optional information. Additional parts can be added and compressed into the FMU, as for instance the documentation and the icon of the model. Compared to the exchange of Modelica models, exchanging of FMUs is at a lower abstraction level and more target-oriented, and consequently less flexible due to the export and import procedures and the conversion of acausal Modelica models to causal explicit form.

1.3 State of the Art for FMU Import

Currently, some Modelica simulators already support FMU import, for example SimulationX ®, Dymola ®, and the open source tool JModelica.org[2], and some others. Although the FMI interface is vendor neutral, the implementations of import of the FMU realized by the above mentioned simulator tools are not. Either the import of FMU is implemented on the basis of some specific programming language [1], Python in this case, or the Modelica language is internally extended [4] because the Modelica language lacks support for the full range of functionalities specified in FMI. However, since each tool vendor offers a proprietary implementation of FMU import, we are actually keeping reinventing the wheel in the sense of multiple implementations of FMU import. Moreover, there still exists some additional difficulties in the FMI specification of FMU import that hamper the proliferation of the FMI standard in practice, for example the lack of support of multiple instances of FMU models generated by some simulation tool, the lack of coupling of imported FMUs with Modelica models, to name just some of them.

1.4 Motivation for a Generic FMU Interface to Modelica

As discussed before, the benefits of a standardized interface for model exchange are widely agreed and have led to the FMI specification by MODELISAR. But furthermore there is also a need for a generic and uniform implementation of the import functionality for Modelica tools. A holistic modeling of a heterogeneous system may incorporate the combined simulation of Modelica models and a number of imported FMUs within several Modelica simulators. In this way, the systematic behavior of the total system can be analyzed without revealing the modeling know-how embedded of the FMU components. A key role to generalize the model exchange and to glue all these benefits together is a tool-independent, uniform interface from FMUs to Modelica models, which can be also distributed along with the FMUs for potential use by Modelica modelers. Thus, the model exchange facility indeed crosses the boundary of simulators in the Modelica context.

The rest of the paper is organized as follows: In Section 2, the implementation of the generic interface are presented in detail. In section 3, a case study of a hybrid dynamic system is investigated and the results are analyzed in OpenModelica environment. At the end in Section 4, the conclusions of this FMU import approach are given.

2. Implementation of the Interface

2.1 Overview of the Implementation

As already mentioned in the introduction Section 1, we aim at a generic interface for importing FMUs into Modelica models in a way that any FMUs can be directly evaluated and simulated in any Modelica simulator. Thus the key issue is to decouple the FMU import process from specific Modelica simulator implementations. Most of the FMU interface functions will be mapped to Modelica external functions bijectively. Some extra Modelica helper classes and functions are also implemented to perform additional tasks, for example type conversion, special construct instantiation, message printing and so on. The imported FMU itself is represented as a Modelica class through the extension of an external object with variable information from parsing the model description XML file. In this way, multiple instances of this FMU can be easily declared and connected with models that are described in Modelica.

The import process contains following key steps

- To extract the archived .fmu file in some specified folder
- To parse the model description file for model information
- To load the library file or include the source file if provided
- To integrate the functions and the variable information into a Modelica block

Summing up these aspects, the concept of the implementation of FMU import is illustrated in Figure 1.

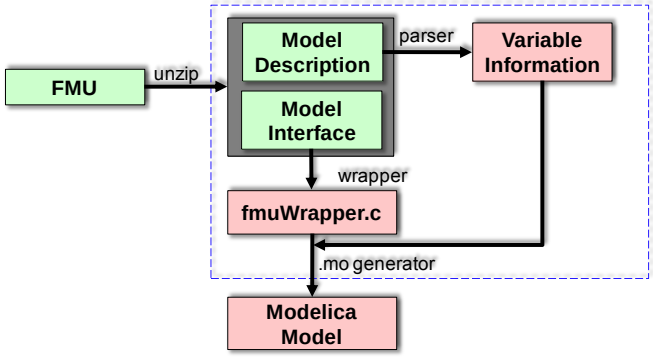


Figure 1. An overview of FMU import.

[2] JModelica.org is an open source Modelica simulation environment tailored for optimization problems. For more information please refer to http://www.jmodelica.org

2.2 Details of the Prototype Implementation

A prototype of the generic FMU interface for Modelica has been implemented and tested in the OpenModelica environment. The details of the implementation are going to be discussed in this section.

Even in the open-source domain, we find numerous possibilities for the decompression of a .fmu archive file, e.g. *zlib*, *gzip*, *PowerArchiver*, *7-zip*, etc. Since this step is rather trivial compared to the others, it will not be discussed further.

After having decompressed the archive file, the model description XML file and the model interface implementation are stored in a specified directory, e.g. in OpenModelica under Microsoft Windows®the directory will be "<OPENMODELICAHOME> \tmp\fmu". Information about model variables and other additional information can be extracted from the XML file by an appropriate XML parser, we will use the parser in the *libxml2* library provided by the OpenModelica environment.

For the current prototype, the model interface implementation is supposed to be a shared dynamic link library (DLL). By successfully parsing the XML and loading the DLL, the handle of the DLL, the addresses of the exported interface functions and the information data from the XML file are stored in a data structure called *FMI*, which is defined in a wrapper header file. The instantiation of this structure in Modelica is achieved by a class, which extends the Modelica *External Object* as shown below.

```
class fmuFunctions "List of interface functions"
  extends ExternalObject;
  function constructor
    input String dllPath;
    output fmuFunctions fmufun;
    external "C" fmufun = instantiateFMIFun(dllPath)
    annotation(Include = "#include <fmuWrapper.c>");
  end constructor;
  function destructor
    input fmuFunctions fmufun;
    external "C" freeFMIFun(fmufun)
    annotation(Include = "#include <fmuWrapper.c>");
  end destructor;
end fmuFunctions;
```

An FMU is represented as a Modelica class called `fmuModelInst` in the prototype. In the generic interface, the detailed structure of this class is however hidden behind the so-called *Modelica External Object*. Consequently, the internal structure of the instance cannot be accessed and manipulated directly by Modelica. Provided the necessary inputs are given in Modelica, the instance of an FMU can be declared and immediately instantiated, e.g. `fmuModelInst inst=fmuModelInst(argument_list);`. Later on, this instance reference may be passed as an argument to some external code for various purposes such as data updating, data evaluation, memory freeing, etc. In fact, the actual instantiation is done by calling the external C function `fmiInstantiate(...)` defined in the constructor section of the class `fmuModelInst`, see the following Modelica code for details:

```
class fmuModelInst
  extends ExternalObject;
  function constructor
    input fmuFunctions in_fmufun;
    input String in_instName;
    input String guid;
    input fmuCallbackFuns functions;
    input Boolean logFlag;
    output fmuModelInst inst;
    external "C" inst=fmiInstantiate(in_fmufun,
     in_instName,guid,functions,logFlag)
     annotation(Include="#include <fmuWrapper.c>");
  end constructor;
  function destructor
    input fmuModelInst in_inst;
    external "C" fmiFreeModelInst(in_inst)
    annotation(Include="#include <fmuWrapper.c>");
  end destructor;
end fmuModelInst;
```

There are 24 standard interface functions defined in the *FMI specification v1.0*. Most of them are wrapped in the `fmuWrapper.c` and then mapped one-to-one into Modelica external functions with a slight difference in naming convention to indicate the wrapped functions are not the implementation of FMI but exclusive for FMU import. For instance, the `fmiGetDer(...)` in C

```
void fmiGetDer(void* in_fmi, void* in_fmu,
     double* der_x, size_t nx, const double* x){
  FMI* fmi = (FMI*) in_fmi;
  fmiStatus status;
  if(fmi->getDerivatives){
    status = fmi->getDerivatives(in_fmu,der_x,nx);
    if(status>fmiWarning){
      printf("fmiGetDerivatives(...) failed...\n");
      exit(EXIT_FAILURE);} }
  return; }
```

will be mapped in Modelica as the following snippet:

```
function fmuGetDer "calculate the derivatives"
  input fmuFunctions fmufun;
  input fmuModelInst inst;
  input Integer nx;
  input Real x[nx];
  output Real der_x_out[nx];
  external "C" fmiGetDer(fmufun,inst,der_x_out,nx,x)
  annotation(Include = "#include <fmuWrapper.c>");
end fmuGetDer;
```

2.3 Implementation of FMI Calling Sequence

The UML 2.0 compliant state machine in Figure 2 shows the calling sequence of the interface functions specified in *FMI for Model Exchange v1.0*. This calling sequence has to be implemented in the prototype in Modelica as well.

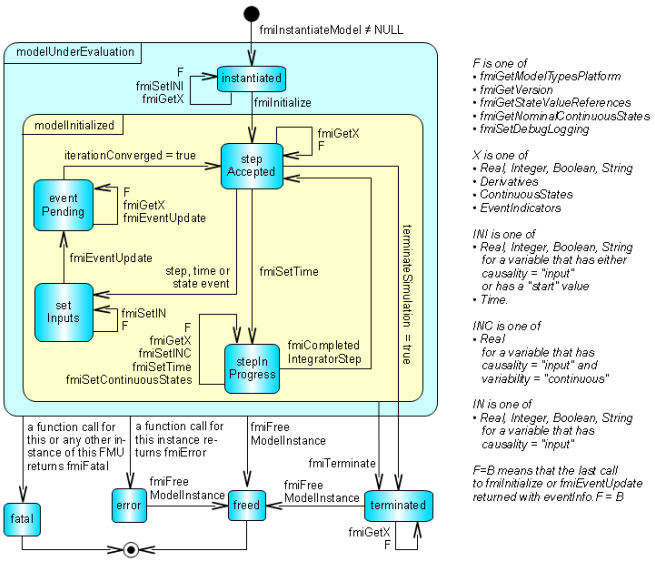


Figure 2. Calling sequence of interface functions[3].

A schematic view of communication between an FMU instance and the environment is also given in the FMI Specification in Figure 3. In this figure, it is shown that different kinds of variables are defined to describe the state of an FMU and the information flowing into\out of an FMU.

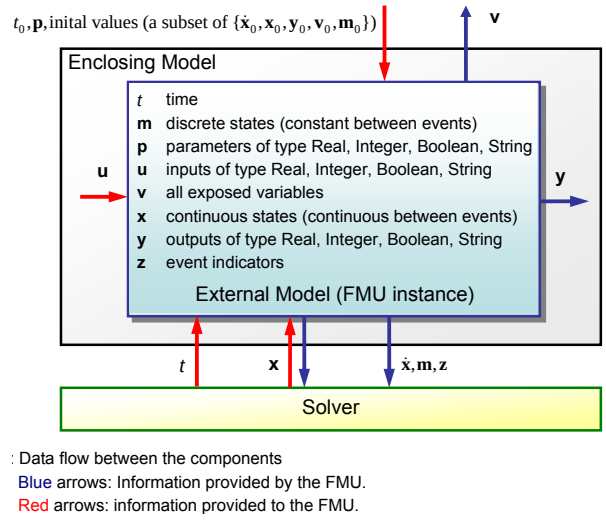


Figure 3. Data flow for information exchange[3].

In order to satisfy the requirements of the function calling sequence and the data exchange of an FMU, a suitable construct in Modelica should be defined and standardized. Despite the fact that Modelica is primarily designed as a modeling language based on acausal equation systems[2], Modelica does offer non-declarative algorithmic constructs, in which procedural methods may be called in a specific order. Due to the arguments discussed previously, the calling sequence and specific data-flows can be realized as a Modelica block incorporating with internal variables, **input****output** signals, **algorithm** and **equation** sections. A snippet of such a Modelica block is given below for illustration.

```
block FMUBlock "a prototype"
  output Real y[ny];
  input Real u[nu];
protected
  constant String dllPath="fmu\\bouncingBall
   \\binaries\\win32\\bouncingBall.dll";
  constant String instName="bouncingBall";
  constant String guid=
     "{9d95f943-e636-4f71-8d7d-6f54c4128f13}";
  parameter Boolean logFlag=false,tolControl=true;
  parameter Integer nx = 2;
  parameter Integer ni = 1;
  replaceable parameter Real relTol = 0.0001;
  replaceable Real x[nx](start={1.0,0.0}),der_x[nx];
  Real out_x[nx],out_der_x[nx];
  ...
  fmuModelInst inst=fmuModelInst(fmufun,instName,
                          guid,functions,logFlag);
  fmuEventInfo evtInfo = fmuEventInfo();
  ...
  fmuFunctions fmufun = fmuFunctions(dllPath);
  fmuCallbackFuns functions = fmuCallbackFuns();
  initial algorithm
  fmuSetTime(fmufun, inst, time);
  fmuInit(fmufun,inst,tolControl,relTol,evtInfo);
  x:=fmuGetContStates(fmufun, inst);
  algorithm
    der_x:=fmuGetDer(fmufun, inst, x);
  equation
    der(x) = der_x;
  algorithm
    fmuSetContStates(fmufun, inst, x);
    fmuCompIntStep(fmufun, inst, stepEvt);
    prez:=z;
    z:=fmuGetEventInd(fmufun, inst, ni);
    fmuStateEvtCheck(stateEvt, ni, z, prez);
    fmuEvtUpdate(fmufun,inst,evtInfo,timeEvt,
           stepEvt,stateEvt,interMediateRes);
    ...
    when zeroCrossing<0 then
      reinit(x[1], out_x[1]);
      reinit(x[2], out_x[2]);
      ...
    end when;
end FMUBlock;
```

2.4 Other Technical Issues

In the FMI Specification v1.0, the *fmiCallbackFunctions* is defined as a *struct* containing function pointers provided by the environment. In Modelica there is no corresponding **record** to store information about function pointers. Nevertheless, Modelica provides the *external object construct*, which allows the user to define and access his or her own external data structures. Thus it is possible to declare a subclass of this `ExternalObject` to instantiate and de-instantiate the *fmiCallbackFunctions* data structure via externally defined C functions.

The FMI specification also states that some functions, such as *fmiInitialize(...)*, should be called only once, but this can not be ensured by the Modelica simulation environment, because during the initialization phase the runtime module may traverse the `initial` **algorithm** section several times. A solution to solve this issue is to define a flag as an initialization indicator in the above mentioned **class** `fmuFunctions`. Whenever this flag records a call for initialization, the *fmiInitialize(...)* will not be called again.

Since the predefined type `Boolean` in Modelica and *fmiBoolean* in FMI are implemented differently in terms of C types, when importing an FMU into most Modelica simulators, the conversion between these two types needs to be considered carefully. The former, i.e. the `Boolean` type, corresponds to *unsigned char* in C in the OpenModelica context. The latter, i.e. the *fmiBoolean* in FMI, corresponds to *char* in C. As a consequence, external functions have been implemented in the wrapper to perform the necessary bi-directional conversions and these functions can be called directly from Modelica.

Connecting imported FMU blocks together hierarchically might arise the difficulty in determining the correct call sequence of setter and getter methods in FMUs for input and output signals. A general alternative is to determine the call sequence according to the connection graph based on some causal relationships between these signals. As soon as the call sequence has been sorted, flags associating with the setter or getter methods can be added in the argument list to indicate the correct order. In this paper, since the imported FMUs are transformed into Modelica models, the correct call sequence of setter and getter methods of input and output signals can be automatically determined by the Modelica simulation environment, [3] in Appendix B.5.

2.5 Difficulties in the Implementation

Some functionalities defined in FMI specification cannot be mapped directly into Modelica without specific treatment.

FMI functions *fmiCompletedIntegratorStep*, *fmiEventUpdate* and *fmiTerminate* are not constant functions, so they do not behave as mathematical functions and there might be a need to mark these functions as "impure" in Modelica via some extension to the language.

The FMI function *fmiCompletedIntegratorStep* cannot be defined and called properly in Modelica, since Modelica language does not provide the functionalities, through which the status of a integration step can be queried and when an integrator step is finished can be detected.

When calling the FMI function *fmiGetEventIndicators* from Modelica model, it will introduce the hysteresis twice to the event indicators. According to the FMI specification, FMUs will add a small hysteresis to the event indicators to avoid event "chattering". And a Modelica tool will do the same when calculating the "zero crossing". So the resulted event triggered by the imported FMU is slightly inaccurate. This problem can be solved by calling external functions, e.g. `fmuStateEventCheck(...)`, which will check the events according to the current values and the previous values of the event indicators given as function arguments.

This paper focuses mainly on FMU import in OpenModelica environment, some of these tricky issues requiring specific treatments are temporarily handled in OpenModelica.

In the presented prototype, the Modelica model is partially automatically generated by the code generator. A full automation of the generation process will be completed soon. After the translation from an FMU to a Modelica model, any instances of this FMU can now be declared as " `FMUBlock fmu1,fmu2,...;` " and connected with other Modelica models. Such capabilities are shown below about the the Bouncing Ball case study.

In the case study discussed in Section 3, the only kind of events are state events. The state event is triggered by the simulation environment outside the FMU model via the `FMUBlock`. The function `fmuStateEvtCheck(...)` will check for the state event and return information through the formal *fmiBoolean* parameter `stateEvt`. If a state event occurs, the `fmuEvtUpdate(...)` is called and updates the states accordingly. Other events, such as time events and step events, might also occur during simulations of dynamic systems, but they are not covered in the case study Bouncing Ball.

3. Case Study: Bouncing Ball

The bouncing ball example is a good candidate to show the behavior of a hybrid dynamic system and the capability of event handling in the presented FMU import prototype. The bouncing ball FMU is generated using the free software development kit *FMU SDK* provided by QTronic GmbH. [5]

3.1 Multiple Instances of an FMU

As mentioned previously, multiple instances of an FMU block can be easily created as below by the standard variable declaration in Modelica. Furthermore, modifications can also be applied to the declared variables e.g.:

```
model FMUMultipleInstance
  FMUBlock fmu1,fmu2(redeclare parameter Real
  relTol = 0.002, redeclare Real x[nx]
  (start = {2.0,0.0}));
end FMUMultipleInstance;
```

The simulation results of the Modelica code above in *OpenModelica 1.7.0* are illustrated in Figure 4, which shows the expected property of multiple instances of an FMU block.

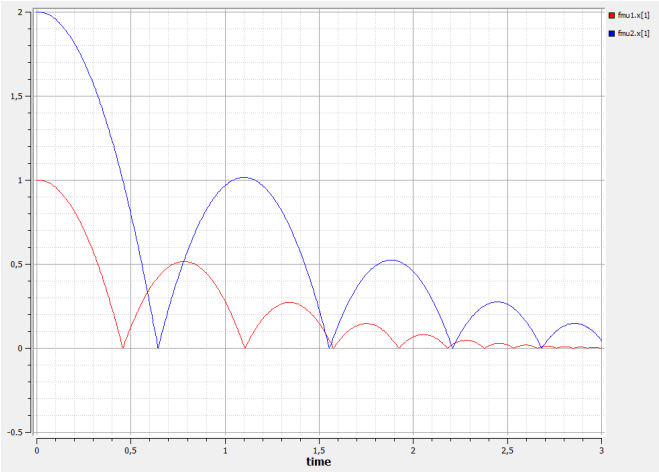


Figure 4. Results of multiple instances of an FMU

3.2 Connection with another Modelica Model

A simple connection between an FMU instance and a nonlinear block from the Modelica Standard Library 3.1 (MSL 3.1) is shown in Figure 5. From the user perspective, the connection between an FMU instance and any Modelica models becomes nearly trivial when using our generic interface. The Modelica code is:

```
model FMUConnection
  FMUBlock fmu;
  Modelica.Blocks.Nonlinear.VariableDelay vd
    (delayTime = 0.2);
equation
  vardelay.u = fmu.x[1];
end FMUConnection;
```

and the simulation result from *OpenModelica* is given in Figure 5, which also produced the expected delay of the input signal.

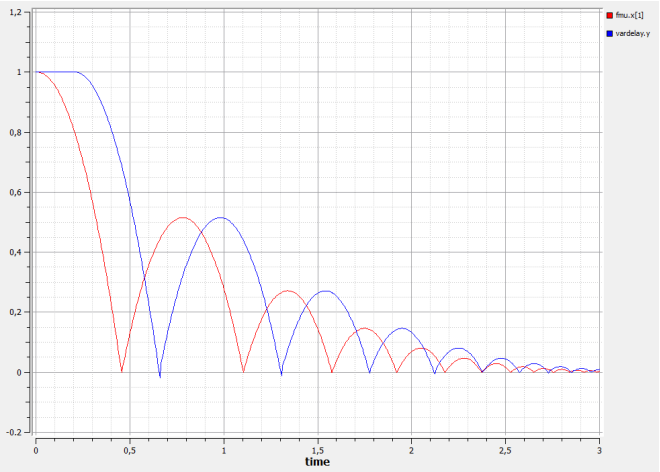


Figure 5. Results of connection between an FMU instance and Modelica model

4. Conclusions and Outlook

In this paper, we presented the concept of a generic interface in Modelica and a prototypical implementation. The interface enables multiple instances of an imported FMU and a simple connection between FMU instances and Modelica models. Of course, any properties provided by the Modelica modeling language, e.g. inheritance, modification, extension and so on, can be applied to the FMU block and its declared variables. The approach presented here fills the gap between the vendor-neutral FMI for model exchange and currently existing vendor-dependent FMU import implementations. An obvious drawback of our approach is the the higher memory consumption compared to simulating a pure Modelica model or simulating an FMU in a stand-alone simulator, because storages have to be assigned for both Modelica objects and FMU instances. Future work includes more test cases of the prototype, full automation of the code generation and tuning to support other Modelica simulators. It is also worth mentioning that FMI specification is not Modelica specific, so a similar approach for FMU import may be checked with other behavioral languages, e.g. VHDL-AMS, provided they support some mechanism to access external functions.

However, the following fundamental requirements need to be satisfied to facilitate FMU import. Obviously, for the generic FMU interface for Modelica to work properly, the provided FMU needs to be fully compliant with the *FMI for Model Exchange v1.0 Specification*. For instance, the FMUs generated from Dymola®7.4 do not support multiple instances of an FMU at the moment. Secondly, the simulator must not modify the generated Modelica models during loading the FMU block, which is not guaranteed by some simulators. Some simulators adapt the loaded Modelica code according to their specific rules, which creates problems in this situation. And thirdly, but not last, the construction of the *Modelica External Object* construct has to be well supported by the simulator.

Acknowledgments

This work is funded by the Federal Ministry of Education and Research (BMBF), Germany, in the ITEA2 project OPENPROD (grant 01IS09029D).

We are also thankful to Martin Sjölund and Mohsen Torabzadeh-Tari from Linköping University for the productive discussions on this topic.

References

[1] Christian Andersson, Johan Åkesson, Claus Führera, and Magnus Gäfvert. Import and Export of Functional Mock-up Units in JModelica.org. In *Proceedings of 8th Modelica Internationl Conference*. Modelica Consortium, 2011.

[2] Peter Fritzson. *Principle of Object-Oriented Modeling and Simulation with Modelica 2.1*. WILEY-INTERSCIENCE, 2003.

[3] MODELISAR. Functional Mock-up Interface for Model Exchange 1.0. http://www.functional-mockup-interface.org.

[4] Christian Noll, Torsten Blochwitz, Thomas Neidhold, and Christian Kehrer. Implementation of Modelisar Functional Mock-up Interfaces in SimulationX. In *Proceedings of 8th Modelica Internationl Conference*, Webergasse1, 01067 Dresden, Germany, 2011. Modelica Consortium.

[5] QTronic. FMU SDK (FMU Software Development Kit). `http://www.qtronic.de/de/fmusdk.html`.

Modeling Language Design

Equation-Based Model Data Structure for High Level Physical Modelling, Model Simplification and Modelica-Export

Hisahiro Ito[1,*] Akira Ohata[1] Ken Butts[2,**]
Jürgen Gerhard[3,***] Masoud Abbaszadeh[3] David Linder[3] Erik Postma[3] Elena Shmoylova[3]

[1]Toyota Motor Corporation, 1200 Mishuku, Susono, Shizuoka, 410-1193, Japan
[*]ito@hisahiro.tec.toyota.co.jp
[2]Toyota Technical Centre, 1555 Woodridge Avenue, Ann Arbor, Michigan, 48105-9748, USA
[**]ken.butts@tema.toyota.com
[3]Maplesoft, 615 Kumpf Drive, Waterloo, Ontario, N2V 1K8, Canada
[***]jgerhard@maplesoft.com

Abstract

This paper proposes a novel data structure for equation-based plant models. The data structure facilitates the plant modelling process starting from physics-based component creation through model simplification to Modelica model export.

Keywords DAE, HLMD, HLMT, Maple, Modelica, Model Simplification, MSModel, Symbolic Manipulation

1. Introduction

The importance of plant models has been increasing in the automotive industry where control systems must address more stringent requirements for emissions, fuel economy, and functional safety than ever before. Moreover, competitive business pressures dictate that development time and effort are effectively managed. One key strategy to meet these challenges is to embrace Model-Based Development (MBD), thereby enabling concurrent development of the plant and controller subsystems as opposed to the conventional serial process where the plant is developed first and the controller second.

Concurrent plant–controller development implies that there is no hardware (i.e. target plant) available for experiment. Thus, in order to service controller development, it is critical that the plant model can be developed in a timely manner. Although there are many plant model libraries available, they are not (and will never be) sufficient to meet the new and more challenging requirements that arise from new and more sophisticated control system development projects. As a consequence, we conclude there is always a need to create new plant models.

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zürich, Switzerland.
 Proceedings available at:
http://www.ep.liu.se/ecp/056/
EOOLT 2011 website:
http://www.eoolt.org/2011/

The traditional approach to build a plant model is to first construct a set of equations to describe the dynamics of the system. However, in this approach, it is the author's responsibility to ensure that the model adheres to physics-based principles such as conservation and it is very difficult for other people to check the quality of the model (i.e. conformance to physics-based principles), especially when the model is large and complex.

In order to overcome this problem, we developed the High Level Modelling (HLM) framework [1, 5] wherein a formalized and physics-based plant-model-development process is defined. If one follows this process, a proper description of the system in question can be created. We call this description the High Level Model Description (HLMD). With HLMD, those who are not the author of the model can readily review and critique the design of the model. In addition, HLMD-based models are easier to reuse and/or modify than traditional equation-only models due to the clear exposition of design intent.

To verify the feasibility of our HLM framework, we also developed a software package called the High Level Modelling Tool (HLMT). With HLMT, one can create an HLMD of a system by following the formalized modelling process and successively run a simulation. Due to the nature of HLMD where the system is represented in highly redundant, non-linear differential algebraic equations (DAE), Maple's [3] symbolic manipulation technology is used to derive simulatable equations. To date, several application models of various physical domains with different levels of complexity have been successfully created and simulated in HLMT. These models range from simple electric circuits to an internal combustion engine with crankshaft angle resolved gas flow and mechanical dynamics.

With the aim of streamlining the plant modelling process even further, we have developed a generalized data structure called `MSModel` to accommodate information about an equation-based plant model. While this data structure can store information generated in HLMT, it is designed to be flexible enough so that one can efficiently ap-

ply a variety of model simplification methods to the model. Furthermore, `MSModel` contains sufficient information to export the model to the Modelica language, which is becoming one of the most widely accepted plant modelling languages [4].

This paper is organized as follows. In Section 2, the HLM framework is briefly introduced. In Section 3, the `MSModel` data structure is explained with some examples. The full specification of the `MSModel` data structure is described in Appendix A.

2. High Level Modelling Framework

As mentioned in Section 1, the HLM framework was developed to enable peer review of the design of a plant model from the standpoint of adherence to physical principles and to realize rapid modelling. In this section, HLM is briefly introduced.

2.1 Formalized Modelling Process and HLMD

In the HLM framework, the process to create a plant model is formalized as follows:

1. Partition the system in question into components
2. Define conserved quantities (CQ) in each component
3. Connect CQs to specify how they flow from one component to another
4. Set physical constraints as needed

By following this process, a physical description called HLMD of the system can be created.

As an example, let us consider a chemical reaction process in a closed chamber where the mixture of hydrogen gas and oxygen gas is burned, and water and heat are generated (Figure 1).

$$H_2 + O_2 \rightarrow H_2O + \frac{1}{2}O_2 + c\,[J]$$

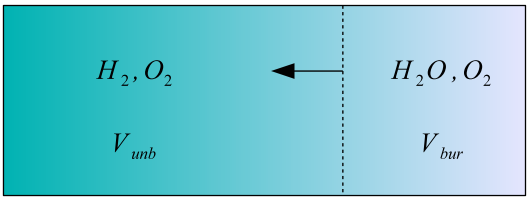


Figure 1. Combustion inside a chamber.

In this case, homogeneous mixing is assumed, and also the combustion is assumed to occur at a thin layer as indicated in Figure 1 by the dashed line traveling inside the chamber from right to left. When modelling of this system is considered, the following constraints apply:

$$p_{\text{unb}}(t) = p_{\text{bur}}(t)$$
$$V_{\text{unb}}(t) + V_{\text{bur}}(t) = V_0 \qquad \text{(const)}$$

where $p_{\text{unb}}(t)$ and $p_{\text{bur}}(t)$ are the pressures of the unburned and burned gas, respectively. $V_{\text{unb}}(t)$, $V_{\text{bur}}(t)$ and V_0 are the volumes of the unburned gas, burned gas, and combustion chamber, respectively.

The HLMD of this system is shown in Figure 2 where a component (e.g. “unburned”, “flame front” and “burned”) is represented as a square with rounded corners, a CQ is represented as a circle, a flow of CQ is represented as an arrow whose direction specifies the sign convention, and constraints are represented as a square with dashed lines connected to components.

Equations appearing in the HLMD are assembled to form the system equations. (Note that some of the equations are omitted in Figure 2 for the sake of simplicity.) For example, we have the number of moles of generated water molecule when the conservation law is applied:

$$\frac{d}{dt}N_{H_2O,\text{ff}}(t) = n_{H_2\text{to}H_2O,\text{ff}}(t) + n_{O_2\text{to}H_2O,\text{ff}}(t) - n_{H_2O,\text{ff}}(t) - e_{\text{ff}}(t)$$

The HLM framework allows the use of intermediate variables in addition to CQs and flows. For example, the average degrees of freedom of the unburned gas $f_{\text{unb}}(t)$, and burned gas $f_{\text{bur}}(t)$, the gas pressure $p(t)$ which is used instead of either $p_{\text{unb}}(t)$ or $p_{\text{bur}}(t)$, as well as $V_{\text{unb}}(t)$ and $V_{\text{bur}}(t)$, are defined as intermediate variables, from which the energies of the unburned gas $E_{\text{unb}}(t)$, and burned gas $E_{\text{bur}}(t)$, are computed as CQs:

$$E_i(t) = \frac{f_i(t)}{2} \cdot p(t) \cdot V_i(t)$$

where i is “unb” or “bur”.

Since the system equations assembled from HLMD are highly redundant and nonlinear, some pre-processing is necessary in order to perform a simulation (i.e. numerical integration). Redundancy removal is obviously needed, but we also need to deal with non-linearities which may lead to multiple solutions. Consequently, a more sophisticated symbolic manipulation algorithm must be used to derive the physically meaningful single solution. Although we have a working algorithm, we consider that this is one of the most challenging aspects of the HLM approach, and, thus, it is still a research topic to improve the robustness and scalability of the multiple solution algorithm.

Apart from the mathematical aspects of the HLMD, the purpose of creating an HLMD is to enable peer review on the design of the model at the physics level. For example, by looking at the HLMD in Figure 2, one can notice that no interaction between gas and chamber is considered. Although it is possible to make such an analysis by only examining the equations, what is done in such a case is to construct a kind of HLMD in the mind. Also, the acceptability of the current model design depends on the purpose of the model. The HLMD allows a physics level examination in an efficient manner.

2.2 Tool Support for HLMD

To evaluate the feasibility of the HLMD and its mathematical framework, a software package called the HLMT was developed by Maplesoft and Toyota. The tool is still a prototype and Toyota owns the intellectual property. With HLMT, one can author an HLMD and perform simulation.

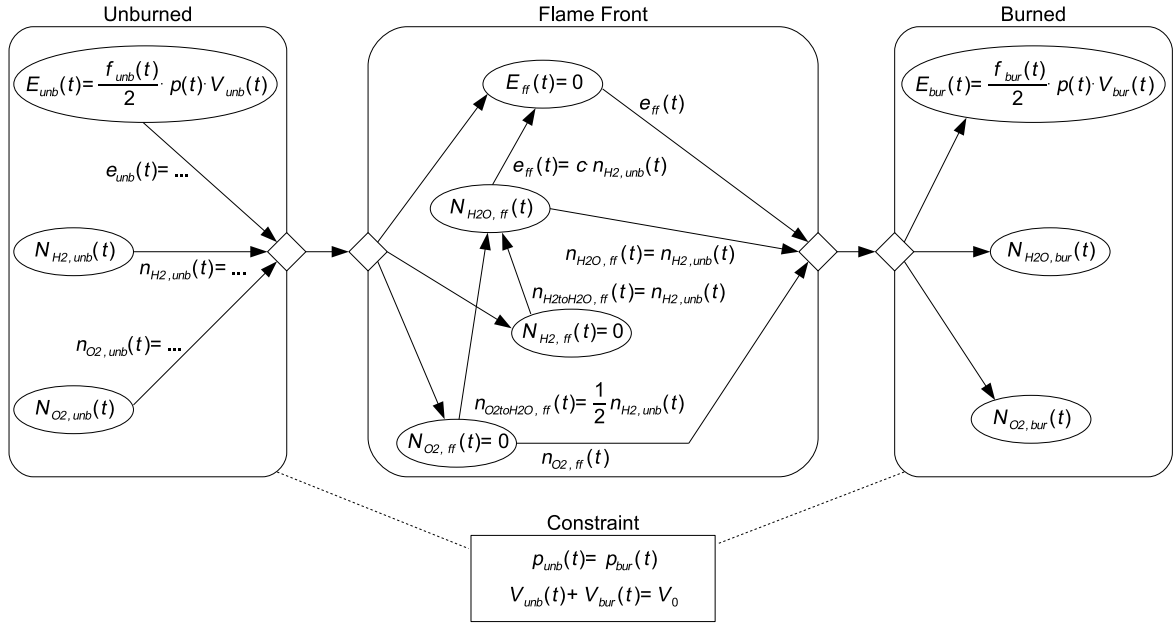


Figure 2. High Level Model Description (HLMD) example - hydrogen-oxygen combustion in a closed chamber.

HLMT also allows export of the model at the equation level to Maple where symbolic model manipulation and numerical simulation services can be applied.

A simulation result of the combustion model explained above is shown in Figure 3. It can be confirmed that the two algebraic constraints, one for pressure and the other for volume, are met.

By exporting the model from HLMT to Maple, some equation-level information about the model can be obtained. For example, it can be known that, after our simplification algorithm derived a single simulatable solution, this model has 11 differential equations. Seven of those are explicit for 7 differential variables $\frac{d}{dt}p(t)$, $\frac{d}{dt}N_{H_2,\mathrm{unb}}(t)$, $\frac{d}{dt}N_{O_2,\mathrm{unb}}(t)$, $\frac{d}{dt}N_{H_2O,\mathrm{ff}}(t)$, $\frac{d}{dt}N_{H_2O,\mathrm{bur}}(t)$, $\frac{d}{dt}N_{O_2,\mathrm{bur}}(t)$ and $\frac{d}{dt}f_{\mathrm{bur}}(t)$. Four other differential equations are implicit interms of 1 differential variable $\frac{d}{dt}V_{\mathrm{unb}}(t)$, and 3 algebraic variables $e_{\mathrm{unb}}(t)$, $n_{O_2,\mathrm{unb}}(t)$ and $n_{O_2,\mathrm{bur}}(t)$. It can also be confirmed that there is 1 DAE constraint, and that $V_{\mathrm{bur}}(t)$ which does not appear in the aforementioned variables is calculated from $V_{\mathrm{unb}}(t)$.

Since there are not only statistics but also full access to all of the equations once the model is exported to Maple, it is also possible to perform other types of manipulation such as applying additional model simplification methods including approximation, or exporting to yet another modelling and simulation environment.

Eventually, after successful modelling and simulation in HLMT for models with different levels of complexity, together with research progress in Maple-based model simplification for HLMT-generated models, we decided to develop a flexible data structure by which the modelling process including simplification can be facilitated.

3. Data Structure for Model Simplification

Requirements for the model simplification data structure include 1) it can store information generated in HLMT, 2) it can store a simplified set of equations, 3) it can provide convenience for model simplification methods, and 4) it can generate a Modelica representation of the stored model.

The third requirement may need more breakdown as we develop our model simplification methods, but we now have a realization of this data structure called `MSModel`. An `MSModel` data structure is captured as a Maple Record and it serves as a modelling research artefact with which more a thorough design of the data structure can be made.

In this section, the current implementation of the `MSModel` is explained with some examples. For the complete coverage of the specification of `MSModel`, please see Appendix A.

3.1 Overview of `MSModel` Data Structure

The `MSModel` data structure consists of the following pieces of information.

- An independent variable (e.g. time t)
- Differential equations (DE) and variables (DV)
- Algebraic equations (AE) and variables (AV)
- Intermediate equations (IE) and variables (IV)
- Dependent equations and variables
- Parameters
- Inputs and outputs
- Name, type and value of variables
- Blackbox functions (e.g. lookup tables and user-defined functions)

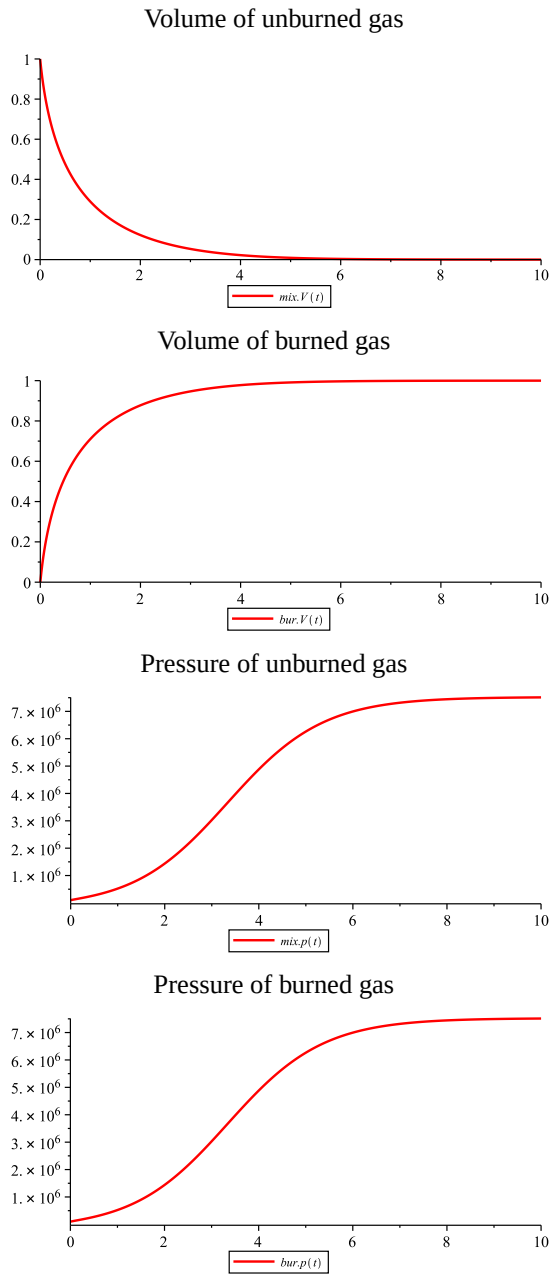


Figure 3. Simulation result of the combustion model, produced in HLMT.

These elements are stored in a Maple Record in a structured manner. A fictitious example of `MSModel` is shown in Figure 4.

In this example, each Record entry is accessible by, for instance, `msm:-DE[1]` for a list / array element or `msm:-variables[x1]` for a table element. The name `msm` is just an example and any Maple variable name is fine.

The combination of differential equations, `msm:-DE`, algebraic equations, `msm:-AE`, and intermediate equations, `msm:-intermediate`, comprise the minimum set of equations (i.e. *core* equations) with which the time evolution of the system can be computed.

```
msm := Record(MSMODEL,
 DE=[
  ( diff(x1(t),t)=-a*x1(t)+u1(t) ),
  ...],
 DV=[ 'x1' , ...],
 AE=[],
 AV=[],
 t='t',
 intermediate=(Array(1..0,[])),
 intermediateVariables=[],
 dependent=(Array(1..3,{
  1=[{ e1(t)=-1/2*sin(x1(t)) },{e1(t)}],
  2=[{ e2(t)=u1(t)*e1(t) },{e2(t)}],
  3=[{ y(t)=e1(t)+e2(t) },{y(t)}] })),
 dependentVariables=[ 'e1' , ...],
 parameters=[ 'a' , ...],
 inputs=[ 'u1' , ...],
 outputs=[ 'e1' , ...],
 variables=(table([
  (x1)=Record(MSVARIABLE,
   name=x1,
   type="differential",
   value=.9,
   unit=(NULL)),
  (a)=Record(MSVARIABLE,
   name=a,
   type="parameter",
   value=2,
   unit=(NULL)),
  ...])),
 blackboxes=[]
);
```

Figure 4. `MSModel` data structure example.

While `msm:-DE` and `msm:-AE` are a list of equations in no particular order, `msm:-intermediate` is an array ordered in a straight-line causal arrangement. The example in Figure 4 has no element in the `msm:-intermediate` field, but the example `msm:-dependent` field contains an array of such an arrangement with 3 elements. In both `msm:-intermediate` and `msm:-dependent`, each element has two sub-elements. The first sub-element is the equation, the second is the variable to be determined by that equation. There are two significant differences between `msm:-intermediate` and `msm:-dependent`. 1) In `msm:-intermediate`, there are no derivatives contained whereas in `msm:-dependent`, equations can be implicit and/or differential. 2) equations in `msm:-intermediate` must be computed at every integration step time while those in `msm:-dependent` have to be computed only when necessary.

Differential, algebraic, intermediate and dependent variables are stored in `msm:-DV`, `msm:-AV`, `msm:-intermediateVariables`, and `msm:-dependentVariables` as a list of variable names, respectively. Variables in these lists are stored, not as a function of the independent variable, but just as a name.

A single independent variable by which all other variables may be parameterized is stored at `msm:-t`.

Parameters, inputs and outputs are stored in `msm:-parameters`, `msm:-inputs` and `msm:-outputs` as a list of their names, respectively.

The `MSModel` Record has an entry called `variables` which is a table of all the variables including parameters described above with their type, initial value and unit information. In the example Record in Figure 4, the unit has not been assigned to any variable.

To deal with special types of functions, specifically lookup table functions and user-defined functions, the `MSModel` Record has an entry called `blackboxes`. It can store information such as the dimension and the data of a lookup table, or the definition of a user-defined Maple procedure.

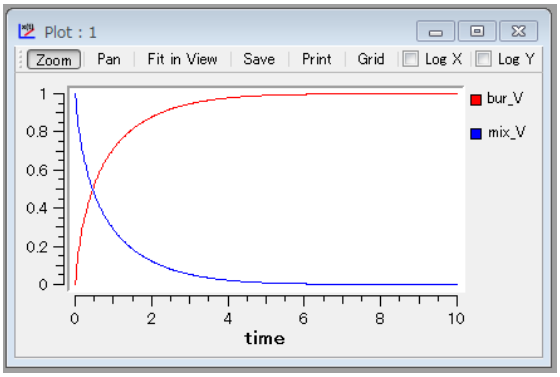
3.2 Workflow Example for HLMT, `MSModel`, Simplification and Modelica-Export

To illustrate the use of an `MSModel` data structure as part of the plant modeling process, here the hydrogen-oxygen combustion model mentioned in the previous section will be exported from HLMT to Maple, and stored in `MSModel`. Then simplification methods will be applied to it, and the reduced model will be exported to Modelica, and finally run in a Modelica-based simulation tool.

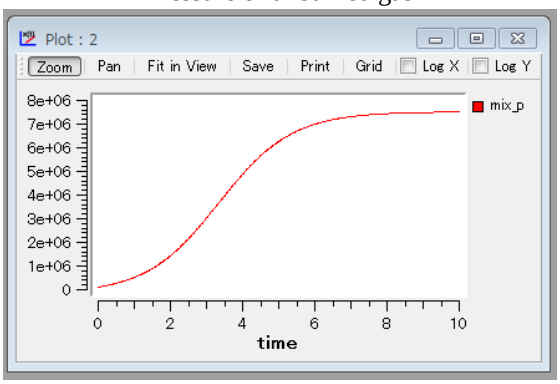
After the first equation-based simplification algorithm was applied to the HLMD of the system, the model consisted of 11 DEs for 8 DVs and 3 AVs (see Section 2.2) with 0 IEs. It also has 32 dependent equations and the same number of dependent variables. Once this model is stored in `MSModel`, other symbolic simplification algorithms which Maplesoft and Toyota developed were applied to it. As a result, 10 DEs for 7 DVs and 3 AVs with 4 new IEs for 4 new IVs consisted of the core equations while 1 DE for $\frac{d}{dt}N_{H_2O,\mathrm{ff}}(t)$ was categorized as one of the 8 dependent equations.

From this `MSModel`, a Modelica language file was generated, and it was simulated in OpenModelica [2] with the dassl solver as shown in Figure 5.

Volumes of unburned gas and burned gas



Pressure of unburned gas



Pressure of burned gas

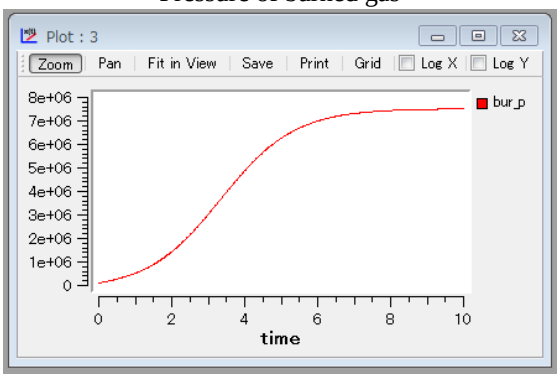


Figure 5. Simulation result of the combustion model, produced in OpenModelica.

4. Conclusion

Part of a plant modelling process for Model Based Development was introduced. A physics-based modelling framework as well as a novel data structure for model simplification and Modelica-export were developed. It was shown that these tools and data structure can greatly streamline the plant modelling process.

References

[1] Bakus J. Bernardin L. Gerhard J. Kowalska K. Léger M. Wittkopf A. High-Level Physical Modeling Description and Symbolic Computing. *IFAC Proceedings of the 17th World Congress*, pages 1054–1055, 2008.

[2] Fritzson P. *et al.* http://www.openmodelica.org/. *The OpenModelica Project.*

[3] Maplesoft. Maple 11 User Guide. 2007.

[4] Modelica Tools. http://www.modelica.org/tools. *Modelica Association*, 2011.

[5] Ohata A. Ito H. Gopalswamy S. Furuta K. Plant Modeling Environment Based on Conservation Laws and Projection Method for Automotive Control Systems. *SICE Journal of Control, Measurement and System Integration*, 1(3):227–234, 2008.

A. Appendix - The `MSModel` Data Structure

The `MSModel` data structure is implemented as a Maple Record, described in Section A.1. The Record structure is designed as a general container whose nested contents may be efficiently accessed for both reading and writing. A Record has a number of exports, or fields, which can be arbitrary Maple objects. A given field can itself be, for ex-

ample, a list of lists or a table of Records. In particular, two types of sub-Records occur inside the MSModel Record: the MSVariable Record used to give detailed information about variables and described in Section A.2, and the MSBlackBox Record, used to give detailed information about black box functions and described in Section A.3.

A.1 MSModel Record type

The MSModel data structure is implemented as a Maple Record. The particular structure and purpose of the individual fields of an MSModel Record are described below. The fields are references to some MSModel record named model.

model:-DE

A list of the core differential equations of the model. There is no additional structure or order to the appearance of the equations in this list. Together with the algebraic equations model:-AE and the intermediate equations model:-intermediate, these equations comprise the set of *core* equations which capture the characteristic dynamics of the system.

model:-DV

A list whose elements are of type name. This list designates the full set of differential variables present in the equations in either the DE, AE, or intermediate fields. In particular, the derivative of each variable in DV occurs in DE.

model:-AE

A list of the core algebraic equations of the model. Equations appear in this field by virtue of not being core differential or intermediate equations of the model. There is no additional structure or order to the appearance of the equations in this list.

model:-AV

A list whose elements are of type name. This list designates the full set of the algebraic variables present in the equations in either model:-DE, model:-AE, or model:-intermediate, but excludes the input and intermediate variables. These variables do not occur differentiated anywhere in the model.

model:-t

This field is a single name, which designates the independent variable (e.g., time) by which all other variables may be parameterized.

model:-intermediate

An Array specifying the intermediate equations. These equations are grouped into subgroups, each subgroup giving the equations to determine a certain subset of the intermediate variables. The elements of the Array are lists consisting of two sets: first a set of explicit equations, then the set of intermediate variables determined by those equations (represented as functions of model:-t); in other words, the second set forms the left hand sides of the equations making up the first set.

These lists are ordered in a straight-line causal arrangement; that is, the only variables occurring in right-hand sides of equations are either input variables, or differential variables, or they are intermediate variables that have been defined in earlier elements of the Array. Furthermore, model:-intermediate contains no derivatives.

The equations could be substituted into each other and then into model:-DE and model:-AE to obtain the core equations in a more explicit form.

model:-intermediateVariables

This is a list of the variable names appearing in all the second sets of the elements of model:-intermediate, or equivalently, on the left hand side of the equations in that field. These variables do not occur differentiated anywhere in the model.

model:-dependent

This is a list of differential or algebraic equations which are not part of the core dynamics of the model. Like model:-intermediate, the equations are given by an Array of lists consisting of two sets, where each first set specifies algebraic or differential equations or expressions determining the values of the dependent variables specified (as functions of model:-t) in the corresponding second set. Also, like model:-intermediate, the lists are ordered in a straight-line causal arrangement; in this case, that only means that dependent variables occurring in the equations are never defined in later elements of the Array (algebraic and differential core variables, intermediate variables, and input variables can occur throughout model:-dependent).

A difference between model:-dependent and model-:-intermediate is that equations in the former can be implicit or even differential.

Dependent equations can be solved after the core variables have been solved.

model:-dependentVariables

This is a list of all dependent variable names. That is, it is the list of variables that are not part of the core system or input or intermediate variables. They are to be solved in terms of core variables, and correspond to the entries of the second set of each list in model:-dependent.

model:-parameters

A list of parameter names. This list designates the full set of parameters of the model. A parameter is understood to be a quantity that, by design, does not vary over time.

model:-inputs

A list of input variable names. If the model is considered as a subsystem, then these variables are the input ports.

model:-outputs

A list of output variable names. If the model is considered as a subsystem, then these variables are the output ports.

Each output variable also occurs in exactly one of the fields DV, AV, intermediateVariables, dependentVariables, and inputs.

model:-variables

A table whose indices are of type name and whose corresponding entries are themselves Records. The indices are precisely the names of the variables, parameters, and blackbox functions occurring in the model. That is, the indices are precisely those names occurring in the fields DV, AV, intermediateVariables, dependentVariables, inputs, parameters, and blackboxes. Again in other words, the name of every function called in model:-DE, model:-AE, model:-intermediate, and model:-dependent that is not a function built-in to Maple is an index into the model:-variables table.

Each entry in the table indexed by the name of a variable or parameter is a Record of MSVariable type, which is described below. Each entry in this table indexed by the name of a blackbox function has as its entry a Record of MSBlackBox type, which is also described below.

model:-blackboxes

A list of undefined names representing lookup table functions and user defined functions, which may occur in any equation of the model. For example, as $z_{23}(t) = L1(z_{35}(t))$, where the right hand side is an unevaluated function call.

A.2 MSVariable Record type

This type of Record is used to store detailed information about any variable or parameter of the model. This includes core, input, dependent, and intermediate variables, and parameters. Such Records appear as entries of the variables field of the parent MSModel Record.

The fields of an MSVariable Record, named here as variable, are as follows:

variable:-name

The name of the variable described by the Record. This is the same as the index of the Record in the model:-variables table.

variable:-type

The role of the variable, given by one of the strings "algebraic", "differential", "intermediate", "dependent", "parameter", or "input". This corresponds to the field of the MSModel Record that variable:-name occurs in, as follows:

variable occurs in model:-	variable:-type
AV	"algebraic"
DV	"differential"
intermediateVariables	"intermediate"
dependentVariables	"dependent"
parameters	"parameter"
inputs	"input"

variable:-value

For parameters, this field holds the value for this parameter. For differential variables (those in model:-DV and the differential ones in model:-dependent), it contains the initial value, expressed as a numerical value or an expression in terms of the parameters. For algebraic variables, an initial value can also be given; this may contribute to determining the initial conditions. For input variables this entry will be a function of the independent variable (model:-t). This field may be empty.

variable:-unit

The unit in which the variable is measured. This field may be empty.

A.3 MSBlackBox Record type

This type of Record is used to store the particular information of any implicit lookup table function in the model, and for storing user-defined functions. The otherwise implicit blackbox functions are listed by name in the blackboxes field of the parent Record.

The MSBlackBox Record for a lookup table function contains the original numeric data, stored in tabular form. This allows for easy exporting of a lookup table to corresponding fields of a Modelica representation of the parent model.

The MSBlackBox Record for a user-defined function is similar, but it contains the Maple procedure that implements the user-defined function.

The fields of a MSBlackBox Record, named here as blackbox, are as follows:

blackbox:-name

The name of the blackbox function call described by the Record. This is the same as the index of the Record in the model:-variables table.

blackbox:-type

This string is the type of the black box function: either "lookup" for a lookup table, or "procedure" for a user-defined function.

blackbox:-dimension

The number of arguments of the blackbox function as occurring in the equations. For lookup table functions, this number corresponds to the dimension of the lookup table, and it must currently be either 1 or 2, which corresponds to the current support for lookup tables in MapleSim's implementation of Modelica.

blackbox:-data

For lookup table functions, this field is a list of lists of independent data points; each list contains all values for one of the arguments in strictly increasing order. The number of such lists is equal to blackbox:-dimension. The order of the lists corresponds to the order in which the arguments to the lookup table are given. The number of data points in each sublist must agree with the number of points

in the dependent `blackbox:-value` list for the given lookup table.

For user-defined functions, this field is not used; it may have any value.

`blackbox:-value`

For lookup table functions, this field contains the dependent values that the function attains at the respective data points; that is, the values corresponding to measurements of the dependent variable or quantity. These should either be given as one long list of numerical values, or a list of sublists each containing numerical values. If the lookup table is one-dimensional, then the list should simply contain the dependent values at the points in the same order as given in `blackbox:-data`. If the lookup table is two-dimensional, then the first entry of `blackbox:-value` should be a list of all dependent values where the first coordinate is the lowest value, ordered by increasing value of the second coordinate; the second entry should be a similar list for the second lowest value of the first coordinate, and so on. Alternatively, `blackbox:-value` may be the concatenation of these lists.

For user-defined functions, this field should contain the Maple language procedure that is to be used to evaluate the function.

Safe Compositional Equation-based Modeling of Constrained Flow Networks*

Nate Soule[1] Azer Bestavros[1] Assaf Kfoury[1] Andrei Lapets[1]

[1]Department of Computer Science, Boston University, USA, {nsoule,best,kfoury,lapets}@bu.edu

Abstract

Numerous domains exist in which systems can be modeled as networks with constraints that regulate the flow of traffic. Smart grids, vehicular road travel, computer networks, and cloud-based resource distribution, among others all have natural representations in this manner. As these systems grow in size and complexity, analysis and certification of safety invariants becomes increasingly costly. The NetSketch formalism and toolset introduce a lightweight framework for constraint-based modeling and analysis of such flow networks. NetSketch offers a processing method based on type-theoretic notions that enables large scale safety verification by allowing for compositional, as opposed to whole-system, analysis. Furthermore, by applying types to the modeled networks, analysis of composite modules containing incomplete or underspecified components can be conducted. The NetSketch tool exposes the power of this formalism in an intuitive web-based graphical user interface. We describe the NetSketch formalism and tool, a translation from an instantiation of the NetSketch formalism to the equation-based modeling language Modelica, and the development of an accompanying Haskell library, HModelica, that enables the integration of NetSketch and the OpenModelica modeling platform.

Keywords Flow networks, Network analysis, Safety verification, Constraint based modeling

1. Introduction

Many large scale systems can be modeled as assemblies of subsystems, each of which produces, consumes, or regulates a flow. Such models can contain variables and constraints representing the safe operation of the system. Networks that may be represented in this manner cross many domains within software, hardware, electrical, material, structural and other areas. Electric grids, vehicular road networks, and computer networks are all modeled cleanly in this structure; in addition, so are less immediately obvious examples, such as the governance of service level agreements (SLAs) in cloud computing environments. In the case of SLAs, a physical processor may generate a flow that is regulated by schedulers and consumed by computing processes. In electric grids, power plants may act as nodes producing flow, with transmission lines, and transformers routing and regulating flow to commercial and residential customers (who may in turn act not only as sinks, but as sources when, for example, they have solar panels). Extended detail and further examples are described in separate papers [5, 22, 23]. Verification of safety invariants across such a system is a critical analysis task, but this task can quickly grow costly as the complexity of a model increases. The NetSketch formalism and accompanying tool offer a constraint-based modeling solution capable of handling such complexity while providing an efficient analysis engine.

The nodes in a constrained flow network may contain arbitrarily complex constraints that serve to connect them and regulate their operation. Solving for a set of feasible values for the variables of the system will produce the inputs and outputs that constitute "safe" usage. This is a desirable task both from a modeling perspective: ensuring or discovering the range of safe values, and from a design perspective: considering alternative "what if" scenarios and inspecting their properties in search of optimal values. In large systems the size and complexity of the set of constraints and variables under consideration can limit or even prohibit whole-system analysis. To allow for an analysis under these circumstances, NetSketch employs techniques from type theory to simplify the constraints of the network at various levels of the system's composition. A *type* is given to various subsets of the network under consideration. Each sub-network of nodes can then, for the purposes of analysis, be replaced with an opaque container that exposes only the ports at its interfaces. This new component is then regulated by a type at each of its ports. By considering only the types, and not the potentially complex set of internal constraints, it is possible to more efficiently analyze this new component in the context of the larger network, and to determine safe ways to connect this component to others during a design process.

In this paper we describe the NetSketch formalism and tool, and make connections between this work and the broader equation-based objected-oriented modeling do-

*This work is supported in part by NSF awards CNS-0952145, CCF-0820138, CSR-0720604, CNS-1012798, and EFRI-0735974.

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zürich, Switzerland.

EOOLT 2011 website: http://www.eoolt.org/2011/

$$\textsc{Hole}\quad \frac{(X, \mathsf{In}, \mathsf{Out}) \in \Gamma}{\Gamma \vdash (X, \mathsf{In}, \mathsf{Out}, \{\,\})}$$

$$\textsc{Module}\quad \frac{(\mathcal{A}, \mathsf{In}, \mathsf{Out}, \mathsf{Con})\ \text{module}}{\Gamma \vdash (\mathcal{B}, I, O, \{C\})} \qquad (\mathcal{B}, I, O, C) = {}'(\mathcal{A}, \mathsf{In}, \mathsf{Out}, \mathsf{Con})$$

$$\textsc{Connect}\quad \frac{\Gamma \vdash (\mathcal{M}, I_1, O_1, \mathcal{C}_1) \qquad \Gamma \vdash (\mathcal{N}, I_2, O_2, \mathcal{C}_2)}{\Gamma \vdash (\mathbf{conn}(\theta, \mathcal{M}, \mathcal{N}), I, O, \mathcal{C})} \quad \theta \subseteq_{\text{1-1}} O_1 \times I_2,\ I = I_1 \cup (I_2 - \mathsf{range}(\theta)),\ O = (O_1 - \mathsf{domain}(\theta)) \cup O_2,\ \mathcal{C} = \{C_1 \cup C_2 \cup \{\, p = q \,|\, (p,q) \in \theta \,\} \,|\, C_1 \in \mathcal{C}_1, C_2 \in \mathcal{C}_2\}$$

$$\textsc{Loop}\quad \frac{\Gamma \vdash (\mathcal{M}, I_1, O_1, \mathcal{C}_1)}{\Gamma \vdash (\mathbf{loop}(\theta, \mathcal{M}), I, O, \mathcal{C})} \quad \theta \subseteq_{\text{1-1}} O_1 \times I_1,\ I = I_1 - \mathsf{range}(\theta),\ O = O_1 - \mathsf{domain}(\theta),\ \mathcal{C} = \{C_1 \cup \{\, p = q \,|\, (p,q) \in \theta \,\} \,|\, C_1 \in \mathcal{C}_1\}$$

$$\textsc{Let}\quad \frac{\Gamma \vdash (\mathcal{M}_k, I_k, O_k, \mathcal{C}_k) \quad \text{for } 1 \leqslant k \leqslant n \qquad \Gamma \cup \{(X, \mathsf{In}, \mathsf{Out})\} \vdash (\mathcal{N}, I, O, \mathcal{C})}{\Gamma \vdash (\ \mathbf{let}\ X \in \{\mathcal{M}_1, \ldots, \mathcal{M}_n\}\ \mathbf{in}\ \mathcal{N}\ , I, O, \mathcal{C}'\ )}$$

$$\mathcal{C}' = \left\{ C \cup \hat{C} \cup \{\, p = \varphi(p) \,|\, p \in I_k \,\} \cup \{\, p = \psi(p) \,|\, p \in O_k \,\} \,\middle|\, 1 \leqslant k \leqslant n,\ C \in \mathcal{C},\ \hat{C} \in \mathcal{C}_k,\ \varphi : I_k \to \mathsf{In},\ \psi : O_k \to \mathsf{Out} \right\}$$

Figure 1. Rules for Untyped Network Sketches.

main. In Section 2 we describe the domain specific language at the heart of NetSketch. In Section 3 we describe the NetSketch tool and its architecture. We take a deeper look at the type generation algorithms in Section 4. In Section 5 we investigate the relation of NetSketch to current equation-based object-oriented modeling solutions (Modelica in particular) via two avenues. First, we examine the use of Modelica to assist various computational tasks required by NetSketch; here, we introduce a Haskell library that exposes the power of Modelica to the NetSketch engine. Second, we define a translation from NetSketch models to Modelica that allows for whole system analysis of those models via simulation. Finally, we end with a discussion of related and future work in Section 6.

2. NetSketch Formalism

In a constrained flow network each node of the network or system may impose constraints on its inputs and outputs. The network and its entire constraint set form an exact model[1]. Any whole-system analysis of the network must compute the solution space of the constraint set for the given network. Our compositional approach uses types to approximate the constraints on the interface of each node or group of nodes. In this way sub-systems can be analyzed individually at an exact level, whereas the whole system can be analyzed based solely on the results of the subsystem analyses rather than the entire set of constraints. This method allows for efficient analysis of large systems even when the cost of a whole system analysis does not scale linearly with the size of the system. Further, the compositional aspect of this method allows for analysis to occur in cases where it otherwise would require more information *i.e.*, in incomplete systems. When a portion of the overall system has unknown constraints, but a known interface, NetSketch can infer the types that will allow safe operation of the system using the rest of the network and its connectivity to the incomplete "hole".

[1] Here by "exact" we mean with respect to those properties under consideration in the model. Any model is by neccessity an approximation of the system being represented.

The NetSketch formalism defines a domain specific language for describing constrained flow networks. In its original form [4] the DSL consists of five main constructs: *Module*, *Hole*, *Connect*, *Loop*, and *Let*. These are described below, and the corresponding rules for constructing network descriptions are depicted in Figure 1.

Module *Module* defines a new node in the network. This node is atomic *i.e.*, not composed of other nodes.

Hole A *hole* in a network describes an area that is incomplete (*e.g.*, not yet designed or unknown to the modeler). It provides the information that is known about this hole (only the number of inputs and outputs) without the need to fully specify the constraints. NetSketch enables its users to infer the minimal requirements to be expected of (or to be imposed on) such holes. This enables the design of a system to proceed based only on the promised functionality of missing parts.

Connect *Connect* allows for two distinct networks to be combined into a larger network. This construct binds a subset of the output ports of one network to a subset of the input ports of another. The result is a new network that can in turn be connected to others.

Loop *Loop* allows for the connection of an output port of a network to be connected to an input port of the same network.

Let *Let* is used to specify a set of networks that may be placed in a given network hole.

3. NetSketch Tool and Architecture

The NetSketch formalism is partially implemented by the current version of the NetSketch tool.[2] The NetSketch tool offers users the ability to visually create and define modules, and to create connections among them to form *network sketches*. Subsets of networks may then be selected for inclusion in the type generation process. This paper describes the state of the tool as of its first release, which captures many of the core features of NetSketch but leaves

[2] The tool can be found under Projects → NetSketch at the following URL: `http://www.cs.bu.edu/groups/ibench/`.

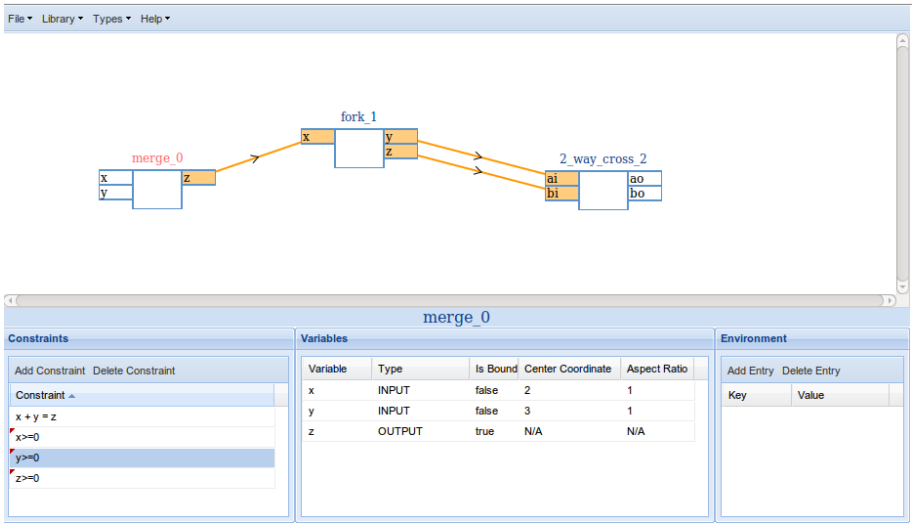


Figure 2. View of a network consisting of 3 connected modules in the NetSketch tool.

others to future implementations. Section 6 discusses some of the functionality yet to be added.

3.1 Interface and User Experience

Figure 2 shows a screen of the NetSketch tool in action. Depicted are three modules from the domain of vehicular traffic: a merge, a fork, and a 2-way cross intersection. The interface of the tool is divided into two main areas. The top represents the canvas onto which users will place modules and create connections between these modules to create networks. The bottom section presents the details of the currently selected module, along with any environment constants.

Creating Modules A user can begin defining a network by first introducing new modules. This can be accomplished by creating a new module from scratch (*i.e.*, with no ports or constraints defined), or by selecting from a library of pre-defined modules and network sketches. Modules from the library come pre-built with a set number of ports (input or output variables), and a base set of linear constraints describing their operational requirements. Both blank and library modules can then be extended by adding, deleting, and modifying ports and constraints.

Ports are only given meaning when included in the constraints of the module containing the port. Thus, port creation is inferred during constraint definition. As a user creates a new constraint, $x + y = z$ for example, the system performs syntactic analysis of the constraint to determine its variables, and automatically updates the list of ports for the module. As constraints are created, modified, and removed, the available ports for the given module will be added or removed as appropriate. Once a port is defined, it must be classified as either an input or an output port[3]. Classifying a port as an input or output causes it to be drawn on the canvas. Input ports align to the left of a module, and output ports to the right.

[3] In future implementations the ability to have internal variables that are neither input or output will be allowed.

Connecting Modules Once constraints are defined, and ports classified a module is ready for interfacing with other modules and networks. The modules can be visually dragged around the canvas to allow for appropriate positioning in relation to other modules with which potential connections exist or to indicate logical groupings/relations. To connect two modules a user creates a line by dragging from the port of one modules to the port of another (or among ports on the same module to create a loop). If port P_1 is connected to port P_2 then either P_1 is an input port, or P_2 is an input port, but not both (*i.e.*, an exclusive-or relationship).

Once two ports are connected their binding status in the **Variables** area of the screen is updated from *false* to *true* and the screen visually indicates this with a line between the ports; an arrow indicates the direction of flow, and both ends of the connection are shaded. Though not represented explicitly in the **Constraints** area of the screen, an implicit constraint is created for every port connection: an equality constraint $P_n = P_m$ is implied for every connection of port P_n to port P_m.

As only the constraints of a single module are displayed on the screen at any given time, variable names need not be unique across a network. Internally, NetSketch performs variable renaming by prepending the module name to the variable name. From the user's perspective only the module specific variable name (*i.e.*, `x, not fork_1.x`) is displayed. This is possible and safe because the system guarantees unique module names through a global counter added to each module name.

Generating Types When a connected set of modules is in a stable state the user can choose to generate a type for that set. By selecting an option from the menubar a type generation window will open. This window, as shown in Figure 3, allows the user to select among the available modules. A type can be created for a single module if the user determines a typed version is easier to manipulate and use than an untyped one, or a subset of connected modules may be collectively typed. The decision regarding the level

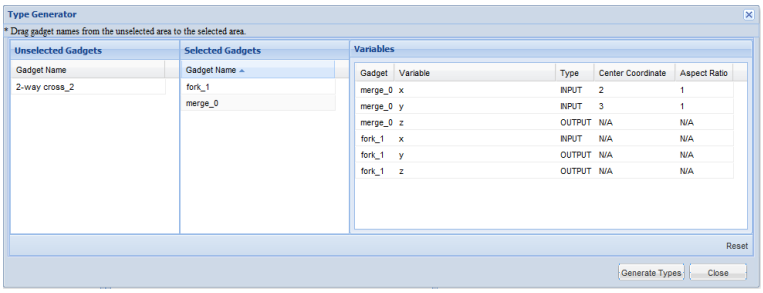


Figure 3. Type generation window with two connected modules selected for type creation.

of granularity in type generation is an important one. This represents the point where exact analysis is replaced with compositional analysis.

At some point the constraint sets in a network of untyped modules may get sufficiently complex such that compositional analysis becomes the preferred (if not only) method for analysis. We define this point as the constraint threshold. The constraint threshold may be determined in any number of ways that might be beneficial to the user (*e.g.*, number of nodes, number of connections, number of constraints, number of variables within the constraints, time taken to bound the feasible region of the solution, the shape of the constraints). Presently, our implementation of NetSketch leaves the decision regarding the value of this threshold to the user.

Once typed, a network is replaced visually by a single container - slightly shaded to differentiate it from untyped modules on the canvas. This process can be infinitely recrusive, in that a network of typed modules can itself be given a type. The constraints shown in the bottom portion of the screen are replaced with the intervals inferred for each port.

The types generated for a set of modules are non-empty intervals over $\mathbb{R}$. For each non-connected port P exposed within the set of modules being typed, an interval of the form P: $[P_{\min}, P_{\max}]$ will be generated. Optimal typings of this form can not be guaranteed to be uniquely generated for the *input* variables without further guidance from the user. In this implementation, this guidance takes the form of a center point and an aspect ratio relating all *input* variables. See Section 4 for the reasons behind this requirement and the details of the center-point/aspect-ratio solution, and Section 6 for a description of work underway to alleviate this need.

With types generated, what were formerly potentially complex and numerous constraints are now simple intervals that can be viewed, composed, and analyzed efficiently. In addition, with this level of typing, unknowns in the network can easily be left as *holes* that can have their typings inferred without further specification simply by connecting them appropriately to defined modules and networks. Holes can be created in a fashion similar to that of modules, with the exception that ports are listed explicitly as opposed to being inferred from the constraint set. The *Let* construct of the formalism, which describes which modules may be placed in a hole, is applied in the tool via the ability to select existing components from the library as potential hole replacements.

Persistence At any point the user can chose to save their canvas in a persistent form. The tool will convert the internal representation of the model into JavaScript Object Notation (JSON), and prompt the user to open or save the generated JSON file. Users can then later load their saved modules from disk to continue their modeling/analysis effort.

3.2 Architecture

The NetSketch tool architecture comprises a client component and a server component as represented by the **User Interface** and **Core Engine** boxes respectively in Figure 4.

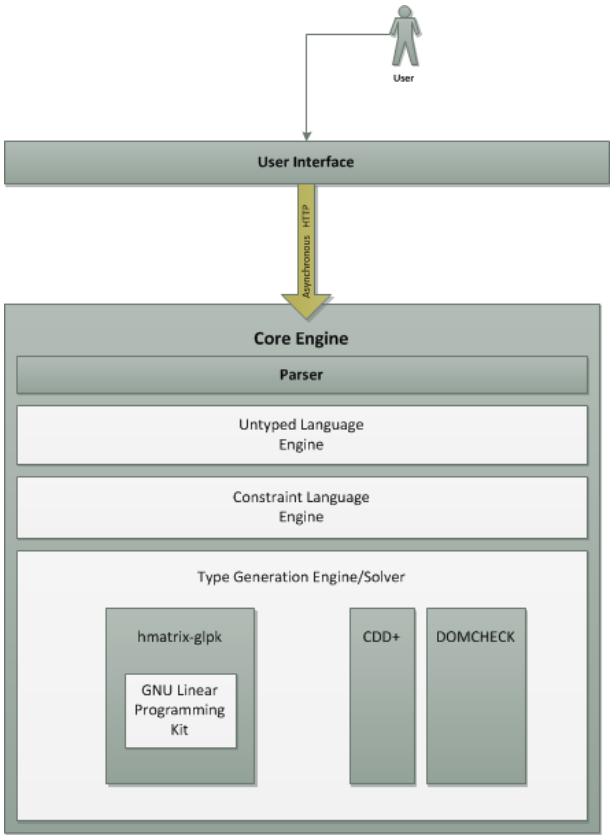


Figure 4. Architecture of NetSketch Tool

The client-server paradigm was employed to allow for a lightweight web deployment, while still retaining a non-browser-resident server component for the linear programming and other computationally heavy tasks. The client

and server communicate over HTTP using AJAX-style requests.

User Interface The user interface was built using pure JavaScript and HTML. Standalone executables offering graphical user interface capabilities were considered (Java, Python), but ultimately a web-based solution was chosen due to a desire for an easily accessible, easily updatable, zero-installation solution. While other web-based platforms (JavaFX, Silverlight, Air, Flash) contain more robust graphical capabilities, it was determined that JavaScript and HTML alone could provide the required GUI capabilities and would avoid attaching the project to a heavyweight proprietary framework.

In order to alleviate some of the burden of ensuring cross-browser compatibility, and development of a rich set of widgets, the ExtJS JavaScript Framework [21] was employed to provide the basic GUI elements. ExtJS is an open source framework that provides a wide array of user interface components as well as JavaScript utilities for DOM (Document Object Model) manipulation, and a simple AJAX model.

In addition to ExtJS, JSGL (the JavaScript Graphics Library) [19], a pure JavaScript vector graphics toolkit, was used. JSGL provided the vector graphics capabilities needed to draw widgets, their ports, and the connections between them. JSGL, as with ExtJS, also servers to hide cross browser incompatibilities.

Core Engine The core of the NetSketch tool is implemented as a server-side component. The server is written in Haskell, with much of the heavy mathematical processing being delegated to external C-based modules, or to an implementation of the Modelica platform. The main executable makes use of the Happstack Web Framework [10]. NetSketch uses the built in HTTP server functionality of Happstack to expose the NetSketch API over the web. HTTP GET requests can be constructed to provide the NetSketch server with the description of the network (including ports, connections, and constraints) in a format based on the domain specific language defined in the work outlining the NetSketch formalism [4].

Once the HTTP server component has received a request it is passed to the *untyped language engine* for parsing. The *untyped language engine* parses the request based on the NetSketch untyped language DSL, and passes the text representing linear constraints to the *constraint language engine*. The grammars for both the NetSketch untyped DSL, and the linear constraint language are defined in annotated BNF. The Haskell parser generator Happy [12] was used to generate parsers based on these grammars. Beyond the parsing functionality, each language engine provides functionality related to the manipulation of its respective language (*e.g.*, simplifying and removing redundancy from linear constraints).

After a successful parse, the structure representing the network described in the request is sent to the *type generation engine*. This module first performs input type generation, followed by output type generation. Input type generation must first project the constraints onto a subset of the original dimensions (specifically those corresponding to the input ports). This projection is done using two external C/C++-based modules: CDD+ [11] and Domcheck [15]. CDD+ is a C++ implementation of the Double Description Method for vertex and extreme ray enumeration. Domcheck is a program that computes minimal linear descriptions of projections of polytopes. These modules are distributed as C/C++ source code. The only modifications made were to Domcheck in order to allow non-interactive execution (*i.e.*, to call in batch without a user present).

Both the output type generator, and the input type generator (after projection) make use of linear programming techniques to identify boundaries of the generated types. The linear programming can be accomplished via one of two mechanisms. The original implementation used a Haskell wrapper, hmatrix-glpk [20], around the GNU Linear Programming Kit (GLPK) [9]. The GLPK is a C-based callable library providing routines for linear programming, mixed integer programming, and other related problems. NetSketch makes use of the GLPK's implementation of the Simplex method. HMatrix-GLPK provides a pure Haskell interface to this and a select set of other features from GLPK. The other mechanism was developed after creating the HModelica Haskell library (see Section 5). In this method NetSketch makes calls via HModelica to an instance of the OpenModelica [17] platform. Here Modelica code is executed to perform the required linear programming tasks. By using OpenModelica 3, external libraries (hmatrix, hmatrix-glpk, and glpk) were no longer required, simplifying the code base.

4. Type Generation

Generating types from sets of untyped modules involves transforming linear constraints into intervals over $\mathbb{R}$. This process is divided into two high-level steps: input port type generation, and output port type generation. As the output type generation can use the results of the input types to create more accurate results, these sub-processes are performed in the order listed above.

Input Port Type Generation In order to generate types for the input ports of a set of modules, it helps to visualize the set of linear constraints that define the set as a convex hull. Figure 5 shows such a hull in 2-space (*i.e.*, for a set of constraints over two input variables). Here we see four constraints labeled Constraint 1 through Constraint 4. The convex hull formed by their intersection defines the set of feasible input values.

To create intervals for the input variables we need to find a largest enclosed hyper-rectangle[4] within the convex hull. Such an area is not necessarily unique. Various options exists for techniques to select a single typing from among these non-unique hyper-rectangles. On the more expressive and accurate side, options exist such as selecting a subset of the possible enclosed hyper-rectangles and defining the type as their union. Work on the use of dynamic adaptive

[4] In this paper all hyper-rectangles are axis-aligned. For brevity we use the term "hyper-rectangle" to 'refer to "axis-aligned hyper-rectangle" throughout.

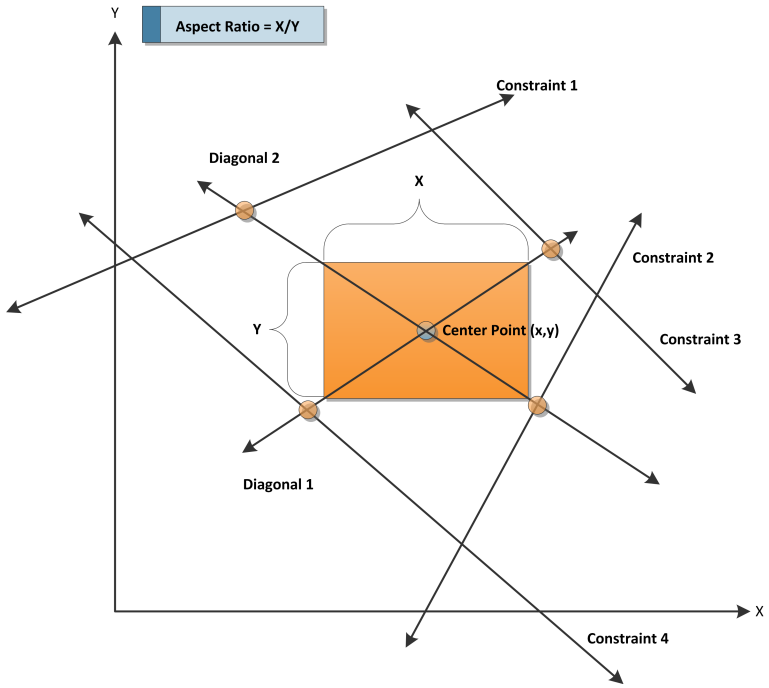


Figure 5. Input Type Generation

types is underway as described in Section 6. For this implementation, a more restrictive process was used that involves defining a center point for the hyper-rectangle, along with an aspect ratio relating all input variables. In Figure 5 the center point (x, y) is displayed along with the aspect ratio relating x to y.

Given a center point and an aspect ratio, a unique maximally enclosed hyper-rectangle can be identified given the set of linear constraints for the modules. Intuitively this can be visualized (in 2 or 3-space) as enlarging a hyper-rectangle (that begins as a single point at the given center point) in increments defined by the given aspect ratio until the hyper-rectangle intersects with the convex hull defined by the linear constraints of the module set. Programmatically, this is accomplished by determining the set of diagonals defined by the hyper-rectangle (labeled Diagonal 1, and Diagonal 2 in Figure 5). There exist 2^{n-1} such diagonals for an n-dimensional hyper-rectangle. Given the center point and aspect ratio of the desired hyper-rectangle, expressions describing the diagonals can be created trivially in parametric form (which the system later converts to the standard linear equation form for use with an existing linear programming solver). With these diagonals defined, the closest intersection (to the center point) with the given linear constraints is then located using linear programming. Four of the eight potential intersection points in Figure 5 are highlighted with circles. Once the closest intersection point $I_{x,y}$ is identified a hyper-rectangle of dimensions $|I_x - C_x|$ by $|I_y - C_y|$ centered at $C_{x,y}$ can be defined. The bounds of this hyper-rectangle on any given axis represent the bounds of the interval for that axis's variable. This example was given in 2-space for visual clarity, but the principles extend to n dimensions where $n \geq 2$ (special case coding exists to handle $n = 1$).

The discussion to this point has assumed that we already have the set of linear constraints to use when generating the input type. It must be noted, however, that the set of linear constraints defined by the user does not equal the set used for these constraints. This is the case for two reasons. First, the set of linear constraints defined by the user does not explicitly contain the equality constraints requiring connected ports between modules to equal each other. These constraints are implied in the visual connections drawn between modules, but made explicit in the inner workings of the NetSketch tool. Secondly, when specifying the set of linear constraints for a given module, the user may well define constraints relating the input and output ports. The generation of the maximally enclosed hyper-rectangle as described above requires the constraints to be restricted to only contain variables from the input ports. To accommodate this need the NetSketch tool first performs a projection of the given constraints, plus the implicit connection constraints, onto only those dimensions representing the input variables. For example, given input ports $I = \{a, b, c\}$, output ports $O = \{x, y, z\}$, and a set of linear constraints C over $I \cup O$, the system will project C onto the 3-dimensional space of I. The resulting constraint set is used in the generation of the maximally enclosed hyper-rectangle.

Output Port Type Generation As with input type generation, it is helpful to visualize the linear constraints as forming a convex hull as depicted in Figure 6. To determine the feasible output values, unlike the maximally *enclosed* hyper-rectangle needed for input ports, a minimally *enclosing* hyper-rectangle must be identified. The determination of this hyper-rectangle is significantly simpler than for that of its input counterpart: an optimal enclosing is unique, so a center point and aspect ratios are not required.

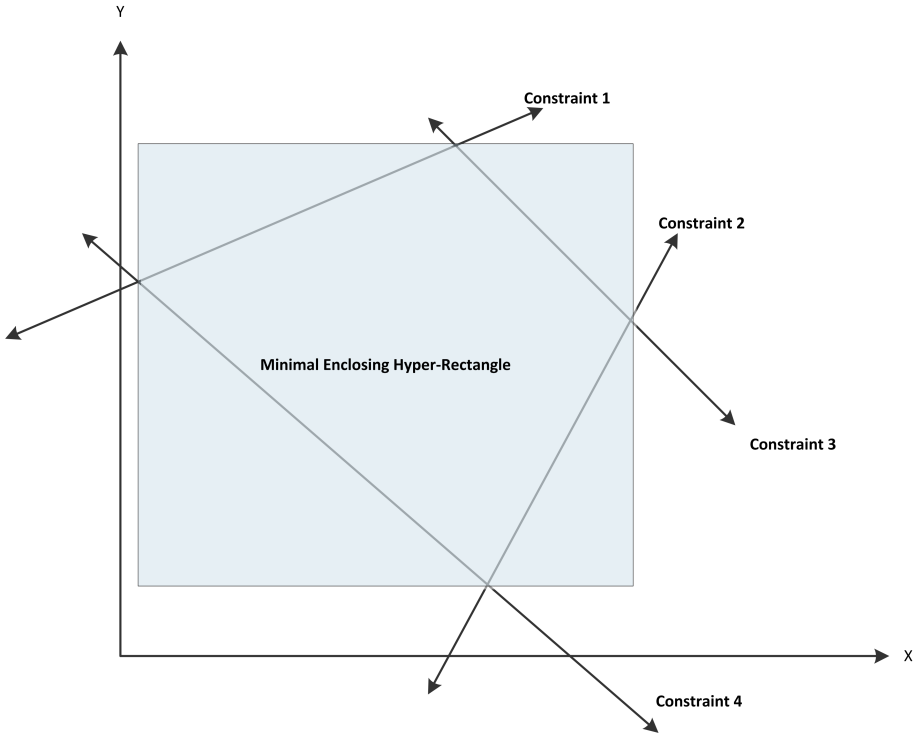


Figure 6. Output Type Generation

The hyper-rectangle can be computed by using linear programming to solve the system of equations and inequalities, first with the objective function $\mathrm{Maximize}(v)$, then again with the objective function $\mathrm{Minimize}(v)$ for each output variable v. The solution that maximizes v will become the upper bound for the variable's type, and the solution that minimizes v will become the lower bound (*i.e.*, $\forall v \in \mathcal{I}, \mathrm{type}(v) = [\mathrm{Solution}_{\mathrm{Min}}, \mathrm{Solution}_{\mathrm{Max}}]$).

As mentioned previously, the constraints used when calculating the output types should include those generated as the input types. The intervals created during input type generation are therefore converted into simple linear constraints (*e.g.*, $x : [0, 100]$ becomes two constraints: $x \geq 0$, and $x \leq 100$). These constraints are then added to the original constraints for use in determining the output types. Without these extra constraints, the result would be correct, but the range of values for the output types would be wider than they truly need to be: in all but the most pathological cases, the valid input values will have been restricted during conversion to intervals.

5. Harnessing Modelica

Well-established constraint-based modeling systems exist today. NetSketch shares a variety of similarities with these tools, but also bears numerous non-trivial differences. Notably, NetSketch in its current form does not explicitly consider time. Other constraint-based modeling tools, such as Modelica [2], are largely centered around time and use simulation over time as their main form of analysis. Some work has been done to show that a variation of NetSketch can be created to more natively incorporate the concept of time. Here, variables of the constraints are replaced by functions of the same name that accept a time variable as an argument. Given the simulation-based nature of Modelica and similar systems, other differences from NetSketch arise, such as the need for balanced systems over equations (rather than inequalities) [1].

Despite such differences, the overlap that does exist offers a great opportunity for various forms of integration. Here, we examine two forms of relation to Modelica: as a computation platform, and as an environment for working with translated NetSketch models.

5.1 Modelica as a Computation Platform

Modelica offers a wealth of functionality as well as a robust library. The extensive library provides both reusable models and reusable functions spanning many domains. This library can be of use to both NetSketch modelers (see sections 5.2 and 6), and to the NetSketch tool implementation itself.

Modelica and the functions defined in the Modelica library can be used directly by the NetSketch implementation as a processing engine. For example, the NetSketch engine requires frequent use of linear programming techniques, namely the simplex method. A function implementing this has been defined in Modelica code and can therefore be used by NetSketch to "farm out" some of the more mathematically heavy computations.

To gain access to the power of Modelica from within the NetSketch tool, a reusable Haskell library was developed to expose the functionality of the OpenModelica implementation to Haskell code. This library, HModelica, enables Haskell developers to create, manipulate, and simulate Modelica models, in addtion to directly executing func-

tions written in the Modelica language. Through the use of this library the NetSketch simplex code was replaced with calls to OpenModelica, alleviating the need for a handful of Haskell- and C-based libraries that previously were tasked with this work. Having a single platform and access mechanism for performing these types of tasks simplifies the NetSketch code base, and this impact will continue to grow as the set of tasks handed to Modelica increases.

The library exposes the OpenModelica API in two ways. The primary mechanism is in place for a subset of the OpenModelica API calls. These functions are implemented as type-safe calls with full translation to and from Haskell types. Second, for any functions not implemented in this manner (the number continues to decrease as development continues), a single function is implemented allowing the caller to send commands to Modelica as a string, and then to receive the results as a string. This allows for the execution of any arbitrary Modelica command.

HModelica has the potential to open Modelica up to the community of Haskell developers. As such, its use can extend outside of NetSketch. To that end the library is being added to the Haskell package repository HackageDB [8]. Here, it will be available for public download and use in the Cabal package format.

5.2 Translation to Modelica

Modelica and NetSketch share enough in common that a translation between the two can be defined. Here, we concentrate on the translation from NetSketch to Modelica; however, a subset of the models developed in Modelica (those with linear constraints) could be directly translated into typed NetSketch networks. This would provide NetSketch users with access to a wider array of pre-built components. A translation in this direction would map Modelica classes and related definitions to NetSketch module definitions with connections between classes and compositions of modules accomplished via NetSketch *Connect* and *Loop* constructs. A formal definition of such a mapping is being considered for future work, as discussed in Section 6.

The reverse direction, a translation from NetSketch to Modelica, generates models that can be used to perform simulation as a safety analysis tool. This process is outlined in detail in an accompanying work [22] and is described at a high level here. To accomplish a translation, two restrictions must be placed on the model during the process. First, any inequalities defined in the NetSketch constraints must be transformed to a form of validation check, as opposed to an active regulator of the system (as the equations section of a Modelica model must contain only that - equations). In some models this may require a binding of a subset of the variables involved in the constraints to specific values for a given simulation of the system (to allow the simulation to uniquely determine the flow). Second, the system must be balanced (not over- or underdetermined). This again may result in the binding of particular variables to concrete values for a given run of the simulation. In these cases single simulations can be run to test "what-if" scenarios corresponding to the particular binding given to the variables, or a set of simulations may be run on the extremes of the valid range of values for each given variable to determine a broader notion of safety across those ranges. Only the extremes of the intervals must be tested because the constraints in the current implementation are linear and thus form a convex hull; no gaps in safe ranges may exist.

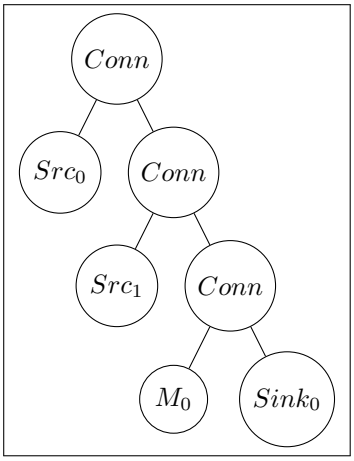


Figure 7. Tree view of the NetSketch network depicted in Figure 8

To reduce the number of variables that must be bound to concrete values, the NetSketch model is first analyzed to construct a minimal covering set. Such an analysis defines a set of variables $\mathcal{S}_{\mathrm{Min}} \subseteq I \cup O$ where I and O represent the set of inputs and outputs, respectively, of the system.

As an example consider Figure 8. Here 6 variables, a, b, c, d, e, f, and a constraint set exist to regulate flow within the system. Since M_0 conserves flow via the constraint $c + d = e$, we need only bind two variables, namely a, and b, to concrete values in order to determine the entire system. Since c, d, e, and f all depend on a and b to determine their values, these variables need not be considered when providing concrete values to drive a Modelica simulation.

An accompanying report [22] defines two algorithms for constructing $\mathcal{S}_{\mathrm{Min}}$. The first is quite efficient, involving two passes of the tree representing a NetSketch model (see Figure 7 for an example), but may not always produce the minimal set. It is causal in nature, and thus does not consider the potential positive impact of variables down the causal chain of the network. The first pass builds two transition relations, and the second actually constructs $\mathcal{S}_{\mathrm{Min}}$ using a set of formal rules and the transition relations from the first pass. A Haskell implementation of this process has been created and will be incorporated directly into the existing implementation as described in Section 6. The second algorithm described in [22] will always produce a minimal set, but has a worst-case exponential running time under a naive implementation. This algorithm transforms a system into a set of propositional logic implication statements representing how knowledge about one variable (or set of variables) implies knowledge about others. The problem is thus transformed into a search for the minimal number of propositional atoms that must be explicitly bound to $true$ in order to imply the conjunction of atoms representing all

variables in the system. A hybrid approach is also described that allows for the use of the first algorithm to set a maximum size of $\mathcal{S}_{\text{Min}}$ from which the second can start. This variant allows for significant savings in computation time.

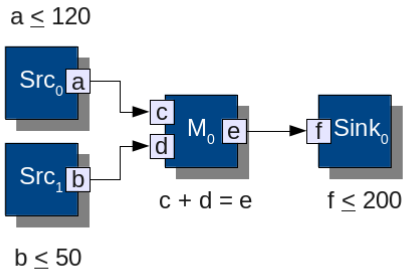


Figure 8. Two source modules, a merge, and a sink.

Once a minimal covering set is constructed, a translation can occur. This again involves a traversal of the tree representing the NetSketch model. Here, as each *Module*, *Hole*, *Conn*, *Loop*, and *Let* node is visited, an abstract representation of a Modelica model is incrementally constructed. NetSketch modules (and holes) are transformed into entities representing Modelica class definitions (or a restricted version thereof) with any equation-based constraints represented directly in the `equation` section of the resulting class definition. For all variables in $\mathcal{S}_{\text{Min}}$ the Modelica `parameter` modifier is used. Inequality constraints are moved to a "driver" class created to organize the system and provide validation checks that the model is safe. Within the driver class all modules/holes are present as instances of their respective classes. Appropriate initial value equations for the variables in $\mathcal{S}_{\text{Min}}$ are present with user-specified bindings. Modelica `connect` statements are used where NetSketch *Connect* and *Loop* constructs existed. The driver is thus a flat representation of the network. The driver also contains a single additional boolean variable, not present in the initial model: *isValid*. This variable is set to equal the conjunction of all the inequality constraints that existed in the individual modules/classes (as these could not be included in the `equation` sections of their owning classes). In this way a user can examine this variable post-simulation to determine if the model is safe under the given parameters. The resulting abstract representation is then transformed into a string which can be written to a text file, or sent directly to Modelica via the HModelica interface described above.

6. Related and Future Work

This work extends and generalizes our work in TRAFFIC [3], and complements our earlier work in CHAIN [7]. An essential functionality of NetSketch is the ability to reason about, and find solution ranges that respect, sets of constraints. In its general form, this is the widely studied *constraint satisfaction problem*. NetSketch types are linear constraints, and linear constraint satisfaction is a classic problem for which many documented algorithms exist. A distinguishing feature of NetSketch and the underlying formalism is that it does not treat the set of constraints as monolithic. Instead, a tradeoff is made in favor of providing users a way to manage large constraint sets through abstraction, encapsulation, and composition. Other formalisms and methods, such as [16], seek to enable early detection of problems in a model by applying types to constraint sets in a modular way, but are intended for providing assurances that compilation prior to analysis/simulation will succeed. In contrast the use of types in NetSketch directly support the analysis of the model itself.

NetSketch leverages a rigorous formalism for the specification and verification of desirable global properties while remaining ultimately lightweight. By "lightweight" we mean to contrast our work to the heavy-going formal approaches – accessible to a narrow community of experts – which are permeating much of current research on formal methods and the foundations of programming languages (such as the work on automated proof assistants [18, 13], or the work on calculi for distributing computing [6]). In doing so, our goal is to ensure that the formalisms presented to NetSketch users are the *minimum* that they would need to interact with, keeping the more complicated parts of these formalisms "under the hood".

A number of planned areas of future work exist related to extending the functionality of the NetSketch tool to more closely match the power expressed in the NetSketch formalism, furthering integration with existing equation-based modeling tools, and extending the formalism itself.

Tool Enhancements Network *Holes* can currently be used with the *Let* construct to select elements from the library to act as hole replacements. Future versions of the tool will allow for selection from additional sources (the current canvas, persisted models, etc). Currently, variables within constraints must be classified as input or output variables. In future implementations, internal variables will be allowed that do not correspond to ports of the module.

NetSketch models the direction of data flow explicitly (*i.e.* ports are marked as either input or output). By default in Modelica's acausal system this is not the case. While NetSketch requires all ports to be causal, bidirectionality can be modeled through either the use of two connections - each representing a direction of flow, or by allowing flow across a single connection to be either positive or negative. Connecting an output port to an input port in NetSketch requires the former be a subtype of the latter. This imples that bidirectional flow over a single connection would require the participating ports have identical types. The NetSketch formalism allows for both of these methods of modeling bidirectionality, though extensions to the current implementation may make such modeling more accesible and transparent. Single connection bidirectionality may benefit, for example, from the extension of the current system's strictly linear constraints to include constructs such as the absolute value function.

Tool Integration As described in Section 5, Modelica offers a wealth of reusable components. Formally defining a translation from a Modelica model to NetSketch would

allow NetSketch users to quickly make use of the breadth of components developed for the Modelica platform. The translation is restricted in the current implementation to a simplified subset of models with linear equations. It should be noted, however, that the restriction to linear constraints is an artifact of the implementation, and not the formalism. The NetSketch formalism is parameterized by the chosen constraint space, and thus allows for a much more general set of constraints than the current tool implements.

The algorithms defined in Section 5 will be integrated into the current tool implementation to allow in-tool exports of NetSketch models to Modelica models. Direct execution of the resulting models will also be implemented as a function of the tool.

Formalism A deeper examination of the proper model for selecting among optimal typings is currently underway and will likely lead to an alteration of both the tool (in its current requirement for a center point and aspect ratio), and potentially of the formalism. Enhancements to the type system to allow for the expression of types as unions of intervals or as function of the state of the network connections is being explored. In addition, work is currently being undertaken on a version of the formalism that restricts the constraints to a particular subset of linear equations resulting in a simplified type inference mechanism, and an expanded set of tractable forms of analysis, while still allowing for an expressive constraint language with real-world applicability. Papers describing the formalism [4], as well as related papers [5, 14] outline a number of additional ideas for furthering the core concepts behind NetSketch.

References

[1] Modelica Association. Modelica Language Specification 3.2. Technical report, Modelica Association, 2010. http://www.modelica.org/documents/ModelicaSpec32.pdf.

[2] Modelica Association. Modelica and the Modelica Association. https://www.modelica.org/, May 2011.

[3] Azer Bestavros, Adam Bradley, Assaf Kfoury, and Ibrahim Matta. Typed Abstraction of Complex Network Compositions. In *Proceedings of the 13th IEEE International Conference on Network Protocols (ICNP'05)*, Boston, MA, November 2005.

[4] Azer Bestavros, Assaf Kfoury, Andrei Lapets, and Michael Ocean. Safe Compositional Network Sketches: Formalism. Technical report, Department of Computer Science, Boston University, Boston, MA, USA, 2009. Tech. Rep. BUCS-TR-2009-029, October 1, 2009.

[5] Azer Bestavros, Assaf Kfoury, Andrei Lapets, and Michael Ocean. Safe Compositional Network Sketches: Tool and Use Cases. Technical report, Department of Computer Science, Boston University, Boston, MA, USA, 2009. Tech. Rep. BUCS-TR-2009-028, October 1, 2009.

[6] Gérard Boudol. The π-calculus in direct style. In *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, pages 228–241, 1997.

[7] Adam Bradley, Azer Bestavros, and Assaf Kfoury. Systematic Verification of Safety Properties of Arbitrary Network Protocol Compositions Using CHAIN. In *Proceedings of ICNP'03: The 11th IEEE International Conference on Network Protocols*, Atlanta, GA, November 2003.

[8] Hackage Community. Hackagedb. http://hackage.haskell.org, May 2011.

[9] GNU Project Developers. GLPK GNU Project. http://www.gnu.org/software/glpk/, January 2011.

[10] Matthew Elder and Jeremy Shaw. Happstack - A Haskell Web Framework. http://happstack.com/index.html, January 2011.

[11] Komei Fukuda. cdd and cddplus homepage. http://www.ifor.math.ethz.ch/~fukuda/cdd_home/cdd.html, January 2011. Swiss Federal Institute of Technology.

[12] Andy Gill and Simon Marlow. Happy - The Parser Generator for Haskell. http://www.haskell.org/happy/, January 2011.

[13] Hugo Herbelin. A λ-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *"Proc. Conf. Computer Science Logic"*, volume 933 of *LNCS*, pages 61–75. Springer-Verlag, 1994.

[14] Andrei Lapets, Assaf Kfoury, and Azer Bestavros. Safe Compositional Network Sketches: Reasoning with Automated Assistance. Technical report, Department of Computer Science, Boston University, Boston, MA, USA, 2010. Tech. Rep. BUCS-TR-2009-028, January 19, 2010.

[15] Francois Margot. Francois Margot Homepage. http://wpweb2.tepper.cmu.edu/fmargot/, January 2011. Carnegie Mellon.

[16] Henrik Nilsson. Type-based structural analysis for modular systems of equations. In *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, July 2008.

[17] Open Source Modelica Consortium (OSMC). Welcome to OpenModelica. http://www.openmodelica.org/, May 2011.

[18] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume LNCS 828. Springer-Verlag, 1994.

[19] Tomas Rehorek. JavaScript Graphics Library (JSGL) official homepage. http://www.jsgl.org/doku.php, January 2011.

[20] Alberto Ruiz. HackageDB: hmatrix-glpk-0.2.1. http://hackage.haskell.org/package/hmatrix-glpk, January 2011.

[21] Sencha. Sencha - Ext JS - Client-side Javascript Framework. http://www.sencha.com/products/js/, January 2011.

[22] Nate Soule, Azer Bestavros, Assaf Kfoury, and Andrei Lapets. Safe Compositional Equation-based Modeling of Constrained Flow Networks. Technical report, Department of Computer Science, Boston University, Boston, MA, USA, 2011. Tech. Rep. BUCS-TR-2011-014, June 5, 2011.

[23] Nate Soule, Azer Bestavros, Assaf Kfoury, and Andrei Lapets. Use Cases for Compositional Modeling and Analysis of Equation-based Constrained Flow Networks. Technical report, Department of Computer Science, Boston University, Boston, MA, USA, 2011. Tech. Rep. BUCS-TR-2011-019, July 5, 2011.

A Compositional Semantics for Modelica-style Variable-structure Modeling

P. Pepper[1] A. Mehlhase[2] Ch. Höger[3] L. Scholz[4]

[1]Institut für Softwaretechnik, TU Berlin, Germany, {peter.pepper}@tu-berlin.de
[2]Institut für Softwaretechnik, TU Berlin, Germany, {a.mehlhase}@tu-berlin.de
[3]Institut für Softwaretechnik, TU Berlin, Germany, {christoph.hoeger}@tu-berlin.de
[4]Institut für Mathematik, TU Berlin, Germany, {lscholz}@math.tu-berlin.de

Abstract

Modelica traditionally has a non-compositional semantic definition, based on so-called "flattening". But in the realm of programming languages and theoretical computer science it is by now an accepted principle that semantics should be given in a compositional way. Such a semantics is given in this paper for Modelica-style languages. Moreover, the approach is also used to consider more general modeling concepts, namely so-called variable-structure systems. As an outlook we discuss the correspondence between such an idealized mathematical semantics and a more pragmatic numeric solver-oriented semantics.

Keywords Modelica, compositional semantics, structure dynamics, uncertainty.

1. Introduction

A large class of technical multi-physics systems can be modeled in languages like Matlab/Simulink/Stateflow, Ascet, Labview, Scicos, Modelica and various others. These systems are essentially based on the paradigms of control theory and differential-algebraic equations (DAEs). Of these systems, Modelica is distinguished by the fact that it integrates the control theoretical aspects with the software engineering principles of object-oriented programming. We consider this a major advantage and therefore concentrate in this paper on Modelica-style modeling.

However, there are a number of modeling concepts that are also missing from Modelica, most notably the so-called *structure dynamics*, also referred to as *variable-structure modeling*. Roughly speaking this means that during its lifetime the system passes through *modes*, in which it obeys different sets of (differential) equations. This paradigm is very helpful in the "natural" specification of many systems, but unfortunately it adds a number of severe conceptual, semantic and implementation problems to classical Modelica. Nevertheless, it has already been – at least partly – addressed in a few systems, notably Mosilab [21] or SOL [29]. Structure dynamics is also possible in languages like Hydra [24, 20, 7, 19] or frameworks like CIF (Compositional Interchange Format) [25, 26, 27]. However, we do not consider such languages or frameworks here, since they are not Modelica-like languages; rather Hydra is a Haskell-based functional language that achieves similar effects with other means (see below); the same is true for MKL [3, 4]. And CIF is a framework that shall integrate all kinds of languages and language paradigms.

We note in passing that variable-structure systems can also be treated in tools like Simulink and Dymola, but only efficiently by way of using scripting techniques and only for systems consisting of a few different modes [16].

In this paper we consider the core principles of variable-structure modeling and try to give a rigorous semantics for them. This semantic treatment bases on ideas taken from approaches such as Phase Transition Systems [14, 15] and Hybrid Automata [11, 8], the latter of which in turn are based on StateCharts [10]. Moreover, we also integrate ideas taken from ESpec [22, 12] as well as from Lustre [5], Ptolemy [23] , CIF and comparable systems. However, since all these approaches and concepts are overlapping, we will not make any attempt to attribute each of our individual design decisions to their respective predecessors.

When comparing formal approaches such as Phase Transition Systems to Modelica-style systems, one can immediately see a fundamental difference in attitude, which makes any semantic comparison almost vain from the beginning: Whereas the former bases on clean mathematical concepts of the real numbers $\mathbb{R}$ and their function space, the latter immediately focuses on numeric solvers and all problems that are coming with them. In order to bridge that gap, we envision a two-level semantics:

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zürich, Switzerland.
 Proceedings available at:
http://www.ep.liu.se/ecp/056/
EOOLT 2011 website:
http://www.eoolt.org/2011/

1. *Ideal semantics*. The first semantics that we give lives in the ideal mathematical world of real numbers, continuous functions, dense time and the like.
2. *Simulation semantics*. The second semantics is derived from the first one by taking issues such as discretization, approximation, rounding errors and the like into account. However, we should emphasize that this is still work in progress.

Through this separation of concerns we do not mix up different and unrelated kinds of aspects in a single monolithic description.

In the realm of programming languages and theoretical computer science it is by now a well established principle that the semantics of languages should be given in a compositional way. This means that the global semantics of the whole can be described by giving local semantics to the parts, from which the overall semantics is derived by suitable composition rules. (This is in accordance with the principle of modularization in software and systems engineering.)

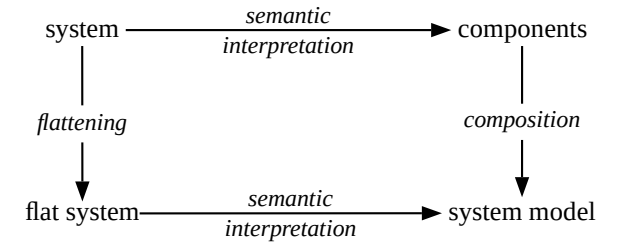


Figure 1. Compositional semantics

Figure 1 illustrates the relationship between compositional and non-compositional semantic definitions. The standard definition of Modelica follows the lower left path: One always considers the system as a whole and flattens it into one large system of equations, which represents its semantics. However, in the realm of programming languages one rather gives individual semantic definitions for all components and then composes the overall semantics from these local definitions. This is the approach, which we will follow in this paper.

Note: In the literature one can also find attempts to give semantics to a continuous modeling language by designing it as an embedded DSL (domain-specific language) over some existing host language. For example, Hydra [20, 7, 6] uses Haskell as the encompassing host language. Also the approaches taken in CIF, which in turn bases on earlier work on Chi, or Ptolemy [23] can be subsumed under this principle. Here the host for the embedding is not a real programming language such as Haskell; rather one employs a powerful mathematical framework (such as special automata with SOS-based semantic definition) into which the modeling language is mapped.

We do not follow this idea here, since it adds considerable complexity to the understandability of the semantics. One first has to fully comprehend the host language – which is not trivial for a language like Haskell or CIF – and then one has to understand the embedding. This may be acceptable for computer scientists, who want to enter the field of continuous modeling, but it is unacceptable for engineers, who are familiar with areas like control theory and possibly can program in C or Fortran. For this audience, Haskell poses an insurmountable obstacle in practice. Therefore we prefer to give an independent semantics to modeling languages such as Modelica, which strives for simplicity and easy understandability.

Part I: Ideal Semantics

2. Preliminaries: Specifications and Models

In the realm of programming languages and their semantics there is by now a well-established way of proceeding. To begin with, one has to clarify what kinds of terms are syntactically allowed. To this end we follow the principles that have been laid down very precisely in the realm of algebraic specifications. (For a comprehensive treatment see [1]; we follow here mainly the approach taken in the Espec system [22, 12].) But we have to adapt the pertinent concepts to the new situation of differential-algebraic equations.

The starting point is a *signature* Σ that determines the admitted types and operations. In our case this signature essentially consists of the standard operations of real arithmetic. (At least we focus on that part of the signature.) As a special feature the signature contains the differential operator $\frac{dx(t)}{dt}$, which we usually denote as $\dot{x}$ or x'.

Over such a signature Σ and a set X of (typed) variables one can then build the set $T(\Sigma; X)$ of well-typed *terms*. And over these terms one can then build *equations*. Due to the availability of the differential operator, these equations are actually so-called *differential-algebraic equations*, short: DAEs.

Such a signature Σ together with a set E of equations over Σ is called a *specification* (in formal logics sometimes also *theory presentation*). As is well-known from algebra and formal logics, such a specification in general has many *models*, where the notion *model* refers to a mathematical structure (usually an algebra) that is conformant to the signature and obeys the equations.[1]

Definition 2.1 (Specification, model) *Let $S = (\Sigma, E)$ be a specification. The set of* ***models*** *of this specification is denoted as Mod$(S) = \{\ A \mid A \models S\ \}$.* $\Diamond$

Note that we include in the notation $A \models S$ not only the validity of the equations but also the conformance to the signature. If the signature is clear from the context, we also write $A \models E$.

3. Fixed-structure Systems

We first consider fixed-structure systems. This kind of system corresponds essentially to the style of models that can

[1] It is unfortunate that the word "model" occurs here in two conflicting meanings. On the one hand, there is the terminology of "continuous-system modeling" and on the other hand there is the formal-logic terminology of "models" of theories. We hope that the two usages will always be clear from the context.

be formulated in Modelica. These models are given as a hierarchy of subsystems that are ultimately based on atomic components. Since these subsystems are usually encapsulated into some suitable context, they look like components themselves.

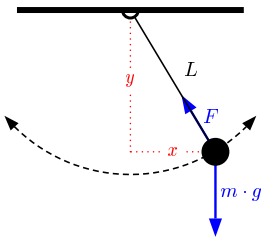
Definition 3.1 *A **system** is built up from interacting components. These **components** are in turn either* subsystems *or* atomic components. ◊

Modelica follows the approach of object-oriented languages and describes components and systems by way of *classes*. Such a class can be seen as a "blueprint", from which at runtime concrete components ("objects") are obtained by instantiation. In the terminology of algebra (see Section 2) these classes correspond to the specifications and the components to models (in the sense of formal logic). In the following we will often just speak of "components" or "systems". It will be clear from the context, whether we are talking about the classes or about the instances.

We present our semantic definition in a bottom-up fashion. That is, we start from atomic components and then compose them into larger systems.

3.1 Atomic Components

The most elementary kind of system is an *atomic component*. As is illustrated by the example in Figure 2 such an atomic component consists of parameters and constants, variables and equations. Parameters and constants are an important and convenient feature in practical applications, but they do not play an important role in the semantics. Therefore we will ignore them from now on and only concentrate on the variables.



Class $Pendulum$	
PARAM $Length\ L$	parameters
PARAM $Mass\ m$	
CONST $Acceleration\ g$	
$Length\ x$	variables V
$Length\ y$	
$Force\ f$	
$m \cdot \ddot{x} = -\frac{x}{L} \cdot f$	equations E
$m \cdot \ddot{y} = -m \cdot g - \frac{y}{L} \cdot f$	
$x^2 + y^2 = L^2$	

Figure 2. An atomic component

In general components (more precisely, the classes describing them) are embedded into the context of some encompassing system. According to the usual scoping principles of programming and modeling languages this means that the component has access to the variables of this context. We refer to these variables as *external* variables of the component. These variables are usually not mentioned explicitly in the component but determined implicitly by the scoping rules. In any case we need to split the set of variables into the two sets of "external" and "local" variables.

Definition 3.2 *An **atomic-component class** is a named triple $C = (V_e, V_l, E)$ consisting of two disjoint sets V_e and V_l of* external and local variables *and a set E of* hybrid differential-algebraic equations (DAEs). ◊

As mentioned in Section 2 such a class in general represents a whole set of possible models. (This is often referred to as "underspecification".) That is, there are many components that fulfill the equations of the class.

In our application of continuous-system modeling the variables of the class are interpreted as time-dependent functions. For example, in Figure 2 the variable x stands for a function $x(t)$ that gives the x-position of the pendulum at any given point in time t.

Definition 3.3 *Let $C = (V_e, V_l, E)$ be an atomic-component class. A **model** for this class is a structure $M = (F_e, F_l)$ consisting of two sets of time-dependent functions, which are in one-to-one correspondence to the sets V_e and V_l of variables. These functions have to fulfill the equations in E. We denote this as usual by $M \models C$. By Mod(C) we denote the* set of all models *of the class C.* ◊

Note that the functions are completely general, that is, they may be continuous or discontinuous or even discrete, that is, only defined at selected time points. In this way the set of equations E may also contain equations that determine the initial values.

In practice the one-to-one correspondence between the variables and the functions is simply established by using the same names for both, that is, e.g. x for the variable and $x(t)$ for the function. However, as we will see in a moment, there may be many instances (objects) of the same class coexisting in one system. Therefore we have many different functions that correspond to the same variable, namely one for each instance. In order to resolve these conflicts systematically we assign unique identifiers to the instances of the class (just like in Java every object of a class is identified by a unique "reference"). The function names are then annotated by these identifiers.

For example, if we have two components K_1 and K_2 as instances of a class C, and if x is a variable in C, then the two instances have functions $x_{K_1}(t)$ and $x_{K_2}(t)$. In this way an equation like $\dot{x} = 2 \cdot x$ of the class is interpreted by the two model equalities $\dot{x}_{K_1}(t) = 2 \cdot x_{K_1}(t)$ and $\dot{x}_{K_2}(t) = 2 \cdot x_{K_2}(t)$.

3.2 Composition of Components

Systems are usually built by composition of atomic components. The most elementary form of such a composition is sketched in Figure 3. We have two classes C_1 and C_2 describing components with individual variable sets V_1, V_2

and equation sets E_1, E_2. Since they coexist in the same context, they share the set V_e of external variables. As mentioned earlier, these external variables are usually not declared in the component but derived implicitly by the scoping rules.

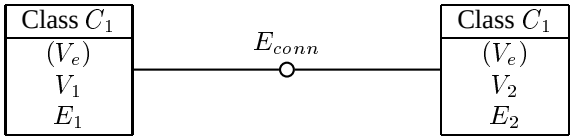


Figure 3. Composition of components

The components are linked to each other by way of so-called *connectors*. From a semantic point of view such connectors only contribute further equations, typically of the form

$$\begin{array}{rcll} C_1.e & = & C_2.e & e\text{: potential/effort} \\ C_1.f + C_2.f & = & 0 & f\text{: flow} \end{array}$$

For the semantics this means that we have to combine all models of the two classes, provided that they coincide on their common external part. This set then needs to be filtered further by the connection equations.

Definition 3.4 (Composition) *Let a system S be given, which consists of the two classes $C_1 = (V_e, V_1, E_1)$ and $C_2 = (V_e, V_2, E_2)$ (sharing the same external variables V_e) together with a set E_{conn} of* connection equations. *Then the models of the* ***combined system*** *are derived from the individual models as follows:*

$$\begin{array}{rcll} \mathit{Mod}(S) & = & (\mathit{Mod}(C_1) \otimes \mathit{Mod}(C_2)) \mid E_{conn} & \\ & \stackrel{def}{=} & \{\, M_1 \cup M_2 \mid & M_1 \in \mathit{Mod}(C_1), \\ & & & M_2 \in \mathit{Mod}(C_2), \\ & & & M_1|_{V_e} = M_2|_{V_e}, \\ & & & M_1 \cup M_2 \models E_{conn} \,\} \end{array}$$

$\Diamond$

Since we assume that the local variables are suffixed by the component names, we can form the union of the models without any danger of name clashes. (The operator $\otimes$ is a pushout in the sense of category theory; that is, we form the direct product of the two sets of models while sharing their common global parts.)

Based on this definition of the semantic meaning we also write the composition of two components in the shorthand form $(C_1 \otimes C_2) \mid E_{conn}$.

This construction generalizes from two to n connected components in the obvious way.

3.3 Subsystems

By connecting n components we obtain subsystems. But in general such a subsystem is embedded into some encompassing component, which contributes both variables and equations. This is illustrated in Figure 4.

Note that this construction is usually embedded into an even larger context. Therefore we have the sets of variables V_e, V_S, V_1, ..., V_n and the equations E_S, E_1, ..., E_n

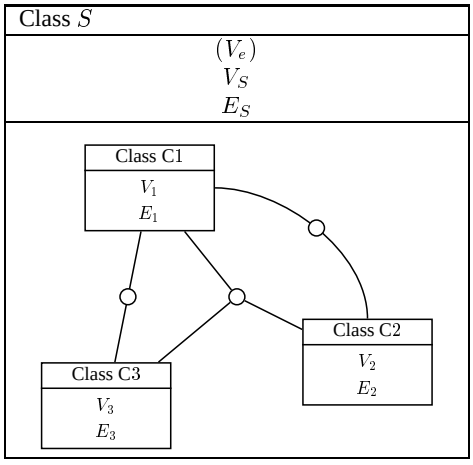


Figure 4. A subsystem

together with the various connection equations $E_{conn_1}, \ldots, E_{conn_k}$. Note also that the subcomponents C_1, C_2 and C_3 have $V_e \cup V_S$ as their set of external variables.

Since the equations in E_S may refer to variables of the components by using prefix notations such as $\dot{x} = f(x, C_1.y, C_2.z)$, we have to respect this in the semantic definition. That is, a class reference such as $C_1.y$ refers on the model level to $y_{K_1}(t)$, where K_1 is the model instance of C_1.

Definition 3.5 *Let a system be given as described in Figure 4. Then the semantics is given by*

$$\mathit{Mod}(S) = \big(\mathit{Mod}(C_1) \otimes \cdots \otimes \mathit{Mod}(C_n)\big) \mid (E_S \cup E_{conn_1} \cup \cdots \cup E_{conn_k})$$

Note that $V_e \cup V_S$ is part of the global variables of each of the C_i such that these variables are taken care of by the operator $\otimes$. $\Diamond$

The construction can be applied iteratively to nested hierarchies of systems. Thus we obtain a bottom-up compositional semantics for Modelica-like languages. This semantics is equivalent to the monolithic flattening-based semantics as it has been sketched in the commuting diagram of Figure 1 in the introduction.

4. Variable-structure Systems

We want to go beyond classical Modelica and also consider variable-structure systems. Such variable-structure systems generalize the effects that can be achieved in Modelica with a combination of the if- and when-constructs or in Matlab/Simulink with the enabling blocks. Vice versa, these constructs can be semantically explained in terms of variable-structure systems. Non-standard Modelica systems such as Mosilab [21] or experimental designs such as SOL [29] already provide principal implementations of this idea.

In the fixed-structure systems discussed so far we only need to consider a single time interval, namely the *life span* of (the simulation of) the system. All components are created at the beginning of this life span and the functions $x(t)$ are defined (and simulated) over this life span.

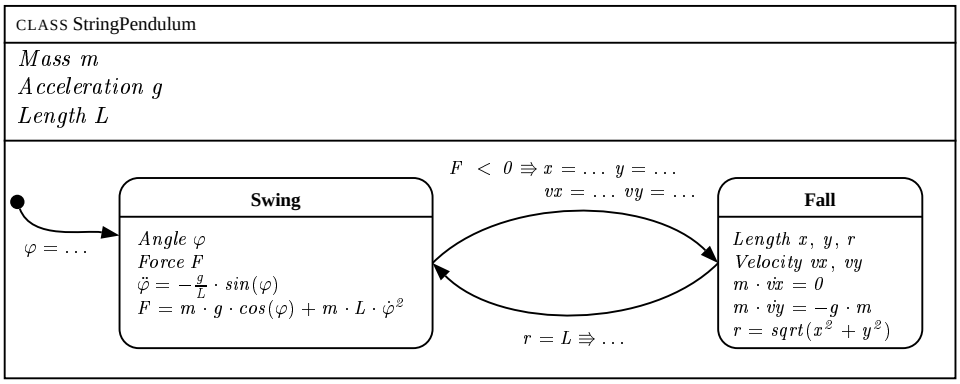


Figure 5. A simple dynamic component

However, it will frequently be the case that a component passes through different modes, in which it exhibits different kinds of behaviors. An example is given in Figure 6 and its specification in Figure 5. Here we consider a string pendulum, which initially starts as a normal pendulum but changes to a free-falling ball as soon as the force F is less than zero.

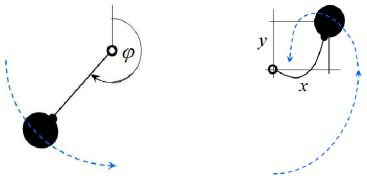


Figure 6. A simple dynamic component

For specifying such variable-structure systems and components we use a notation that is oriented at so-called *Hybrid Automata* [11] which in turn can be seen as a combination of *State Charts* [10] and *Phase Transition Systems* [14, 15]. As can be seen in Figure 5 such a component still has component-external variables and equations. (Actually, in this simple example there are only parameters and constants.) But now it also has modes with additional mode-local variables and equations. Moreover, there are guarded transitions between the modes, which also possess actions that essentially describe, how the initial values of the next mode are to be determined.

We will not indulge further into this slight generalization, since we want to consider even more general settings as described – again by an oversimplified example – in Figure 7. The specification of the two modes is sketched in Figure 8.

Here our initial system consists of two components, a car and a ball. The car exhibits the simple physics of a rolling device on an inclined plane, while the ball only contributes its mass. After hitting the block, there is only one interesting component left, namely the ball, which now follows the physical laws of a bouncing ball.

As can be seen in this example, each mode can contain whole subsystems such that the overall topology of the system under consideration may change dynamically.

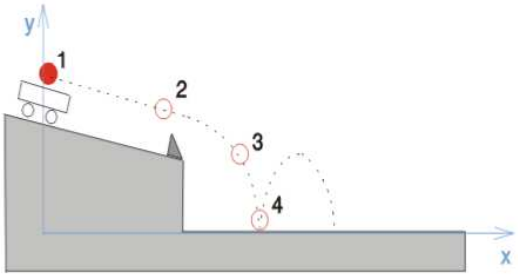


Figure 7. A simple mode-changing system

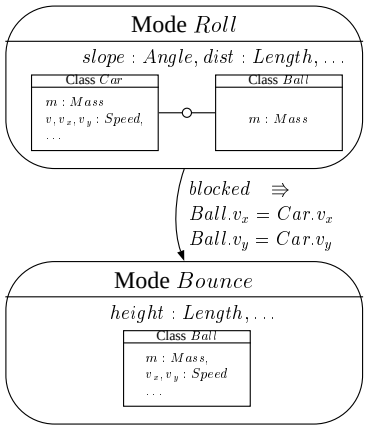


Figure 8. A simple changing topology

This principle is illustrated more abstractly in Figure 9, which shows the general setting.

A system has external variables V, equations E and components $C_1, C_2, C_3, \ldots$. Moreover, it possesses modes $M_1, M_2, \ldots$, which in turn have local variables, equations and subsystems. In order not to overload the pictorial illustration we have refrained in Figure 9 from drawing the connectors between the mode-local subsystems S_{M_i} and the component-subsystem S. Of course, such connectors are possible (and they are easily described in textual form).

This hierarchical structuring can be iterated over arbitrarily many levels.

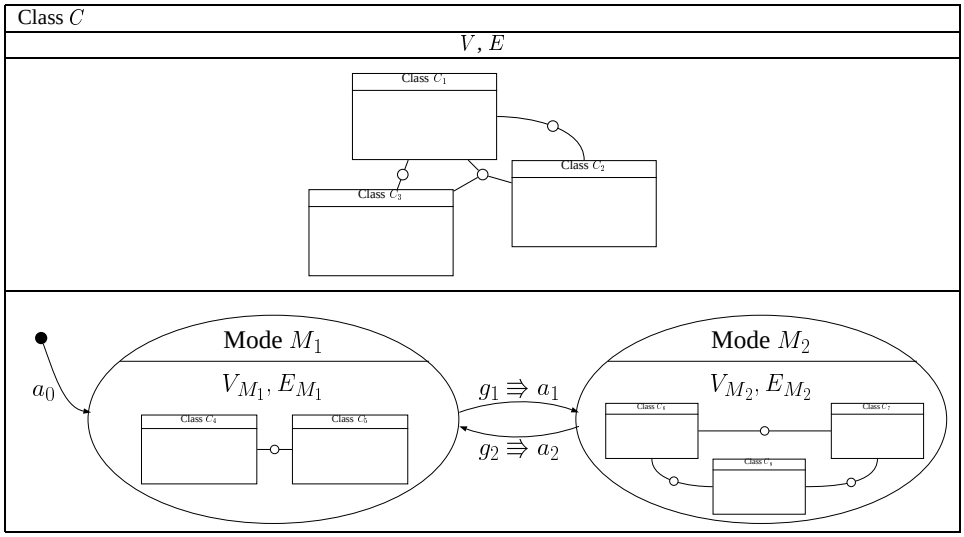


Figure 9. Variable-structure systems

Definition 4.1 (Variable-structure component) *A **variable-structure component** is a named tuple $C = (V, E, S, D)$, where V and E are sets of component-external variables and equations, S is a subsystem, that is, a set of interconnected subcomponents, and D is the **dynamics**, that is, a set of modes and transitions. (The modes and transitions will be defined more rigorously in the next sections.) Each mode $M \in D$ in general comprises mode-local variables V_M and equations E_M as well as a mode-local subsystem S_M of components. The dynamics has an **entry point** t^α and an **exit point** t^ω.* ◊

In the remainder of this paper we will make this informal definition more precise by giving rigorous semantics to its various features.

5. Modes and Time

The situation in a real-world system can be roughly sketched as in Figure 10. The modes (more precisely: instances of modes) follow each other along the time line. The component-external variables correspond to functions that live across the modes; an example is illustrated in Figure 10 by the thick line $f(t)$. The mode-local variables correspond to functions that live only during their mode; examples are given in the figure by the thin lines $g(t)$ and $h(t)$.

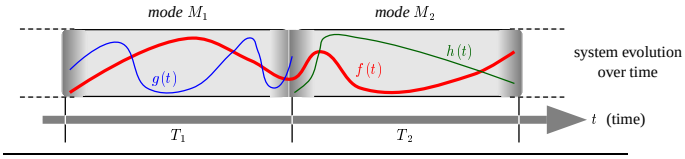


Figure 10. Real-world semantics

Note that we do not put any additional constraints on the functions; that is, within its realm each of the functions may be continuous, possess discontinuities or may even be discrete. In other words, we consider modes to be a *design concept*, not a technical feature that shall handle complications such as discontinuities.

Each mode – more precisely: each *instance* M_i of a mode – is associated to a unique interval $T_i = [t_i^\alpha, t_i^\omega)$ of the time line. For reasons that will be discussed in a moment we use half-open time intervals.

Note: In practical technical systems the transition between modes usually takes some time, in which the system is non-observable. A typical point in case is a diode, for which the switching from *on* to *off* is a very short but continuous process. In the idealized mathematical treatment such transition periods are often modeled as instantaneous transitions (which makes them often discontinuous). It is debatable, whether one might even allow whole transition intervals instead of transition points, or whether one should model such situations by transition modes. Both variations would be mathematically feasible; but for reasons of simplicity we opted for the design with left-closed intervals.

As a final preparatory remark we point out that a fixed-structure system can be considered as a borderline case of a variable-structure system with exactly one mode that spans the whole life time of the system.

Definition 5.1 (Modes) *Let a class $C = (V, E, S, D)$ be given as sketched in Figure 9. The instantiation of this specification (at runtime) leads to the creation of a new component K, which is a model of C. This component has a lifetime $T_K = [t^\alpha, t^\omega)$. The start time t^α is the moment of the creation of the component, the end time t^ω is determined by rules that will be discussed later (e.g. caused by events). We require $t^\alpha < t^\omega$, i.e. we exclude components with zero life time.*

The fixed-structure part $C_{fix} = (V, E, C_1, \ldots C_n)$ of the class is semantically defined as in Definition 3.5. Note that the subcomponents also have the life span T_K.

*The semantics of the system during a **mode** M is derived from the fixed-part semantics by composing it with the mode-local system S_M, that is, $C_{fix} \otimes S_M \mid E_{conn}$ as specified in Definition 3.4. Here E_{conn} are the additional connection equations that link the subsystem of the mode to the fixed-part subsystem.*

The life span of the mode M is determined by rules that will be explained in connection with transitions in the next section. $\Diamond$

Note that this semantic definition entails a subtle aspect. The fixed part $C_{fix} = (V, E, C_1, \ldots C_n)$ of the class in general has a whole set *Mod*(C_{fix}) of models. Since we allow the individual modes to impose further restrictions on the fixed-part variables in $V, V_1, \ldots, V_n$, the behavior of the corresponding functions varies from mode to mode.

Example: Let $x \in V$ be a component variable (without a restricting equation) and let x_1 and x_2 be two local variables of the modes M_1 and M_2, respectively. Let the following equations be given:

$$\begin{array}{lll} M_1: & x_1 = 1 & \qquad M_2: \quad x_2 = 2 \\ & x = x_1 & \qquad\qquad\quad x = x_2 \end{array}$$

Then $x(t)$ is a function, which is constant but different in M_1 and M_2 and has a discontinuous jump between the two modes.

6. Transitions and Events

Next we consider the actual mode transitions as illustrated in the examples of Figure 6 and Figure 7 or more schematically in Figure 9. The principles of such a mode transition are illustrated in Figure 11.

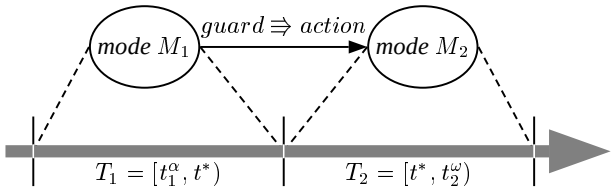


Figure 11. Semantics of transitions

During the execution of the model each mode instance is associated to a time interval. In the situation of Figure 11 we have the two intervals $T_1 = [t_1^\alpha, t^*)$ and $T_2 = [t^*, t_2^\omega)$. The start point of T_1 and the end point of T_2 are of no interest here. The focus of our attention is the point t^*, which represents the transition between (the instances of) the two modes.

Definition 6.1 (Transition point) *Let two modes M_1 and M_2 with a transition $(guard \Rrightarrow action)$ be given as sketched in Figure 11. During runtime the transition point t^* between (instances of) these two modes is defined as follows: t^* is the smallest time instance greater than the entry point t_1^α of mode M_1 such that $guard(t^*) = true$ and $guard(\tau) = false$ for all τ with $t_1^\alpha < \tau < t^*$. Then, $T_1 = [t_1^\alpha, t^*)$ and the entry point of mode M_2 is given by $t_2^\alpha = t^*$.* $\Diamond$

Since modes shall not degenerate to zero length, we have the condition $t_1^\alpha < t^* < t_2^\omega$.

[18] provides a nice discussion of many aspects of such transitions from the point of view of physics. The design with half-open intervals ensures a well-defined solution $x(t)$ for all times t. Nevertheless, there are cases where the left-closedness may have to be relaxed (see also the discussion in [18]). However, these are special cases that certainly have to be dealt with by the solvers, but not an issue of ideal semantics.

Remark 1: The case of the immediate re-firing in the case of self-loops can be repaired by introducing the concept of *superdense time* [15]. (A typical example for such a self-loop is provided by the "bouncing ball".) Here the time is not only $\mathbb{R}$ but the product $\mathbb{R} \times \mathbb{N}$, where the second component could be interpreted as the number of the "clock" that is responsible for the event. Then each occurrence of the event $y = 0$ in the bouncing-ball example could be associated to another clock. Even though this idea of superdense time may turn out helpful in some situations, we currently refrain from introducing it and try to work with our simpler model.

Therefore we would rather consider the immediate re-firing as a modeling error (analogously to infinite while loops in programming). For instance, in the bouncing-ball example the guard should not only be $y = 0$ but $y = 0 \wedge vy < 0$. The transition action (see below) has to set $vy_{new} = -vy_{old}$. When the velocity is zero, the mode should be exited.

Remark 2: There is the possibility of *conflicting guards*. That is, two guards g_1 and g_2 may fire at the same instant t^*. There are several options for treating this situation. One could let the system choose nondeterministically between the transitions. One could avoid this nondeterminism by enforcing priorities between the transitions (like in Matlab/Simulink/Stateflow). Or one could require the disjointness of the guards and report an error, when two of them fire simultaneously. This is a topic for language design. Since each of these options is compatible with our semantic concepts, we can ignore the issue here. We note in passing that there are attempts to utilize the idea of superdense time also for this issue.

So far we have only considered the guards that fire the transitions and thus determine the transition point t^*. We still need to define the meaning of the *action* part of the transition. The purpose of these actions is to provide the *initial values* for the next mode. There are various syntactic means for achieving this effect.

Example: Consider again the bouncing-ball example. When the ball hits the ground, the vertical velocity vy is reversed and deminished by some constant factor c. This could be written in a Pascal-like programming style as $vy := -c * vy$. Or we could use an equation-style notation like $vy_{new} = -c * vy_{old}$, which would necessitate notations for "old" and "new" (which is implicitly contained in the program-style notation by the occurrence of vy on the left or right side of the assignment). Again, the notation is an issue of language design and thus does not concern us here.

But we need to give a semantic meaning to x_{old} and x_{new} for all variables x occurring in M_1 and/or M_2.

Definition 6.2 (Transition action) *Consider the scenario of Definition 6.1. The **action** part of the transition describes a computation that uses (implicitly or explicitly) values x_{old} and x_{new} for certain variables x. Depending on the application, we can define $x_{old} = x(t^*)$ or*

$x_{old} = \lim_{h \to 0} x(t^* - h)$. *$x_{new}$ is computed as a function of x_{old}. Then $x = x_{new}$ is used as the initial-value equation for the differential-algebraic system in mode M_2.* $\Diamond$

This completes the compositional definition of the "ideal" semantics of Modelica-style variable-structure systems.

Part II Simulation Semantics

7. Numerical Integration and Solvers

The ideal semantics lives in the realm of the real numbers $\mathbb{R}$ with infinite precision and infinitely accurate computations. But in reality we have to make do with the limitations of computers, where one struggles with floating-point arithmetic and its rounding errors, with solver techniques realized in packages like DASSL [2] or RADAU5 [9], with causalization and index reduction and so forth. This leads to a different kind of semantics that we baptize "solver semantics" or "simulation semantics".

The reason for dealing with this question is the current situation in modeling languages such as Matlab/Simulink or Modelica. The specification documents of these languages often switch back and forth between concepts that we refer to as "ideal semantics" and the difficulties introduced by the computer-related limitations. Hence it is not always clear, which aspects describe fundamental semantic concepts and which aspects are actually due to shortcomings of certain solver methods. Sometimes one even gets the impression that certain statements actually address specific features of certain compilers or compilation techniques.

7.1 Solver Issues

Solver semantics differs from the ideal semantics primarily in two respects: discretization and finite precision.

1. *Discretization*. Functions over the continuous time domain $\mathbb{R}$ are replaced by functions over discrete time points t_i, where $t_{i+1} = t_i + h_i$ for some discretization step size h_i, starting at the initial time point t_0; the step size may be uniform or varying.

   It should be noted that the principle of discretization is still compatible with ideal real-number arithmetic over $\mathbb{R}$. In the Haskell-oriented literature this is usually modeled by *streams* of sample values $f(t_i)$. For example, in [28] it is shown that this approximation is – under certain constraints – faithful in the limit, as the step size goes to zero. (This is not really surprising in the light of century-old work by mathematicians like Cauchy and others, since their techniques are essentially translated here into Haskell-speak.)

2. *Precision*. The real numbers $\mathbb{R}$ are replaced by machine numbers of limited size, usually `float` or at best `double`, on some processors even by fixed-point emulations of floating-point numbers. This limited precision combined with the need for not only finite but actually efficient computation leads to several kinds of errors:

   - The use of limited machine numbers leads to the well-known *rounding errors*.
   - The numerical solution $\tilde{x}$ obtained by the solver is only an approximation of the solution $x(t)$ at discrete time points. This leads to *discretization errors*.
   - Computations such as integration or Newton iteration for finding zero values are only executed with certain (fixed or variable) step sizes and terminated after a finite number of steps. This also leads to *discretization and approximation errors*.
   - Parameters and input values generally come with a certain error. For example, neither π nor the gravity g have their exact values. This in addition leads to *modeling errors*.
   - Event detection for variable-structure systems usually comprises interpolation of the discrete solution $\tilde{x}$ between mesh points t_i and t_{i+1}, and a root finding procedure. Again, this introduces an error.

We cannot do away with the intrinsic difficulties of Numerics. In particular when dealing with differential-algebraic systems there are a number of additional difficulties that have to be dealt with, e.g. the problem of order reduction of numerical methods and drift-off of the numerical solution due to high index problems or the problem of inconsistencies of initial values. A number of elaborate methods, including numerical integration, index reduction and consistent initialization have been developed over the last decades to deal with these problems, see [2, 9, 13].

The purpose of (this section of) our paper is not to indulge into these numeric issues. Rather we accept their effects as a given fact and study the impacts that this observation has for our semantic considerations.

We should point out quite clearly that this is all work in progress and that the following is but a sketch of a research direction.

7.2 Uncertainty

In order to get a grasp on these difficulties we employ the notion of *uncertainty*. That is, all values are considered to be "uncertain". This applies to all kinds of values that are computed in the simulation, also including the time points t^*. For example, if we compute the so-called zero crossing in the bouncing-ball example, we obtain the point t^* in time, at which the guard $y = 0$ becomes true and fires. However, this is only so in the ideal semantics. In the solver semantics we have an uncertain value $\tilde{y}$ which leads to an uncertain time point $\tilde{t}^*$.

As mentioned before, we are not interested in the Numerics behind this uncertainty. Uncertain values may be represented as simple intervals of real numbers or they may be represented as intervals with a, say, Gaussian distribution. One may even use ideas of fuzzy logic for such a representation. The challenge in Numerical Mathematics is to come up with algorithms and methods that allow us to infer the uncertainty of the result of a computation from the inputs of the computation.

In the following we presume the existence of such a notion of uncertainty. Then we can derive the solver semantics along the same lines as the ideal semantics, but now using the uncertain values $\tilde{x}$ and their computations in the place of the real values x, i.e., $\tilde{x} = x \pm \omega$ for a small uncertainty ω.

Then, the solver semantics are defined by the signature $\tilde{\Sigma}$ containing float or double types and floating point operations (introducing rounding errors) and the set of variables $\tilde{X}$ with corresponding uncertainties contained in the set $\tilde{W}$. Let $\tilde{E}$ be the corresponding set of equations, then for a specification $\tilde{S} = (\tilde{\Sigma}, \tilde{E})$ the set of models is denoted by $\mathit{Mod}(\tilde{S}) = \{ \tilde{A} \mid \tilde{A} \models \tilde{S} \}$ and a component is defined by $\tilde{C} = (\tilde{V}, \tilde{W}, \tilde{E})$. Composition of components and of subsystems can be defined in an analogous way as in Section 3. The same holds true for the definition of variable-structure components.

7.3 Semantics and Uncertainty

If we analyze the definitions in the previous sections, it is easily seen that the difference between the ideal and the solver semantics only plays a role at a few points. To begin with, all composition operators are not affected by the differences in the two semantics. What needs to be considered are the following issues:

- The fulfillment of equations, that is $(\tilde{A} \models \tilde{E})$ in the structure $\tilde{A}$ is influenced by uncertainty. The meaning of an equation $\tilde{x} = \tilde{y}$ as compared to $x = y$ is uncertain again. To analyze such an uncertain equation mathematically interval arithmetic can be employed. Since all our constructions are defined relative to the relation "is-a-model-of" (short: "$\models$"), there is no fundamental problem here.
- The action parts of the mode transitions are analogous to the above equations, since we essentially need to solve an uncertain equation of the form $\tilde{x}_{new} = f(\tilde{x}_{old})$.
- The most severe problem concerns the *guards*. Here we run into problems, since our definition of the transition point t^* now has to be interpreted with uncertainty, i.e., $\tilde{t}^*$ is defined by $\tilde{g}(\tilde{t}^*) = true$ and $\tilde{g}(\tau) = false$ for all τ with $\tilde{t}_1^\alpha < \tau < \tilde{t}^*$. Since both, the guard function $\tilde{g}$ and $\tilde{t}^*$ are blurred, the whole interval $\tilde{T}_1 = [\tilde{t}_1^\alpha, \tilde{t}^*)$ is blurred. This could have major effects. For example, in the case of two very close events the uncertainty could mean that the later event actually fires before the earlier one. Or two events that in reality happen simultaneously are considered as being separate due to uncertainty effects. As can be seen in the bouncing-ball example the events will (towards the end of the simulation) be so close together that – due to uncertainty – they can no longer be distinguished. (This leads in many animated simulations to the funny effect that in the end the ball breaks through the surface and travels towards the center of the earth.)

The question, if the solver semantics are faithful, i.e. converge to the ideal semantics, when the discretization step size tends to zero, strongly depends on the given model and on the employed numerical solver. Statements under which conditions a specific solver semantics converges to its ideal semantics have to be derived for individual cases.

The most challenging problems occur in variable-structure systems: besides the problem of robust treatment in the case of blurring event times one has to ensure that each instance of a mode lives long enough since the event time $\tilde{t}^*$ is determined as zero crossing in the discretization interval $[t_i, t_{i+1}]$. Concerning DAEs, in variable-structure systems not only the initial values have to be ensured to be consistent with the algebraic constraints, but also consistency of each reinitialization after events has to be guaranteed. We will not go further into the details of these problems in this paper. For the analysis and numerical treatment of hybrid DAEs we refer to [17].

These problems do exist and there is no general mechanism to avoid them. As a matter of fact, most of these effects need an application-dependent individual treatment. What we suggest is that in the semantic specification of modeling languages these solver-related effects are clearly separated from the other semantic concepts. Moreover, it should be discussed what kinds of language constructs could be added such that the modeler has the capability to describe these uncertainty effects appropriately.

8. Conclusion

We have presented a compositional semantics for essential parts of Modelica-style modeling languages. Such a compositional semantics is a mandatory prerequisite for a clean design of conceptual ideas such as variable-structure systems or compilation paradigms such as separate compilation.

Moreover, it is important to provide a clean separation of the basic modeling principles of a language from the effects that are caused by the limitations of numerics. Clearly, a deeper analysis of the latter issue is a field for extensive research in the realm of Numerical Analysis. This separation is strongly motivated by the following consideration (as was pointed out explicitly by one of the reviewers): The possibility to handle structural changes in a model and the ease of doing this depends on the computational framework that is used. By separating the semantics of the model from that of the computational framework, different frameworks can be applied to the same model, thus allowing one to realize simulators for a larger set of structural changes in the future.

References

[1] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL: the common algebraic specification language. *Theor. Comput. Sci.*, 286(2):153–196, 2002.

[2] Kathryn E. Brenan, Stephen L. Campbell, and Linda R. Petzold. *Numerical Solution of Initial-Value Problems in Differential Algebraic Equations*, volume 14 of *Classics in Applied Mathematics*. SIAM, Philadelphia, PA, 1996.

[3] David Broman. *Meta-Languages and Semantics for Equation-Based Modeling and Simulation*. PhD thesis,

Linköping University, 2010.

[4] David Broman and Peter Fritzson. Higher-Order Acausal Models. *Simulation News Europe*, 19(1):5–16, 2009.

[5] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[6] George Giorgidze and Henrik Nilsson. Embedding a Functional Hybrid Modelling language in Haskell. In *Revised selected papers of the 20th international symposium on Implementation and Application of Functional Languages, Hatfield, England*, volume 5836 of *Lecture Notes in Computer Science*. Springer, 2008.

[7] George Giorgidze and Henrik Nilsson. Mixed-level embedding and JIT compilation for an iteratively staged DSL. In *Proceedings of the 19th Workshop on Functional and (Constraint) Logic Programming (WFLP'10)*, pages 19–34, 2010.

[8] Vineet Gupta, Thomas A. Henzinger, and Radha Jagadeesan. Robust timed automata. In *Proceedings of the First InternationalWorkshop on Hybrid and Real-time Systems (HART 97)*, Lecture Notes in Computer Science 1201, pages 331–345, 1997.

[9] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer-Verlag, Berlin, second edition, 1996.

[10] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[11] Thomas A. Henzinger. The theory of hybrid automata. In *LICS*, pages 278–292, 1996.

[12] Kestrel Institute, 3260 Hillview Ave., Palo Alto, CA 94304 USA. *Specware System and documentation*, 2003. http://www.specware.org/.

[13] Peter Kunkel and Volker Mehrmann. *Differential-Algebraic Equations — Analysis and Numerical Solution*. EMS Publishing House, Zürich, Switzerland, 2006.

[14] Oded Maler, Zohar Manna, and Amir Pnueli. From timed to hybrid systems. In *REX Workshop*, pages 447–484, 1991.

[15] Zohar Manna and Amir Pnueli. Verifying hybrid systems. In *Hybrid Systems*, pages 4–35, 1992.

[16] Alexandra Mehlhase. Varying the level of detail during simulation. In *to appear in Proc. ASIM 2011*, 2011.

[17] Volker Mehrmann and Lena Wunderlich. Hybrid systems of differential-algebraic equations – analysis and numerical solution. *Journal of Process Control*, 19:1218–1228, 2009.

[18] Pieter J. Mosterman. Hybrid dynamic systems: mode transition behavior in hybrid dynamic systems. In Chick S, P. J. Sanchez, D. Ferrin, and D. J. Morrice, editors, *Proc. 2003 Winter Simulation Conference*, pages 623–631, 2003.

[19] Henrik Nilsson and George Giorgidze. Exploiting structural dynamism in functional hybrid modelling for simulation of ideal diodes. In *Proceedings of the 7th EUROSIM Congress on Modelling and Simulation, Prague, Czech Republic*. Czech Technical University Publishing House, 2010.

[20] Henrik Nilsson, John Peterson, and Paul Hudak. Functional hybrid modeling. In *Proceedings of 5th Int. Workshop on Practical Aspects of Declarative Languages*, volume 2562 of *Lecture Notes in Computer Science*, pages 376–390. Springer, 2003.

[21] Christoph Nytsch-Geusen, Andre Nordwig, Thilo Ernst, Peter Schwarz, Matthias Vetter, Christoph Wittwer, Andreas Holm, Jürgen Leopold, Gerhardt Schmidt, Ulrich Doll, and Alexander Mattes. MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics. In *Proceedings of the 4th International Modelica Conference*, 2005.

[22] Dusko Pavlovic, Peter Pepper, and Douglas R. Smith. Evolving specification engineering. In *AMAST*, pages 299–314, 2008.

[23] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP*, pages 155–179, 2008.

[24] Neil Sculthorpe and Henrik Nilsson. Keeping calm in the face of change: Towards optimisation of FRP by reasoning about change. *Journal of Higher-Order and Symbolic Computation (HOSC)*, 24(1), 2011.

[25] Dirk A. van Beek, Michel A. Reniers, Jacobus E. Rooda, and Ramon R. H. Schiffelers. Foundations of an interchange format for hybrid systems. In Alberto Bemporad, Antonio Bicchi, and Giorgio Butazzo, editors, *10th International Workshop on Hybrid Systems: Computation and Control*, volume 4416 of *Lecture Notes in Computer Science*, pages 587–600. Springer, 2007.

[26] Dirk A. van Beek, Michel A. Reniers, Jacobus E. Rooda, and Ramon R. H. Schiffelers. Revised hybrid system interchange format. Technical Report HYCON Deliverable D3.6.3, HYCON NoE, 2007.

[27] Dirk A. van Beek, Michel A. Reniers, Jacobus E. Rooda, and Ramon R. H. Schiffelers. Concrete syntax and semantics of the compositional interchange format for hybrid systems. In *Proceedings of the 17th IFAC World Congress (IFAC'08) July 11-16, 2008, Seoul, Korea*, 2008.

[28] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *PLDI 2000: Symposium on Programming Language Design and Implementation*, pages 242–252, 2000.

[29] Dirk Zimmer. *Equation-Based Modeling of Variable Structure Systems*. PhD thesis, ETH Zürich, 2010.

Simulation and Model Compilation

LIEDRIVERS — A Toolbox for the Efficient Computation of Lie Derivatives Based on the Object-Oriented Algorithmic Differentiation Package ADOL-C

Klaus Röbenack Jan Winkler Siqian Wang

Technische Universität Dresden, Faculty of Electrical and Computer Engineering, Institute of Control Theory,
{klaus.roebenack,jan.winkler,siqian.wang}@tu-dresden.de

Abstract

Lie derivatives are widely used in mathematics and physics. They are usually computed symbolically using computer algebra software. This symbolic computation might fail for very complicated expressions. Moreover, symbolic differentiation becomes more difficult if the function to be differentiated is not described explicitly as a function but by an algorithm. This is a situation occuring quite often in modeling languages. In this contribution we present an approach for calculating Lie derivatives based on algorithmic differentiation using the software package ADOL-C avoiding the drawbacks of symbolic differentiation.

Keywords Lie derivatives, algorithmic differentiation

1. Introduction

Lie derivatives play an important role in mathematics as well as physics [18, 26, 43]. Many methods in control engineering and system theory require Lie derivatives as well [20, 25]. To get an intuitive understanding, Lie derivatives can be considered as total derivatives of certain fields along the solution of a differential equation [23].

Assume we have described a physical system as lumped parameter model resulting in a set of nonlinear ordinary differential equations

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{F}(\boldsymbol{x}, \boldsymbol{u}), \quad \boldsymbol{y} = \boldsymbol{H}(\boldsymbol{x}, \boldsymbol{u}) \tag{1}$$

with the state $\boldsymbol{x}$, the input $\boldsymbol{u}$, the output $\boldsymbol{y}$ and appropriate maps $\boldsymbol{F}$ and $\boldsymbol{H}$. If system (1) is formulated in terms of a modeling language, this description is usually only employed for simulation and optimization. However, operator overloading techniques known from object-oriented programming allow the simultaneous usage of the description of (1) for control-related tasks such as controller and observer design [32]. More precisely, several algorithms in controller design rely on Lie derivatives [20, 25]. Similarly, several approaches for observer design can be formulated in terms of Lie derivatives [11, 14, 38]. Using operator overloading, these derivatives can efficiently be computed using an alternative differentiation technique called *algorithmic differentiation* [17]. We present a toolbox to carry out these calculations. Our implementation is based on the widely used algorithmic differentiation package ADOL-C (Automatic Differentiation by OverLoading in C++), cf. [16, 46].

The paper is structured as follows. In Section 2 we remind the reader of some definitions concerning Lie derivatives. Section 3 addresses the calculation of derivatives. The toolbox is presented in Section 4. The application of Lie derivatives in nonlinear control is the topic of Section 5. We use the toolbox for an example system in Section 6 and draw some conclusions in Section 7.

2. Basic Definitions from Differential Geometry

2.1 Tensors, Fields and Flows

First, we would like to recall some facts from differential geometry [3, 19, 23]. We restrict ourselves to the n-dimensional real vector space $\mathbb{V} = \mathbb{R}^n$. The elements of a vector space are *vectors*, and we represent the elements of $\mathbb{V}$ as column vectors. We denote the set of linear maps from the vector space $\mathbb{V}$ to another vector space $\mathbb{W}$ by $L(\mathbb{V}, \mathbb{W})$. Formally, the dual space of $\mathbb{V}$, denoted by $\mathbb{V}^*$, is the set of all linear functionals over $\mathbb{V}$, i.e., $\mathbb{V}^* = L(\mathbb{V}, \mathbb{R})$. The dual space $\mathbb{V}^*$ is also a vector space, whose elements are called *covectors*. Due to Riesz's representation theorem we can represent the covectors by row vectors.

A multi-linear map (i.e., a map that is linear separately in each variable)

$$(\mathbb{V}^*)^p \times \mathbb{V}^q \to \mathbb{R} \tag{2}$$

is called a *p-covariant q-contravariant tensor*, or (p, q)-*tensor*. The set $\mathbb{V}_q^p$ of (p, q)-tensors over $\mathbb{V}$ is a vector space itself. In case of $p = q = 0$ we set $\mathbb{V}_0^0 \equiv \mathbb{R}$, i.e., a $(0, 0)$-tensor is a scalar. In case of $p = 0$ and $q = 1$ the map (2) simplifies to a linear map $\mathbb{V} \to \mathbb{R}$, which belongs by definition to the dual space of $\mathbb{V}$, i.e., $\mathbb{V}_1^0 = \mathbb{V}^*$. Hence, $(0, 1)$-tensors are covectors. In case of $p = 1$ and $q = 0$

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zürich, Switzerland.

Proceedings available at:
http://www.ep.liu.se/ecp/056/
EOOLT 2011 website:
http://www.eoolt.org/2011/

the map (2) simplifies to another linear map $\mathbb{V}^* \to \mathbb{R}$, which belongs to the dual space $(\mathbb{V}^*)^*$ of the dual space $\mathbb{V}^*$. Since the vector space $\mathbb{V}$ is finite dimensional, the so-called bidual space is isomorphic to the original vector space $\mathbb{V}$, i.e., $\mathbb{V}_0^1 = (\mathbb{V}^*)^* \cong \mathbb{V}$. Therefore, the $(1,0)$-tensors are vectors.

The following consideration can be carried out for smooth manifolds. Since the paper deals mainly with computational issues, which will always be carried out in local coordinates, we can restrict ourselves to an open subset $\mathcal{M} \subseteq \mathbb{V}$ of $\mathbb{R}^n$. The *tangent space* at $\boldsymbol{x} \in \mathcal{M}$, denoted by $T_{\boldsymbol{x}}\mathcal{M}$, is isomorphic to $\mathbb{V}$, i.e., $T_{\boldsymbol{x}}\mathcal{M} \cong \mathbb{V}$. (See [23, Chap. 3] for the construction of the geometric tangent space of $\mathbb{R}^n$.) The dual space of $T_{\boldsymbol{x}}\mathcal{M}$ is called *cotangent space* and denoted by $T_{\boldsymbol{x}}^*\mathcal{M} \cong \mathbb{V}^*$. In the paper, we will deal with the following special cases: If $\mathcal{S}(\boldsymbol{x}) \in T_{\boldsymbol{x}}\mathcal{M}_0^0 \cong \mathbb{R}$, $\mathcal{S}$ is a *scalar field*. If $\mathcal{S}(\boldsymbol{x}) \in T_{\boldsymbol{x}}\mathcal{M}_0^1 = T_{\boldsymbol{x}}\mathcal{M} \cong \mathbb{V}$, $\mathcal{S}$ is a *vector field*. If $\mathcal{S}(\boldsymbol{x}) \in T_{\boldsymbol{x}}\mathcal{M}_1^0 = T_{\boldsymbol{x}}^*\mathcal{M} \cong \mathbb{V}^*$, $\mathcal{S}$ is a *covector field*. Covector fields are also called *differential forms* of degree 1 (*1-forms*). An *exact* differential form is the gradient of a scalar field.

A vector field $\boldsymbol{f}$, which maps each point $\boldsymbol{x} \in \mathcal{M}$ to the corresponding tangent space $\boldsymbol{f}(\boldsymbol{x}) \in T_{\boldsymbol{x}}\mathcal{M}$ (see Figure 1), can be associated with the differential equation

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t)) \ . \tag{3}$$

The *flow* $\boldsymbol{\varphi}_t$ of the vector field $\boldsymbol{f}$ is the general solution of (3), i.e., the curve $\boldsymbol{\varphi}_t(\boldsymbol{x}_0)$ is the solution of (3) with the initial condition $\boldsymbol{x}(0) = \boldsymbol{x}_0 \in \mathcal{M}$. Moreover, the flow is a one-parametric family of local diffeomorphisms on $\mathcal{M}$ having the structure of a transformation group with $\boldsymbol{\varphi}_0(\boldsymbol{x}_0) = \boldsymbol{x}_0$ and $\boldsymbol{\varphi}_s(\boldsymbol{\varphi}_t(\boldsymbol{x}_0)) = \boldsymbol{\varphi}_{s+t}(\boldsymbol{x}_0)$ for all sufficiently small $|t|, |s|$. The flow of (3) has the following series expansion:

$$\boldsymbol{\varphi}_t(\boldsymbol{x}) = \boldsymbol{x} + \boldsymbol{f}(\boldsymbol{x})t + \tfrac{1}{2}\boldsymbol{f}'(x)\boldsymbol{f}(\boldsymbol{x}) + \mathcal{O}(t^3) \ , \tag{4}$$

which reflects the fact that $\boldsymbol{f}(\boldsymbol{x})$ is tangent to $\boldsymbol{\varphi}_t(\boldsymbol{x})$ in the point $\boldsymbol{x}$.

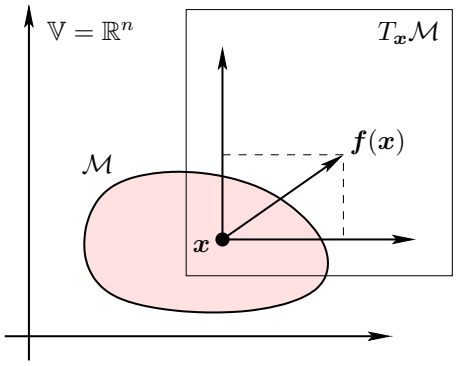


Figure 1. Tangent space and vector field $\boldsymbol{f}$.

2.2 Lie Derivatives

In this section we will define Lie derivatives of scalar, vector and covector fields.

2.2.1 Lie Derivative of a Scalar Field

Consider the differential equation (3) with the vector field $\boldsymbol{f} : \mathcal{M} \to \mathbb{R}^n$ and the associated flow $\boldsymbol{\varphi}_t$. The *Lie derivative of the scalar field* $h : \mathcal{M} \to \mathbb{R}$ is defined as

$$L_{\boldsymbol{f}}h(\boldsymbol{x}) = \frac{\mathrm{d}}{\mathrm{d}\,t}h(\boldsymbol{\varphi}_t(\boldsymbol{x}))\bigg|_{t=0} \ . \tag{5}$$

Using the series expansion (4) of the flow, with

$$\begin{aligned} L_{\boldsymbol{f}}h(\boldsymbol{x}) &= \frac{\mathrm{d}}{\mathrm{d}\,t}h(\boldsymbol{\varphi}_t(\boldsymbol{x}))\bigg|_{t=0} \\ &= h'(\boldsymbol{\varphi}_t(\boldsymbol{x}))\frac{\mathrm{d}}{\mathrm{d}\,t}\boldsymbol{\varphi}_t(\boldsymbol{x})\bigg|_{t=0} \\ &= h'(\boldsymbol{x})\boldsymbol{f}(\boldsymbol{x}) \end{aligned} \tag{6}$$

we obtain the explicit computation rule (6) as the scalar product between the gradient $\mathrm{d}\,h = h'$ and the vector field $\boldsymbol{f}$. In other words: $L_{\boldsymbol{f}}h$ is the directional derivative of h in the direction of $\boldsymbol{f}$ (see Figure 2). The Lie derivative $L_{\boldsymbol{f}}h$ is again a scalar field. Therefore, we can repeat this process and define multiple Lie derivatives along the same vector field $\boldsymbol{f}$ by

$$L_{\boldsymbol{f}}^{k+1}h(\boldsymbol{x}) = L_{\boldsymbol{f}}L_{\boldsymbol{f}}^{k}h(\boldsymbol{x}) = \frac{\partial L_{\boldsymbol{f}}^{k}h(\boldsymbol{x})}{\partial \boldsymbol{x}}\,\boldsymbol{f}(\boldsymbol{x}) \tag{7}$$

with $L_{\boldsymbol{f}}^{0}h(\boldsymbol{x}) = h(\boldsymbol{x})$.

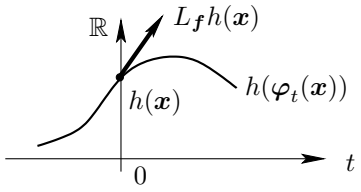


Figure 2. Lie derivative $L_{\boldsymbol{f}}h$ of a scalar field h

2.2.2 Lie Derivative of a Vector Field

We need some preliminary remarks before we introduce the Lie derivative of a vector field. Consider a smooth map $\boldsymbol{\psi} : \mathcal{M} \to \mathcal{M}$, which maps a point $\boldsymbol{x} \in \mathcal{M}$ into the point $\boldsymbol{\psi}(\boldsymbol{x}) \in \mathcal{M}$. The *differential* of *push-forward* is a linear map between the associated tangent spaces $\boldsymbol{\psi}_* : T_{\boldsymbol{x}}\mathcal{M} \to T_{\boldsymbol{\psi}(\boldsymbol{x})}\mathcal{M}$, which can be represented by the Jacobian matrix $\boldsymbol{\psi}'$ of $\boldsymbol{\psi}$. If $\boldsymbol{\psi}$ is a diffeomorphism, the inverse map $\boldsymbol{\psi}^{-1}$ of $\boldsymbol{\psi}$ exists and is continuously differentiable. Then, the linearization of the inverse map $\boldsymbol{\psi}^{-1}$ defines another map $\boldsymbol{\psi}^* := \boldsymbol{\psi}_*^{-1} : T_{\boldsymbol{\psi}(\boldsymbol{x})}\mathcal{M} \to T_{\boldsymbol{x}}\mathcal{M}$ called *pull-back*. Clearly, the Jacobian of the inverse map is the inverse Jacobian matrix of the original map. In case of $\boldsymbol{\psi} = \boldsymbol{\varphi}_t$, we obtain from (4) a series expansion of push-forward

$$\boldsymbol{\varphi}_{t*}(\boldsymbol{x}) = I + \boldsymbol{f}'(\boldsymbol{x})t + \mathcal{O}(t^2) \ , \tag{8}$$

where I denotes the identity matrix. Due to the group property, the inverse map of the flow is given as the flow in reverse time, i.e., $\boldsymbol{\varphi}_t^{-1} = \boldsymbol{\varphi}_{-t}$. This implies $\boldsymbol{\varphi}_t^* = \boldsymbol{\varphi}_{t*}^{-1} = \boldsymbol{\varphi}_{-t*}$ having the series expansion

$$\boldsymbol{\varphi}_t^*(\boldsymbol{z}) = \boldsymbol{\varphi}_{-t*}(\boldsymbol{z}) = I - \boldsymbol{f}'(\boldsymbol{z})t + \mathcal{O}(t^2) \tag{9}$$

with $\boldsymbol{z} = \boldsymbol{\varphi}_t(\boldsymbol{x})$.

We are now able to introduce the *Lie derivative of a vector field* $\boldsymbol{g}$ along $\boldsymbol{f}$ as

$$L_{\boldsymbol{f}}\boldsymbol{g}(\boldsymbol{x}) = \mathrm{ad}_{\boldsymbol{f}}\,\boldsymbol{g}(\boldsymbol{x}) = \left.\frac{\mathrm{d}}{\mathrm{d}\,t}\boldsymbol{\varphi}_t^*\boldsymbol{g}(\boldsymbol{\varphi}_t(\boldsymbol{x}))\right|_{t=0} \quad . \tag{10}$$

This Lie derivative is called *Lie bracket* and often denoted by $\mathrm{ad}_{\boldsymbol{f}}\,\boldsymbol{g} = [\boldsymbol{f}, \boldsymbol{g}]$. Using the series expansions (4) and (9) we obtain

$$\begin{aligned}
\mathrm{ad}_{\boldsymbol{f}}\,\boldsymbol{g}(\boldsymbol{x}) &= \left.\frac{\mathrm{d}}{\mathrm{d}\,t}\boldsymbol{\varphi}_t^*\boldsymbol{g}(\boldsymbol{\varphi}_t(\boldsymbol{x}))\right|_{t=0} \\
&= \left.\frac{\mathrm{d}}{\mathrm{d}\,t}\boldsymbol{\varphi}_{-t*}\boldsymbol{g}(\boldsymbol{\varphi}_t(\boldsymbol{x}))\right|_{t=0} \\
&= \lim_{t\to 0}\frac{\boldsymbol{\varphi}_{-t*}\boldsymbol{g}(\boldsymbol{\varphi}_t(\boldsymbol{x})) - \boldsymbol{g}(\boldsymbol{x})}{t} \qquad (11)\\
&= \boldsymbol{g}'(\boldsymbol{x})\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{f}'(\boldsymbol{x})\boldsymbol{g}(\boldsymbol{x}) \quad . \qquad (12)
\end{aligned}$$

The reason for the seemingly complicated construction (10) becomes apparent in Eq. (11). In general, the derivative of a function is defined as the limit of the difference quotient. Formally, the two terms $\boldsymbol{g}(\boldsymbol{x}) \in T_{\boldsymbol{x}}\mathcal{M}$ and $\boldsymbol{g}(\boldsymbol{\varphi}_t(\boldsymbol{x})) \in T_{\boldsymbol{\varphi}_t(\boldsymbol{x})}\mathcal{M}$ required for the difference quotient belong to different vector spaces. Using pull-back we map $\boldsymbol{g}(\boldsymbol{\varphi}_t(\boldsymbol{x}))$ back from $T_{\boldsymbol{\varphi}_t(\boldsymbol{x})}\mathcal{M}$ to $T_{\boldsymbol{x}}\mathcal{M}$ and obtain $\boldsymbol{\varphi}_t^*\boldsymbol{g}(\boldsymbol{\varphi}_t(\boldsymbol{x})) \in T_{\boldsymbol{x}}\mathcal{M}$. Then, we can compute the difference in (11) since both objects lie in the same tangent space.

The Lie derivative $\mathrm{ad}_{\boldsymbol{f}}\,\boldsymbol{g}$ of a vector field is a vector field again. Higher order Lie derivatives of a vector field $\boldsymbol{g}$ are recursively defined by

$$\mathrm{ad}_{\boldsymbol{f}}^{k+1}\,\boldsymbol{g}(\boldsymbol{x}) = [\boldsymbol{f}, \mathrm{ad}_{\boldsymbol{f}}^{k}\,\boldsymbol{g}](\boldsymbol{x}) \tag{13}$$

with $\mathrm{ad}_{\boldsymbol{f}}^{0}\,\boldsymbol{g} = \boldsymbol{g}(\boldsymbol{x})$.

2.2.3 Lie Derivative of a Covector Field

Consider a smooth map $\boldsymbol{\psi} : \mathcal{M} \to \mathcal{M}$ with its push-forward $\boldsymbol{\psi}_* : T_{\boldsymbol{x}}\mathcal{M} \to T_{\boldsymbol{\psi}(\boldsymbol{x})}\mathcal{M}$ and a covector field $\boldsymbol{\omega}$, i.e., $\boldsymbol{\omega}(\boldsymbol{x}) \in T^*_{\boldsymbol{\psi}(\boldsymbol{x})}\mathcal{M}$ for $\boldsymbol{x} \in \mathcal{M}$. The *dual* or *adjoint* map $\boldsymbol{\psi}^* : T_{\boldsymbol{\psi}(\boldsymbol{x})}\mathcal{M} \to T_{\boldsymbol{x}}\mathcal{M}$ defined by $\langle \boldsymbol{\psi}^*\boldsymbol{\omega}, \boldsymbol{z}\rangle = \langle \boldsymbol{\omega}, \boldsymbol{\psi}_*\boldsymbol{z}\rangle$ for all $\boldsymbol{z} \in T_{\boldsymbol{x}}\mathcal{M}$ and $\boldsymbol{\omega}(\boldsymbol{x}) \in T^*_{\boldsymbol{\psi}(\boldsymbol{x})}\mathcal{M}$ is called *pull-back* of the covector field $\boldsymbol{\omega}$. For $\boldsymbol{\psi} = \boldsymbol{\varphi}_t$ we obtain from (4) the following series expansion of pull-back of the covector field $\boldsymbol{\omega}$ for the flow:

$$\begin{aligned}
\boldsymbol{\varphi}_t^*\boldsymbol{\omega}(\boldsymbol{x}) &= \boldsymbol{\omega}(\boldsymbol{x})\boldsymbol{\varphi}_{t*}(\boldsymbol{x}) \\
&= \boldsymbol{\omega}(\boldsymbol{x})(I + \boldsymbol{f}'(\boldsymbol{x})t + \mathcal{O}(t^2)) \quad .
\end{aligned} \tag{14}$$

The *Lie derivative of the covector field* $\boldsymbol{\omega}$ is defined by

$$L_{\boldsymbol{f}}\boldsymbol{\omega}(\boldsymbol{x}) = \left.\frac{\mathrm{d}}{\mathrm{d}\,t}\boldsymbol{\varphi}_t^*\boldsymbol{\omega}(\boldsymbol{\varphi}_t(\boldsymbol{x}))\right|_{t=0} \quad . \tag{15}$$

The definitions (10) and (15) of the Lie derivatives of a vector and a covector field look exactly the same. However, pull-back acts on different mathematical objects (on a vector field in (10) and on a covector field in (15)) and therefore on different spaces.

Using the series expansions (4) and (14) we obtain by

$$\begin{aligned}
L_{\boldsymbol{f}}\boldsymbol{\omega}(\boldsymbol{x}) &= \left.\frac{\mathrm{d}}{\mathrm{d}\,t}\boldsymbol{\varphi}_t^*\boldsymbol{\omega}(\boldsymbol{\varphi}_t(\boldsymbol{x}))\right|_{t=0} \\
&= \left.\frac{\mathrm{d}}{\mathrm{d}\,t}\boldsymbol{\omega}(\boldsymbol{\varphi}_t(\boldsymbol{x}))\boldsymbol{\varphi}_{t*}(\boldsymbol{\varphi}_t(\boldsymbol{x}))\right|_{t=0} \\
&= \boldsymbol{\omega}(\boldsymbol{x})\boldsymbol{f}'(\boldsymbol{x}) + \boldsymbol{f}^T(\boldsymbol{x})\left(\frac{\partial \boldsymbol{\omega}^T(\boldsymbol{x})}{\partial \boldsymbol{x}}\right)^T \qquad (16)
\end{aligned}$$

an explicit computation rule for the Lie derivative (15).

As mentioned in Section 2.1, the gradient $\mathrm{d}\,h$ of a scalar field h is a covector field. Therefore, the Lie derivative $L_{\boldsymbol{f}}\,\mathrm{d}\,h$ can be calculated by (16) with $\boldsymbol{\omega} = \mathrm{d}\,h$. Alternatively, the Lie derivative $L_{\boldsymbol{f}}\,\mathrm{d}\,h$ can also be obtained from the gradient of the Lie derivative $L_{\boldsymbol{f}}h$:

$$L_{\boldsymbol{f}}\,\mathrm{d}\,h(\boldsymbol{x}) = \mathrm{d}\,L_{\boldsymbol{f}}h(x) \quad . \tag{17}$$

Eq. (17) can be stated as follows: The exterior derivative commutes with the Lie derivative.

3. Computation of Derivatives

3.1 General Differentiation Techniques

The *(Fréchet) derivative* of a smooth map $\boldsymbol{F} : \mathcal{M} \to \mathbb{R}^m$ in the point $\boldsymbol{x}_0 \in \mathcal{M} \subseteq \mathbb{R}^n$ is a linear map $A \in L(\mathbb{R}^n, \mathbb{R}^m)$, for which

$$\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{F}(\boldsymbol{x}_0) + A(\boldsymbol{x} - \boldsymbol{x}_0) + o(\|\boldsymbol{x} - \boldsymbol{x}_0\|)$$

holds for all $\boldsymbol{x}$ in a neighbourhoud of $\boldsymbol{x}_0$. The linear map A can be identified with an $m \times n$-matrix $A = \boldsymbol{F}'(\boldsymbol{x}) \in \mathbb{R}^{m\times n}$, the so-called *Jacobian matrix*.

In this paper we assume that the map $\boldsymbol{F} : \mathcal{M} \to \mathbb{R}^m$ is given as a computer program (such as a function, procedure, method etc.), i.e., $\boldsymbol{F}$ is described as a finite sequence of elementary functions and operations. The derivative of $\boldsymbol{F}$ can be computed systematically using elementary differentiation rules in combination with the chain rule. This differentiation process can be carried out by computer algebra systems such as MATHEMATICA [48], MAPLE [10] or MAXIMA [2]. The result of this *symbolic computation* is a symbolic expression.

The computer algebra system might not be able to carry out a symbolic differentiation if the function under consideration is very complicated. Moreover, symbolic computation is usually not (directly) possible if the function to be differentiated is not given explicitly but by algorithms containing branches, loops and subroutines. Moreover, the sizes of symbolic expressions usually increase significantly w.r.t. the order of the derivative.

Another widely used method to compute derivative values is *numeric differentiation* by divided differences. Depending on the step size, the resulting derivative will be affected by truncation or cancellation errors. Even for an optimal step size, the derivative value will have a significantly reduced precision compared to the function value.

The disadvantages of symbolic and numeric differentiation can be circumvented by an alternative technique

called *algorithmic* or *automatic differentiation* [17]. Similar to symbolic differentiation, elementary differentiation rules are applied systematically. In contrast to symbolic differentiation, all intermediate results are not symbolic expressions but numerical values, i.e., the intermediate values are immediately evaluated and stored as floating point numbers.

3.2 Symbolic Computation of Lie Derivatives

Symbolically, Lie derivatives of scalar, vector and covector fields are computed based on Eqs. (6), (12) and (16). Hence, one needs to compute gradients or Jacobian matrices. These operations are well-supported by the usual computer algebra systems. For example in MAPLE, the Lie derivative (6) of the scalar field h along the vector field $\boldsymbol{f}$ can be implemented as follows:

```
with(linalg);
LieScalar:=proc(f,h,x)
  multiply(jacobian(h,x),f)
end_proc;
```

Higher order Lie derivatives can be calculated easily using finite recursions such as (7) or (13). The symbolic computation of Lie derivatives was implemented in [7, 22, 27, 36] for MATHEMATICA and in [12, 21, 24, 37] for MAPLE. However, the symbolic calculation of Lie derivatives is not restricted to commercial systems. The open source computer algebra system MAXIMA contains toolboxes for tensor calculus, which were developed for computations in general relativity and support the calculation of arbitrary types of Lie derivatives [42]. Alternatively, the Lie derivatives (6), (12) and (16) can also be implemented in MAXIMA using the build-in linear algebra package. A possible implementation of the Lie derivative (6) reads as follows:

```
load("linearalgebra");
LieScalar(f,h,x):=jacobian([h],x).f;
```

3.3 Algorithmic Differentiation

3.3.1 Forward Mode

Assume we represent a vector-valued function $\boldsymbol{F} : \mathcal{M} \to \mathbb{R}^m$, $\mathcal{M} \subseteq \mathbb{R}^n$ with

$$\boldsymbol{z} = \boldsymbol{F}(\boldsymbol{x}) \tag{18}$$

by appropriate C++ code. Input values $\boldsymbol{x}_0 \in \mathcal{M}$, output values (i.e., function values) $\boldsymbol{z}_0 \in \mathbb{R}^m$ as well as intermediate values are stored as floating point numbers, usually with the build-in type `double`. Considering $\boldsymbol{x}$ and $\boldsymbol{z}$ as curves, we can compute the time derivative of (18) using the chain rule:

$$\dot{\boldsymbol{z}} = \boldsymbol{F}'(\boldsymbol{x})\,\dot{\boldsymbol{x}} \ . \tag{19}$$

For a given direction $\dot{\boldsymbol{x}}$ of $\mathbb{R}^n$ we obtain the directional derivative $\dot{\boldsymbol{z}}$ of $\boldsymbol{F}$. The tangent value $\dot{\boldsymbol{z}}$ can be obtained calculating the tangent values of all intermediate values occurring during the function evaluation. We repleace the floating point type `double` by a new class, e.g. `ddouble`, containing the function value and additionally the derivative value:

```
class ddouble
{
public:
  double val; // function value
  double der; // derivative value
};
```

All differentiable functions (e.g. $\sin$, $\cos$, $\exp$, $\log$) that act on the type `double` must be replaced by appropriate methods for the class `ddouble` such that in addition to the function value one also computes the derivative value using elementary differentiation rules in connection with the chain rule:

```
ddouble sin (ddouble x)
{
 ddouble z;
 z.val = sin(x.val);
 z.der = x.der*cos(x.val);
}
```

In a similar way we have to provide the usual binary operation (e.g. $+$, $-$, $*$, $/$) for the new class `ddouble`. In case of multiplication we also have to take the product rule into account:

```
ddouble operator * (ddouble x, ddouble y)
{
 ddouble z;
 z.val = x.val*y.val;
 z.der = x.val*y.der+y.val*x.der;
}
```

This approach to algorithmic differentiation is called *forward mode* because the program flow of function evaluation and derivative computation have the same orientation.

3.3.2 Reverse Mode

In the reverse mode, the elementary statements are differentiated in reverse order, i.e., from the end to the beginning of the program. Therefore, the implementation is much more complicated. The reverse mode can be interpreted as a generalization of the backpropagation algorithm known from neuronal networks [47].

Mathematically, for a given covector (i.e., row vector) $\bar{\boldsymbol{z}} \in (\mathbb{R}^m)^*$, the reverse mode yields a covector $\bar{\boldsymbol{x}} \in (\mathbb{R}^n)^*$ with

$$\bar{\boldsymbol{x}} = \bar{\boldsymbol{z}}\,\boldsymbol{F}'(\boldsymbol{x}) \ , \tag{20}$$

which can be seen as a weighted derivative. For $m < n$, especially in case of a functional, i.e., $m = 1$, the reverse mode is more efficient than the forward mode. Therefore, the reverse mode is widely used in optimization [15].

3.3.3 Taylor Arithmetic

Assume the function $\boldsymbol{F} : \mathcal{M} \to \mathbb{R}^m$ is sufficiently smooth to map a truncated Taylor series

$$\boldsymbol{x}(t) = \boldsymbol{x}_0 + \boldsymbol{x}_1 t + \boldsymbol{x}_2 t^2 + \cdots + \boldsymbol{x}_d t^d + \mathcal{O}(t^{d+1}) \tag{21}$$

with $\boldsymbol{x}_0 \in \mathcal{M} \subseteq \mathbb{R}^n$, $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_d \in \mathbb{R}^n$ into a curve

$$\begin{aligned} \boldsymbol{z}(t) &= \boldsymbol{F}(\boldsymbol{x}(t)) \\ &= \boldsymbol{z}_0 + \boldsymbol{z}_1 t + \boldsymbol{z}_2 t^2 + \cdots + \boldsymbol{z}_d t^d + \mathcal{O}(t^{d+1}) \end{aligned} \tag{22}$$

with $\boldsymbol{z}_0, \ldots, \boldsymbol{z}_d \in \mathbb{R}^d$. Clearly, each Taylor coefficient $\boldsymbol{z}_k$ depends only on the Taylor coefficients $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_k$. The Taylor coefficient $\boldsymbol{z}_k$ can be calculated using the forward mode of automatic differentiation.

In reverse mode of automatic differentiation, one can compute the coefficient matrices $A_0, \ldots, A_d \in \mathbb{R}^{m \times n}$ of the Jacobian path

$$\mathbf{F}'(\mathbf{x}(t)) = A_0 + A_1 t + \ldots + A_d t^d + \mathcal{O}(t^{d+1}) \ . \quad (23)$$

The matrices $A_0, \ldots, A_d$ are the partial derivatives of the Taylor coefficients of the curves $\mathbf{x}$ and $\mathbf{z}$. We have the following identity [8]:

$$\frac{\partial \mathbf{z}_j}{\partial \mathbf{x}_i} = \frac{\partial \mathbf{z}_{j-i}}{\partial \mathbf{x}_0} = \begin{cases} A_{j-i} & \text{for} \quad j \geq i \\ 0 & \text{otherwise.} \end{cases} \quad (24)$$

As illustrated, the curves $\boldsymbol{x}$ and $\boldsymbol{z}$ associated with a function $\boldsymbol{F}$ are represented by the coefficients of their corresponding truncated Taylor series expansions (21) and (22). Thus, it is necessary to introduce another datatype holding the values of these coefficients. As an example, one might define

```
#include <vector>
class tdouble: public std::vector<double>;
```

and implement the operations required for an appropriate handling of these truncated series expansions.

3.4 Implementations and Tools

As sketched in Section 3.3, algorithmic differentiation can easily be implmented if the programming language under consideration supports operator overloading. For C++, this approach is implemented in several algorithmic differentiation packages such as ADOL-C [16], FADBAD [5], FADBAD++ [41], TADIFF [6], AUTODIFF [39], CppAD [1]. Operator overloading is also supported in Fortran 90/95 and used by the tools AUTO_DERIV [40] and TaylUR [45]. Moreover, there are also tools that use operator overloading in Matlab for algorithmic differentiation, e.g. ADMAT [44] and TOMLAB/MAD [13].

It should be mentioned that algorithmic differentiation can also be implemented based on source code transformation, especially for programming languages such as C and Fortran 77 which do not support operator overloading. For more details on algorithmic differentiation tools and techniques we refer to [4, 17].

3.5 Differentiation Package ADOL-C

The software package ADOL-C provides routines for evaluation of derivatives of scalar or vector functions defined by computer programs written in the programming languages C or C++. The resulting derivative evaluation routines can be used in C, C++, Fortran, or any other language that can be linked against C [16]. ADOL-C uses operator overloading as described in Section 3.3, however with some additional features that will be discussed in the next section.

3.5.1 Tape generation

The fundamental working principle of ADOL-C is that in a first step the code representing e.g. a scalar valued function $f : \mathbb{R} \to \mathbb{R}$ is marked as a so called active section. Input and output variables are assigned as variables of type `adouble`, similar to the type `ddouble` discussed above:

```
trace_on(tag); // Begin of active section

adouble x,z;    // Active variables
double x0;      // point of expansion
double z0;      // function value

x <<= x0;       // Assignment of independents
z = f(x);       // Evaluation
z >>= z0;       // Assignment of dependents

trace_off();    // End of active section
```

This code creates a data structure called *tape* holding the trace of the function evaluation. This tape is used in the following for the calculation of the derivatives using so called *drivers*, e.g. `gradient`, `jacobian`, `hessian`, cf. next section. It is important to understand that the tape has only to be created once a time for an aribitrary input value x_0 while repetitive calls of the drivers for different input values differing from the values the tape was generated from may follow. This holds as long as there are no user defined quadratures and all comparisons involving `adouble`s yield the same result. The drivers acting on these tapes provide a C as well as C++ interface, i.e., once a tape is generated it can also be used in environments that do not support C++. The tape is referenced by the integer `tag`.

3.5.2 Drivers

The package ADOL-C provides several commands called *drivers* for evaluation of different types of derivatives. Some of these drivers are sketched in the following:

Drivers for forward and reverse mode Given a tape of the sufficiently smooth function $\boldsymbol{F} : \mathcal{M} \to \mathbb{R}^m$ and the Taylor coefficients $X = (\boldsymbol{x}_0, \ldots, \boldsymbol{x}_d)$ of the series expansion (21) of the curve $\boldsymbol{x}$ of the independent variable one can compute the Taylor coefficients $Z = (\boldsymbol{z}_0, \ldots, \boldsymbol{z}_d)$ as given in (22) using the ADOL-C function `forward`:

```
int forward(tag,m,n,d,keep,X,Z)
short int tag;      // tape tag of F
int m;              // number of dependent variables m
int n;              // number of independent variables n
int d;              // highest derivative degree d
int keep;           // flag for reverse sweep
double X[n][d+1];   // Taylor coefficients (x_0, ..., x_d)
double Z[m][d+1];   // Taylor coeffcients (z_0, ..., z_d)
```

The keep flag must be set to `keep=1` if a further calculation using the reverse mode is intended. The 2-dimensional arrays X and Z are allocated as arrays of pointers.

The reverse mode can be employed for an efficient computation of the coefficient matrices $A_0, \ldots, A_d \in \mathbb{R}^{m \times n}$ of the Jacobian path (23). They are obtained by calling

the ADOL-C function `reverse` and stored in the 3-dimensional array `A`:

```
int reverse(tag,m,n,d,A)
short int tag;        // tape tag of F
int m;                // number of dependent variables m
int n;                // number of independent variables n
int d;                // highest derivative degree d
double A[m][n][d+1];  // resulting matrices A_0, ..., A_d
```

Drivers for ordinary differential equations Given an initial value problem

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t)) \quad \text{with} \quad \boldsymbol{x}(0) = \boldsymbol{x}_0 \in \mathcal{M} \subseteq \mathbb{R}^n \tag{25}$$

of the ordinary differntial equation (3). We want to compute a Taylor series expansion (21) of a solution of (25), i.e., of the flow of the vector field $\boldsymbol{f}$ passing through $\boldsymbol{x}_0$:

$$\boldsymbol{x}(t) = \boldsymbol{\varphi}_t(\boldsymbol{x}_0) \ . \tag{26}$$

On the other hand, we can treat $\boldsymbol{f}$ as a functions mapping the curve (21) into the curve (22) via

$$\boldsymbol{z}(t) = \boldsymbol{f}(\boldsymbol{x}(t)) \ . \tag{27}$$

Because of $\boldsymbol{f} : \mathcal{M} \subseteq \mathbb{R}^n \to \mathbb{R}^n$, both curves $\boldsymbol{x}$ and $\boldsymbol{z}$ belong to spaces of the same dimension. However, since $\boldsymbol{f}$ is a vector field, the curve $\boldsymbol{z}$ in the image of $\boldsymbol{f}$ must belong to the tangent space $T_{\boldsymbol{x}}\mathcal{M}$, i.e., we have the identify $\boldsymbol{z}(t) = \dot{\boldsymbol{x}}(t)$. Therefore, the Taylor coefficients of the series expansions (21) and (22) must satiesfy the identity

$$\boldsymbol{x}_{k+1} = \frac{1}{1+k}\boldsymbol{z}_k \ . \tag{28}$$

Knowing certain Taylor coefficients $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_k$ of the solution of (25), one can use the forward mode to compute the Taylor coefficients $\boldsymbol{z}_0, \ldots, \boldsymbol{z}_k$ of the curve (27). Eq. (28) yields the next Taylor coefficient $\boldsymbol{x}_{k+1}$. To carry out this recursion, ADOL-C provides the function `forode`:

```
int forode(Tape_F, n, d, X)
short Tape_F;         // tape tag of vector field f
short n;              // dimension n
short d;              // highest degree d
double X[n][d+1];     // Taylor coefficients x_0, ..., x_d
```

The initial value $\boldsymbol{x}_0$ has to be stored in the zeroth position of the array `X`, i.e., in `X[0][0]`, ..., `X[n-1][0]`.

Carrying out a reverse sweep after `forode` we obtain the coefficient matrices $A_0, A_1, \ldots, A_{d-1} \in \mathbb{R}^{n\times n}$ similar to (23). Recall that these matrices can be interpreted as partial derivatives between Taylor coefficients as described in (24). Taking the dependencies resulting from (28) into account we can calculate the total derivatives

$$B_k = \frac{\mathrm{d}\boldsymbol{x}_{k+1}}{\mathrm{d}\boldsymbol{x}_0} \in \mathbb{R}^{n\times n} \tag{29}$$

for $k = 0, \ldots, d-1$ with the ADOL-C function `accode`:

```
int accode(n, d-1, A, B)
short n;              // dimension n
short d;              // highest degree d
double A[n][n][d];    // partial derivatives A_0, ..., A_{d-1}
double B[n][n][d];    // total derivatives B_0, ..., B_{d-1}
```

Note that the total derivatives (29) are the coefficient matrices of the series expansion of push-forward (8) of the flow:

$$\boldsymbol{\varphi}_{t*}(\boldsymbol{x}_0) = I + B_0 t + B_1 t^2 + B_2 t^3 + \cdots \ .$$

ADOL-C provides much more drivers, e.g. for optimization and nonlinear equations (calculation of gradients, Jacobians and Hessians) and higher derivative tensors which are not discussed here. For details refer to [16, 46].

3.5.3 Calculation of Lie derivatives using ADOL-C

In contrast to the symbolic computation the calculation of the Lie derivatives employing automatic differentiation is based directly on the definitions (6), (10) and (15). In order to use ADOL-C for the calculation of Lie-derivatives one has to combine certain calls of the drivers discussed above. This is illustrated by example of the Lie derivative $L^k_{\mathbf{f}}h(\mathbf{x})$ of a scalar field $h : \mathcal{M} \to \mathbb{R}$ along a vector field $\boldsymbol{f} : \mathcal{M} \to \mathbb{R}^n$. Using the ADOL-C driver `forode`, in a first step, one computes the Taylor coefficients $\boldsymbol{x}_1, \ldots \boldsymbol{x}_d \in \mathbb{R}^n$ of the series expansion (21) of (26) for a given initial value $\boldsymbol{x}_0 \in \mathcal{M} \subseteq \mathbb{R}^n$. In a second step we have to compute the Taylor coefficients $y_0, y_1, \ldots, y_d \in \mathbb{R}$ of the series expansion

$$y(t) = y_0 + y_1 t + \cdots y_d t^d + \mathcal{O}(t^{d+1}) \tag{30}$$

of the curve

$$y(t) = h(\boldsymbol{\varphi}_t(\boldsymbol{x}_0)) \ . \tag{31}$$

This can easily be done with the driver `forward`. Recall that the first order Lie derivative (5) is the total derivative of h along the flow of $\boldsymbol{f}$. Taking higher order Lie derivatives into account yields a so-called Lie series

$$\sum_{k=0}^{\infty} L^k_{\boldsymbol{f}} h(\boldsymbol{x}_0)\frac{t^k}{k!} = h(\boldsymbol{\varphi}_t(\boldsymbol{x}_0)) \ . \tag{32}$$

Matching the coefficients of (30) and (31) allows an explicit computation of the function values of the Lie derivatives by

$$L^k_{\boldsymbol{f}} h(\boldsymbol{x}_0) = k!\, y_k \quad \text{for} \quad k = 0, \ldots, d \ . \tag{33}$$

Assuming that the recorded tapes of $\boldsymbol{f}$ and h are referenced by the integers `Tape_f` and `Tape_h`, respectively, one has:

```
// Taylor coefficients X = (x_k)
forode(Tape_f,n,d,X);

// Taylor coefficients Y = (y_k)
forward(Tape_h,m,n,d,X,Y);

// Lie-Derivatives up to order d
double Lfh[d+1];
int fak = 1;
for (i=0; i <= d; i++) {
   Lfh[i] = y[i]*fak;
   fak *= (i+1);
}
```

In a similar way one can compute the gradients of Lie derivatives, the Lie derivatives of a covector field and Lie brackets. For example, consecutive calls of `reverse` and `accode` provide the coefficient matrices of the series expansion (8) required for the computation of Lie derivatives of covector fields, see Eqs. (14) and (16).

4. Toolbox of Lie-Derivatives

The drivers provided for calculating Lie derivatives of scalar, vector, covector fields and gradient of Lie derivatives of scalar or vector fields are prototyped in the header <lie_tool.h>. Drivers utilizing C or C++ are provided all of them working on the tapes generated previously as described above, i.e., no repetitive direct evaluation of the function source code is required. In the following we give an overview about the routines provided by the Liepackage.

4.1 Lie Derivatives of a Scalar Field

In order to compute the Lie derivatives of the scalar field h along the vector field $\boldsymbol{f}$ at the point $\boldsymbol{x} = \boldsymbol{x}_0 \in \mathbb{R}^n$, we need the tape numbers of active sections of $\boldsymbol{f}$ and h, the number of independent variables n and the highest derivative degree d. The values of the Lie derivatives $L^0_{\boldsymbol{f}}h(\mathbf{x}_0), \ldots, L^d_{\boldsymbol{f}}h(\mathbf{x}_0)$ will be stored in a one-dimensional array (vector) of size $d+1$. The C function `Lie_scalarc` has the following arguments:

```
int Lie_scalarc(Tape_F,Tape_H,n,x0,d,res)
short Tape_F;      // tape tag of vector field f
short Tape_H;      // tape tag of scalar field h
short n;           // dimension n
double x0[n];      // vector x_0
short d;           // highest degree d
double res[d+1];   // Lie derivatives
```

Now, consider a vector-valued maps $\boldsymbol{h} : \mathbb{R}^n \to \mathbb{R}^m$. The component maps $h_1, \ldots, h_m : \mathbb{R}^n \to \mathbb{R}$ of which $\boldsymbol{h}$ consists of are scalar fields. We understand the Lie derivative of $\boldsymbol{h}$ along $\boldsymbol{f}$ as follows

$$L_{\boldsymbol{f}}\boldsymbol{h}(\boldsymbol{x}) = \begin{pmatrix} L_{\boldsymbol{f}}h_1(\boldsymbol{x}) \\ \vdots \\ L_{\boldsymbol{f}}h_p(\boldsymbol{x}) \end{pmatrix}, \tag{34}$$

which is essentially a simplified notation for the Lie derivatives $L_{\boldsymbol{f}}h_1, \ldots, L_{\boldsymbol{f}}h_m$ of the component maps. This notation is occasionally used in nonlinear control and the associated software implementations [22, 27]. Higher order Lie derivatives $L^k_{\boldsymbol{f}}\boldsymbol{h}$ are definied by the same recursion as in (7). Note that the Lie derivative (34) of the vector-valued map $\boldsymbol{h}$ should not be confused with the Lie derivative of a vector field.

The Lie derivatives of $\boldsymbol{h}$ along $\boldsymbol{f}$ at the point $\boldsymbol{x}_0$ can be computed with the C function `Lie_scalarcv`. The resulting values of the Lie derivatives $L^0_{\boldsymbol{f}}\boldsymbol{h}(\boldsymbol{x}_0), \ldots, L^d_{\boldsymbol{f}}\boldsymbol{h}(\boldsymbol{x}_0)$ will be stored in a two-dimensional array (i.e., a matrix) of size $m \times (d+1)$:

```
int Lie_scalarcv(Tape_F,Tape_H,n,m,x0,d,res)
short Tape_F;      // tape tag of vector field f
short Tape_H;      // tape tag of vector map h
short n;           // dimension n
short m;           // dimension m
double x0[n];      // vector x_0
short d;           // highest degree d
double res[m][d+1]; // lie derivatives
```

In addition, we implemented the C++ wrapper function `Lie_scalar` which supports both calling conventions, i.e., for $m = 1$ and arbitrary $m \geq 1$. Details on the computational algorithms can be found in [29, 34].

4.2 Gradients of Lie Derivatives of a Scalar Field

The gradients

$$\mathrm{d}\,h(\boldsymbol{x}_0), \mathrm{d}\,L_{\boldsymbol{f}}h(\boldsymbol{x}_0), \ldots, \mathrm{d}\,L^d_{\boldsymbol{f}}h(\boldsymbol{x}_0) \tag{35}$$

of Lie derivatives (7) of a scalar field h along a vector field $\boldsymbol{f}$ at $\boldsymbol{x} = \boldsymbol{x}_0$ are row-vectors. They can be computed with the C function `Lie_gradientc`, where the result will be stored in a two-dimensional array of size $n \times (d+1)$:

```
int Lie_gradientc(Tape_F,Tape_H,n,x0,d,res)
short Tape_F;      // tape tag of vector field f
short Tape_H;      // tape tag of scalar field h
short n;           // dimension n
double x0[n];      // vector x_0
short d;           // highest degree d
double res[n][d+1]; // gradients of Lie derivatives
```

Different scalar fields $h_1, \ldots, h_m$ can be put together in a vector-valued map $\boldsymbol{h}$ as in Eq. (34). The classical derivative of (34) is a Jacobian matrix of size $m \times n$. The Jacobian matrices

$$\mathrm{d}\,\boldsymbol{h}(\boldsymbol{x}_0), \mathrm{d}\,L_{\boldsymbol{f}}\boldsymbol{h}(\boldsymbol{x}_0), \ldots, \mathrm{d}\,L^d_{\boldsymbol{f}}\boldsymbol{h}(\boldsymbol{x}_0) \tag{36}$$

can be computed at once with the C function `Lie_gradientcv`, where the result will be stored in a three-dimensional array of size $m \times n \times (d+1)$:

```
int Lie_gradientcv(Tape_F,Tape_H,n,m,x0,d,res)
short Tape_F;      // tape tag of vector field f
short Tape_H;      // tape tag of vector map h
short n;           // dimension n
short m;           // dimension m
double x0[n];      // vector x_0
short d;           // highest degree d
double res[m][n][d+1]; // Jacobians
```

In addition, we implemented the C++ function `Lie_gradient` supporting both cases (35) and (36) at once. The computation is discussed in [33].

4.3 Lie Derivatives of a Covector Field

If we consider the elements of $\mathbb{R}^n$ as comlun-vectors, the elements of the associated dual space $(\mathbb{R}^n)^*$ can be written as row-vectors. For programs, there are no differences between a vector field $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^n$ and a covector field $\boldsymbol{\omega} : \mathbb{R}^n \to (\mathbb{R}^n)^*$. The Lie derivative of the covector field $\boldsymbol{\omega}$ along the vector field $\boldsymbol{f}$ is given by (16). The Lie derivatives $L^0_{\boldsymbol{f}}\boldsymbol{\omega}(\boldsymbol{x}_0), \ldots, L^d_{\boldsymbol{f}}\boldsymbol{\omega}(\boldsymbol{x}_0)$ at the point $\boldsymbol{x}_0$ can be computed with the C function `Lie_covector`. The results will be stored in a two-dimensional array (i.e., a matrix) of size $n \times (d+1)$:

```
int Lie_covector(Tape_F,Tape_W,n,x0,d,res)
short Tape_F;          // tape tag of vector field f
short Tape_W;          // tape tag of covector field ω
short n;               // dimension n
double x0[n];          // vector x_0
short d;               // highest degree d
double res[n][d+1];    // Lie derivatives
```

4.4 Lie Derivatives of a Vector Field (Lie Brackets)

Let $\boldsymbol{g} : \mathbb{R}^n \to \mathbb{R}^n$ be a further vector field. The Lie derivative of the vector field $\boldsymbol{g}$ along $\boldsymbol{f}$ is also called Lie bracket and defined by (12). The Lie derivatives $\mathrm{ad}_{\boldsymbol{f}}^k \boldsymbol{g}(\boldsymbol{x})$ along $\boldsymbol{f}$ at the point $\boldsymbol{x}_0$ can be computed with the C function `Lie_vector`. The resulting values of the Lie derivatives $\mathrm{ad}_{\boldsymbol{f}}^0 \boldsymbol{g}(\boldsymbol{x}_0), \ldots, \mathrm{ad}_{\boldsymbol{f}}^d \boldsymbol{g}(\boldsymbol{x}_0)$ will be stored in a two-dimensional array (i.e., a matrix) of size $n \times (d+1)$:

```
int Lie_vector(Tape_F,Tape_G,n,x0,d,res)
short Tape_F;          // tape tag of vector field f
short Tape_G;          // tape tag of vector field g
short n;               // dimension n
double x0[n];          // vector x_0
short d;               // highest degree d
double res[n][d+1];    // Lie brackets
```

The computation of these Lie derivatives using algorithmic differentiation is explained in [28, 31, 35].

5. Lie Derivatives in Nonlinear Control

The Lie derivatives described in the previous sections are often used in nonlinear control. In this section, we will sketch some application for nonlinear control systems of the form

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{g}(\boldsymbol{x})\, u, \quad y = h(\boldsymbol{x}) \tag{37}$$

with the vector fields $\boldsymbol{f}, \boldsymbol{g} : \mathcal{M} \to \mathbb{R}^n$ and the scalar field $h : \mathcal{M} \to \mathbb{R}$ defined on $\mathcal{M} \subseteq \mathbb{R}^n$. Here, $\boldsymbol{x}$ denotes the state, u the input and y the output.

5.1 Exact Input-Output Linearization

Lie derivatives can also be computed along different vector fields, which results in mixed Lie derivatives such as

$$L_{\boldsymbol{g}} L_{\boldsymbol{f}} g(\boldsymbol{x}) = \frac{\partial L_{\boldsymbol{f}} h(\boldsymbol{x})}{\partial \boldsymbol{x}} \boldsymbol{g}(\boldsymbol{x}) \ .$$

System (37) has *relative degree* r in $\boldsymbol{x}_0 \in \mathcal{M}$, if $L_{\boldsymbol{g}} h(\boldsymbol{x}) = 0, \ldots, L_{\boldsymbol{g}} L_{\boldsymbol{f}}^{r-2} h(\boldsymbol{x}) = 0$ for all $\boldsymbol{x}$ in a neighbourhood of x_0, and $L_{\boldsymbol{g}} L_{\boldsymbol{f}}^{r-1} h(\boldsymbol{x}_0) \neq 0$. Roughly speaking, the relative degree is the lowest time derivative of the output y that depends explicitly on the input u:

$$y^{(r)} = L_{\boldsymbol{f}}^r h(\boldsymbol{x}) + L_{\boldsymbol{g}} L_{\boldsymbol{f}}^{r-1} h(\boldsymbol{x})\, u \ . \tag{38}$$

These nonlinearities can be compensated exactly with an additional stabilisation of the resulting linear dynamics using the state-feedback

$$u = -\frac{p_0 h(\boldsymbol{x}) + \cdots + p_{r-1} L_{\boldsymbol{f}}^{r-1} h(\boldsymbol{x}) + L_{\boldsymbol{f}}^r h(\boldsymbol{x})}{L_{\boldsymbol{g}} L_{\boldsymbol{f}}^{r-1} h(\boldsymbol{x})}, \tag{39}$$

where $p_0, \ldots, p_{r-1}$ are the coefficients of the desired characteristic polynomial [20, Chapter 4].

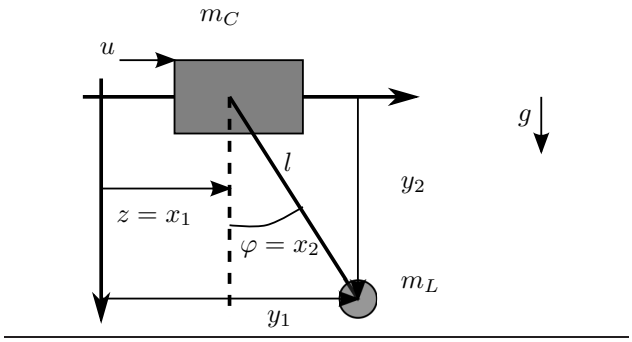


Figure 3. Gantry crane

5.2 High-Gain Observer

The matrix

$$Q(\boldsymbol{x}) := \mathbf{q}'(\boldsymbol{x}) = \begin{pmatrix} \mathrm{d}\, h(\boldsymbol{x}) \\ \mathrm{d}\, L_{\boldsymbol{f}} h(\boldsymbol{x}) \\ \cdots \\ \mathrm{d}\, L_{\boldsymbol{f}}^{n-1} h(\boldsymbol{x}) \end{pmatrix} \tag{40}$$

consisting of gradients of Lie derivatives is called *observability matrix*. If the observability matrix is regular, we can design a high-gain observer

$$\dot{\hat{\boldsymbol{x}}} = \boldsymbol{f}(\hat{\boldsymbol{x}}) + \boldsymbol{g}(\hat{\boldsymbol{x}}) u + \mathbf{k}_{HG}(\hat{\boldsymbol{x}}) \cdot (y - h(\hat{\boldsymbol{x}})) \tag{41}$$

with the state-dependend observer gain

$$\mathbf{k}_{HG}(\hat{\boldsymbol{x}}) = Q^{-1}(\hat{\boldsymbol{x}})\, (p_{n-1}, \ldots, p_0)^T$$

for calculation of estimates $\hat{\boldsymbol{x}}$ of the not directly measured state $\boldsymbol{x}$ using the measured variable y, see [9, 14, 30].

Note that there are many other applications of Lie derivatives in nonlinear control. For example, Lie brackets $\mathrm{ad}_{\boldsymbol{f}}^k \boldsymbol{g}$ are used in controllability analysis [25] and in the design of extended Luenberger observers [49].

6. Example

The usefullness of the approach is illustrated by the example of the control of a gantry crane, as shown in Figure 3.

In this underactuated system one has the mass m_c of the travelling crab, the mass m_L of the load, the length l of the cable, the earth acceleration g and the input force u. The equations of motion are given by

$$\begin{pmatrix} m_L + m_C & m_L l \cos\varphi \\ m_L l \cos\varphi & m_L l^2 \end{pmatrix} \begin{pmatrix} \ddot{z} \\ \ddot{\varphi} \end{pmatrix} + \begin{pmatrix} -m_L l \dot{\varphi}^2 \sin\varphi \\ m_L g l \sin\varphi \end{pmatrix} = \begin{pmatrix} u \\ 0 \end{pmatrix}. \tag{42}$$

Introducing the states $x_1 := z$, $x_2 := \varphi$, $x_3 := \dot{z}$, and $x_4 := \dot{\varphi}$ one obtains a state space representation of (42) which is of the form

$$\dot{\mathbf{x}} = \underbrace{\begin{pmatrix} x_3 & x_4 & * & * \end{pmatrix}^T}_{\mathbf{f}(\mathbf{x})} + \underbrace{\begin{pmatrix} 0 & 0 & * & * \end{pmatrix}^T}_{\mathbf{g}(\mathbf{x})} u \tag{43}$$

with vector fields $\mathbf{f} : \mathbb{R}^4 \to \mathbb{R}^4$ and $\mathbf{g} : \mathbb{R}^4 \to \mathbb{R}^4$.

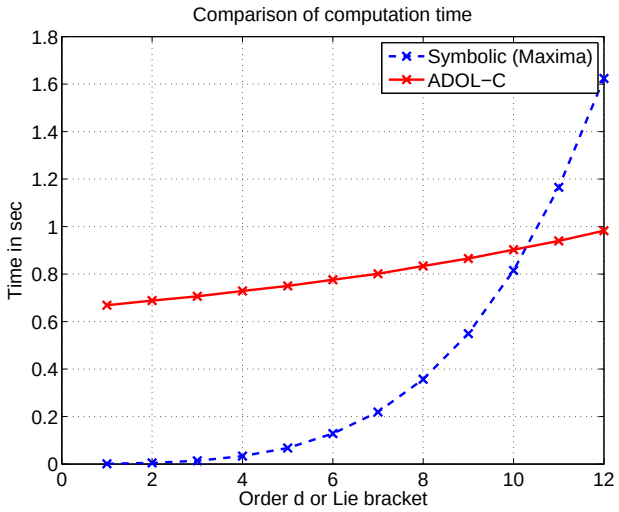


Figure 4. Comparison of computation time for Lie brackets (10) of system (43) using ADOL-C and symbolic expressions ($m_L = m_C = 1\,\text{kg}$, $l = 1\,\text{m}$).

The iterated Lie derivatives (Lie brackets) of the vector field $\mathbf{g}$ along the vector field $\mathbf{f}$ with $n = 4$ and $d = 12$ are calculated using ADOL-C 2.1.5 on a 2.5 GHz Phenom PC under Windows 7. The run time to compute the Lie derivatives for the model (43) of the gantry crane using ADOL-C is shown in Figure 4 (solid line). For a comparison the symbolic expressions for the calculation of the Lie brackets up to order $d = 12$ have been computed using the computer algebra system Maxima. These expressions have been exported as C and Fortran90 code after simplification using the command `trigsimp`. The generation time and source code size can be found in Table 1. The evaulation time of the C-compiled code is given in Figure 4 (dashed line).

Order	C-Code	Fortran90-Code	Generation
1	0.23 kB	0.16 kB	0.060 s
2	0.92 kB	0.77 kB	0.115 s
3	1.76 kB	1.54 kB	0.357 s
4	3.67 kB	3.16 kB	0.795 s
5	6.23 kB	5.46 kB	1.526 s
6	10.71 kB	9.30 kB	2.825 s
7	16.13 kB	14.13 kB	4.949 s
8	24.54 kB	21.61 kB	8.038 s
9	34.52 kB	30.77 kB	12.726 s
10	48.69 kB	43.44 kB	19.458 s
11	65.11 kB	58.63 kB	28.949 s
12	86.92 kB	79.19 kB	42.472 s

Table 1. Source code size and corresponding generation time using Maxima.

As can be seen there is some overhead using ADOL-C when computing low order Lie brackets compared to the evaluation of the compiled C-code of the symbolic expressions. However, in case of ADOL-C computational effort does only slightly increase with the order d of the Lie brackets. Furthermore one circumvents the use of large amount of C- or Fortran90 code as well as large computation times for its generation, cf. Table 1.

7. Conclusions

We presented a package for computation of several kinds of Lie derivatives utilizing automatic differentiation based on the software package ADOL-C. The routines allow the convinient and efficient computation of even high order Lie derivatives of complex systems. An example illustrates the usefulness of the approach. Currently, we are working toward a native integration of the toolbox into ADOL-C.

References

[1] CppAD: A Package for Differentiation of C++ Algorithms. `http://www.coin-or.org/CppAD/`.

[2] Maxima, a computer algebra system. `http://maxima.sourceforge.net/`.

[3] R. Abraham, J. E. Marsden, and T. Ratiu. *Manifolds, Tensor Analysis, and Applications*. Springer, New York, 2nd edition, 1983.

[4] `www.autodiff.org`. Web-Portal über Automatisches Differenzieren.

[5] C. Bendtsen and O. Stauning. FADBAD, a flexible C++ package for automatic differentiation. Technical Report IMM-REP-1996-17, TU of Denmark, Dept. of Mathematical Modelling, Lungby, 1996.

[6] C. Bendtsen and O. Stauning. TADIFF, a flexible C++ package for automatic differentiation. Technical Report IMM-REP-1997-07, TU of Denmark, Dept. of Mathematical Modelling, Lungby, 1997.

[7] G. L. Blankenship and H. G. Kwatny. Computational methods for control of dynamical systems. In N. Munro, editor, *Symbolic methods in control system analysis and design*, volume 56 of *IEE Control Engineering Series*, chapter 15. IEE Press, London, 1999.

[8] B. Christianson. Reverse accumulation and accurate rounding error estimates for Taylor series. *Optimization Methods & Software*, 1:81–94, 1992.

[9] G. Ciccarella, M. Dalla Mora, and A. Germani. A Luenberger-like observer for nonlinear systems. *Int. J. Control*, 57(3):537–556, 1993.

[10] J.-M. Cornil and P. Testud. *An Introduction to Maple V*. Springer, 2001.

[11] M. Dalla Mora, A. Germani, and C. Manes. A state observer for nonlinear dynamical systems. *Nonlinear Analysis, Theory, Methods & Applications*, 30(7):4485–4496, 1997.

[12] B. de Jager. The use of symbolic computation in nonlinear control: Is it viable? *IEEE Trans. on Automatic Control*, 40(1):84–89, 1995.

[13] S. A. Forth. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Transactions on Mathematical Software*, 32(2):195–222, jun 2006.

[14] J. P. Gauthier, H. Hammouri, and S. Othman. A simple observer for nonlinear systems — application to bioreactors. *IEEE Trans. on Automatic Control*, 37(6):875–880, 1992.

[15] R. Giering and Th. Kaminski. Recomputations in reverse mode ad. In G. Corliss, Ch. Faure, A. Griewank, Hascoët, and U. Naumann, editors, *Automatic Differentiation: From Simulation to Optimization*, chapter 33, pages 283–290.

Springer, 2002.

[16] A. Griewank, D. Juedes, and J. Utke. ADOL-C: A package for automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software*, 22:131–167, 1996.

[17] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2nd edition, 2008.

[18] D. D. Holm, T. Schmah, and C. Stoica. *Geometric Mechanicas and Symmetry*. Oxford University Press, 2009.

[19] C. J. Isham. *Modern Differential Geometry for Physicists*. World Scientific, 2 edition, 2001.

[20] A. Isidori. *Nonlinear Control Systems: An Introduction*. Springer-Verlag, London, 3. edition, 1995.

[21] A. Kugi, K. Schlacher, and R. Novaki. *Symbolic Computation for the Analysis and Synthesis of Nonlinear Control Systems*, volume 2 of *Software Studies*, pages 255–264. WIT-Press, Southampton, 1999.

[22] H. G. Kwatny and G. L. Blankenship. *Nonlinear Control and Analytical Mechanics: A Computational Approach*. Birkhäuser, Boston, 2000.

[23] J. M. Lee. *Introduction to Smooth Manifolds*, volume 218 of *Graduate Texts in Mathematics*. Springer, New York, 2006.

[24] M. Lemmen, T. Wey, and M. Jelali. NSAS – ein Computer-Algebra-Packet zur Analyse und Synthese nichtlinearer systeme. Forschungsbericht Nr. 20/95, Gerhard-Mercator-Universität-GH Duisburg, Meß-, Steuer- und Regelungstechnik, 1995.

[25] H. Nijmeijer and A. J. van der Schaft. *Nonlinear Dynamical Control systems*. Springer, New York, 1990.

[26] R. Oloff. *Geometrie der Raumzeit*. Vieweg Verlag, Wiesbaden, 3. edition, 2004.

[27] V. Polyakov, R. Ghanadan, and G. L. Blankenship. Symbolic numerical computational tools for nonlinear and adaptive control. In *Proc. IEEE/IFAC Joint Symposium on Computer-Aided Control System Design*, pages 117–122, Tucson, Arizona, 1994.

[28] K. Röbenack. On the efficient computation of higher order maps $ad_f^k g(x)$ using Taylor arithmetic and the Campbell-Baker-Hausdorff formula. In Alan Zinober and David Owens, editors, *Nonlinear and Adaptive Control*, volume 281 of *Lecture Notes in Control and Information Science*, pages 327–336. Springer, 2002.

[29] K. Röbenack. Automatic differentiation and nonlinear controller design by exact linearization. *Future Generation Computer Systems*, 21(8):1372–1379, 2005.

[30] K. Röbenack. Computation of high gain observers for nonlinear systems using automatic differentiation. *Journal Dynamic Systems, Measurement, and Control*, 127(1):160–162, 2005.

[31] K. Röbenack. Computation of Lie derivatives of tensor fields required for nonlinear controller and observer design employing automatic differentiation. *Proc. in Applied Mathematics and Mechanics*, 5(1):181–184, 2005.

[32] K. Röbenack. Nonlinear controller design based on algorithmic plant description. *Mathematical and Computer Modelling of Dynamical Systems*, 13(2):193–209, 2007.

[33] K. Röbenack and K. J. Reinschke. A efficient method to compute Lie derivatives and the observability matrix for nonlinear systems. In *Proc. 2000 International Symposium on Nonlinear Theory and its Applications (NOLTA'2000), Dresden, Sept. 17-21*, volume 2, pages 625–628, 2000.

[34] K. Röbenack and K. J. Reinschke. Reglerentwurf mit Hilfe des Automatischen Differenzierens. *Automatisierungstechnik*, 48(2):60–66, 2000.

[35] K. Röbenack and K. J. Reinschke. The computation of Lie derivatives and Lie brackets based on automatic differentiation. *Z. Angew. Math. Mech.*, 84(2):114–123, 2004.

[36] R. Rothfuss and M. Zeitz. Einführung in die Analyse nichtlinearer Systeme. In S. Engell, editor, *Entwurf nichtlinearer Regelungen*, pages 3–22. Oldenbourg-Verlag, München, 1995.

[37] C. Rui, I. V. Kolmanovsky, and N. H. McClamroch. Symbolic computation in nonlinear control of multibody space systems. In *ACC'1998 (American Control Conference), Philadelphia*, 1998.

[38] J. Schaffner and M. Zeitz. Variants of nonlinear normal form observer design. In H. Hijmeijer and T. I. Fossen, editors, *New Direction in Nonlinear Observer Design*, volume 244 of *Lecture Notes in Control and Information Science*, pages 161–180. Springer-Verlag, London, 1999.

[39] H. Skaug and D Fournier. Automatic approximation of the marginal likelihood in nonlinear hierarchical models. *Computational Statistics and Data Analysis*, 51(2):699–709, 2006.

[40] S. Stamatiadis, R. Prosmiti, and S. C. Farantos. AUTO_DERIV: Tool for automatic differentiation of a FORTRAN code. *Comput. Phys. Commun.*, 127(2&3):343–355, may 2000. Catalog number: ADLS.

[41] O. Stauning and C. Bendtsen. FADBAD++: Flexible automatic differentiation using templates and operator overloading in ANSI C++. `http://www2.imm.dtu.dk/~km/FADBAD/`.

[42] V. Toth. Tensor manipulation in GPL Maxima. arXiv:cs/0503073v2 [cs.SC], 2005.

[43] V. S. Varadarajan. *Lie Groups, Lie Algebras, and Their Representation*. Springer-Verlag, 1984.

[44] A. Verma. ADMAT: Automatic differentiation for MATLAB using object oriented methods. In *SIAM workshop on object oriented methods*, pages 174–183, 1999.

[45] G. M. von Hippel. Taylur, an arbitrary-order automatic differentiation package for fortran 95. *Comput.Phys.Commun.*, 174:569–576, 2006.

[46] A. Walther, A. Griewank, and O. Vogel. ADOL-C: Automatic differentiation using operator overloading in C++. *Proc. in Applied Mathematics and Mechanics*, 2(1):41–44, 2003.

[47] P. J. Werbos. *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. Wiley, 1994.

[48] S. Wolfram. *The MATHEMATICA Book*. Cambridge University Press, 1999.

[49] M. Zeitz. The extended Luenberger observer for nonlinear systems. *Systems & Control Letters*, 9:149–156, 1987.

Debugging Symbolic Transformations in Equation Systems

Martin Sjölund[1] Peter Fritzson[1]

[1]Department of Computer and Information Science, Linköping University, Sweden,
{martin.sjolund,peter.fritzson}@liu.se

Abstract

How do you debug application models in an equation-based object-oriented (EOO) programming language? Compilers for these tools tend to optimize the model so heavily that it is hard to tell the origin of an equation during runtime.

This work proposes and implements a prototype of a method that is efficient, yet manages to keep track of all the transformations/operations that the compiler performs on the model. The method also considers the ability to collapse certain operations so that they appear to the user as a single expandable operation.

Using such a method enables makers of compilers for EOO programming languages to create debugging tools that contain sufficiently detailed information while still being appealing to the user as they minimize duplicate information.

Keywords debugging, modeling, simulation, compilation, Modelica

1. Introduction

1.1 General

Equation-based object-oriented (EOO) programming languages have significant advantages in describing large models since it is easy to construct a hierarchy of models and connecting them.

However, in order to simulate such models efficiently, EOO simulation tools perform a lot of symbolic manipulation in order to reduce the complexity of models and prepare them for efficient simulation. By removing redundancy, the generation of simulation code and the simulation itself can be sped up significantly. The cost of this performance gain is error-messages that are not very user-friendly due to symbolic manipulation, renaming and reordering of variables and equations. For example, the following error message says nothing about the variables involved or its origin:

```
Error solving nonlinear system 2
  time = 0.002
residual[0] = 0.288956
x[0] = 1.105149
residual[1] = 17.000400
x[1] = 1.248448
...
```

It is usually hard for a typical user of the EOO simulation tools to determine what symbolic manipulations have been performed and why. If the tool only emits a binary executable this is almost impossible. Even if the tool emits source code in some programming language (typically C), it is still quite hard to know what kind of equation system you have ended up with. This makes it difficult to understand where the model can be changed in order to improve the speed or stability of the simulation.

While some tools allow you to export a description of the translated system of equations [17], this is not enough. After symbolic manipulation, the resulting equations no longer need to contain the same variables or structure as the original equations. As a simple example, the resulting equation $r = t$ in (1) holds given that r does not change value during events and $t_{start} = r_{start} = 0$.

$$der(r) = 1.0 \Leftrightarrow r = 1.0 * (t - t_{start}) + r_{start} \Leftrightarrow r = t \quad (1)$$

There are two main aspects of debugging application models. One is debugging the simulation executable itself [18]. In general, simulation tools can emit the value of each variable at each time step so that it is possible to see if the results are correct. However, it is usually not possible to understand the cause(s) of slow simulations by studying such data.

By using profiling techniques [12], it is possible to detect which equations cause slowness in a simulation. By using simple techniques to sample and log high-precision clocks at each time step, you can see how much time is spent calculating each strongly connected component. This essentially means that the profiling returns a set equations that calculate a set of variables. But these are variables in the optimized equation system which makes it is hard to know their origin [6].

Thus, you need to look earlier in the compilation process where the equation system is symbolically optimized. Moreover, to understand the causes of possible erroneous variable values, you also need to be able to look into the chain of symbolic transformations of the original model. This can be regarded as debugging of the model compiler

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zürich, Switzerland.
 Proceedings available at:
http://www.ep.liu.se/ecp/056/
EOOLT 2011 website:
http://www.eoolt.org/2011/

regarding the efficiency of the generated code with your application model as input.

1.2 Comparison with Traditional Debugging

When most people hear the word debugging, they think of a statement or instruction-level debugger that uses breakpoints, like GDB [19]. A debugger is a computer program designed to help a programmer in the task of finding faults in programs. But debugging is not limited to just instruction-level debugging.

But there is a fundamental difference between our needs and that of a debugger for a general-purpose programming language. We need to be able to debug the symbolic transformations/optimizations performed on the equation system defined in our application model.

While you could use a statement-level debugger to find out what is happening in the compiler, it is hard to use for a regular user due to requiring compiler sources and a debug version of the compiler itself. You also need some knowledge about the data structures used since representing data in certain ways improves the performance of many algorithms used in these tools. This generally limits the use of such debuggers to the compiler developers themselves.

However, even if you are a compiler developer, it is problematic to use a regular debugger simply due to the size of the equation systems that you are required to debug as it becomes harder and harder to setup the debugger to only look at a single variable or equation. Thus, these traditional instruction-level debuggers are unsuitable also from a compiler developer's point of view.

Since equation-based modeling languages are quite different from general-purpose programming languages like C, it is understandable that debuggers made for conventional imperative languages do not work as well in our domain. Thus, we need a domain-specific debugger that can show the user the flow of transformations and calculations. This is analogous to how a tool such as ANTLRWorks [4] can show which rules were used to parse some input given a certain ANTLR grammar, where the language to write grammars is a domain-specific language.

We propose an approach to debugging the symbolic optimization and transformation of equation systems that is usable by tool developers and modelers alike. The approach can be used for debugging during translation (compilation), statically after translation and dynamically during simulation runtime. The focus of this article is on the method, producing the information required to later on write an efficient graphical debugger.

1.3 Common Operations on Continuous Equation Systems

In order to create a debugger adapted for debugging the symbolic transformations performed on equation systems, we need to list its requirements. There are many symbolic operations that we may perform on equation systems. The description of operations described below also include a rationale for each operation since it is not always apparent why we perform such operations. There are of course many more operations we may perform than the ones listed below. The operations shown below are the ones we are most concerned with, and the ones that our debugger for models translated by the OpenModelica Compiler [10] should be able to handle.

1.3.1 Variable aliasing

An optimization that is very common in Modelica compilers is variable aliasing. This is due to the connection semantics of the Modelica language. For example, if a and b are connectors with the effort-variable v and flow-variable i, a connection (2) will generate alias equations (3) and (4).

$$connect(a, b) \tag{2}$$

$$a.v = b.v \tag{3}$$

$$a.i + b.i = 0 \Leftrightarrow b.i = -a.i \tag{4}$$

In a result-file, this alias relation can be stored instead of a duplicate trajectory, saving both space and computation time. In the equation system, $b.v$ may be substituted by $a.v$ and $b.i$ by $-a.i$, which may lead to further optimizations of the equations.

1.3.2 Known variables

Known variables are similar to alias variables in that you may perform variable substitutions on the rest of the equation system if you find such an occurrence. For example, (5) and (6) can be combined into (7). In the result-file, you no longer need to store a for each time step; once is enough for known variables, parameters and constants.

$$a = 4.0 \tag{5}$$

$$b = 4.0 - a + c \tag{6}$$

$$b = 4.0 - 4.0 + c \tag{7}$$

1.3.3 Equation Solving

If the tool has determined that x needs to be solved for in (8), we need to symbolically solve the equation, producing a simple equation with x on one side as in (9). Solving for x is not always straight-forward, and it is not always possible to invert user-defined functions such as (10). Since x is present in the call arguments and we cannot invert or inline the function, we fail to solve the equation symbolically and instead solve it numerically using a non-linear solver during runtime.

$$15.0 = 3.0 * (x + y) \tag{8}$$

$$x = 15.0/3.0 - y \tag{9}$$

$$0 = f(3 * x) \tag{10}$$

1.3.4 Expression Simplification

Expression simplification is a symbolic operation that does not change the meaning of the expression, while making it faster to calculate. It is related to many different optimization techniques such as constant folding. We may change the order in which arguments are evaluated (11). Constant subexpressions are evaluated during compile-time (12). We may also rewrite non-constant subexpressions (13) and choose to evaluate functions fewer times than in the original expression (14). We may also use special knowledge about an expression in order to make it run faster (15) and (16).

$$and(a, false, b) \Rightarrow and(false, a, b) \Rightarrow false \quad (11)$$

$$4.0 - 4.0 + c \Rightarrow c \quad (12)$$

$$max(a, b, 7.5, a, 15.0) \Rightarrow max(a, b, 15.0) \quad (13)$$

$$f(x) + f(x) + f(x) \Rightarrow 3 * f(x) \quad (14)$$

$$if\ cond\ then\ a\ else\ a \Rightarrow a \quad (15)$$

$$if\ not\ cond\ then\ false\ else\ true \Rightarrow cond \quad (16)$$

1.3.5 Equation System Simplification

It is of course also possible to solve some equation systems statically. For example a linear system of equations with constant coefficients (17) can be solved using one step of symbolic Gaussian elimination (18), generating two separate equations that can be solved individually after causalization (19). A simple linear equation system as (17) may also be solved numerically using e.g. LAPACK [1] routines.

$$\begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 4 \\ 5 \end{pmatrix} \quad (17)$$

$$\begin{pmatrix} 1 & 2 \\ 0 & -3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 4 \\ -3 \end{pmatrix} \quad (18)$$

$$\begin{aligned} x &= 2 \\ y &= 1 \end{aligned} \quad (19)$$

1.3.6 Differentiation

Symbolic differentiation [7] is used for many purposes. It is used to expand known derivatives (20) or as one operation in index reduction. Jacobian matrices have many applications, e.g. to speed up simulation runtime [5]. The matrix is often computed using automatic differentiation [7] which combines symbolic differentiation with other techniques to achieve fast computation.

$$\frac{\partial}{\partial t} t^2 \Rightarrow 2t \quad (20)$$

1.3.7 Index reduction

In order to solve DAE's numerically, we use discretization techniques and methods to numerically compute derivatives (often referred to as solvers). Certain DAE's need to be differentiated symbolically to enable stable numeric solution. The differential *i*ndex of a general DAE system is the minimum number of times that certain equations in the system need to be differentiated to reduce the system to a set of ODEs, which can then be solved by the usual ODE solvers, Chapter 18 in [9]. While there are techniques to solve DAE's of higher index than 1, most of them require index-1 DAE's (no second derivatives). This makes it more convenient to reformulate the problem using index reduction algorithms [2]. One such technique uses dummy derivatives [13]; this is the algorithm currently used in the OpenModelica Compiler.

1.3.8 Function inlining

Writing functions to do common operations is a great way to reduce the burden of maintaining your code. When you inline a function call (21), you treat it as a macro expansion (22) and may increase the number of symbolical manipulations you can perform on an expression such as (23).

$$2 * f(x, y)/\pi \quad (21)$$

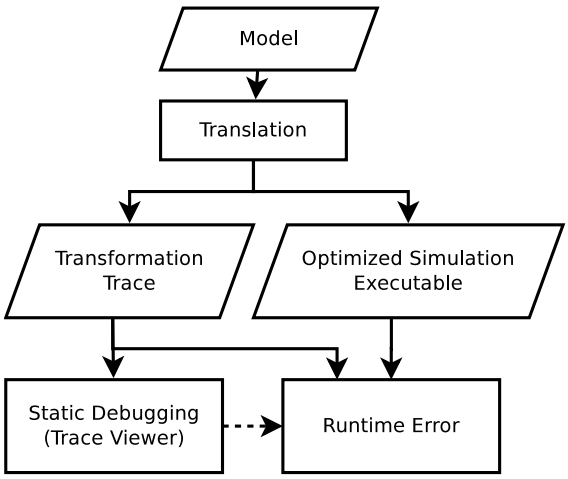
$$2 * \pi * (sin(x + y) + cos(x + y - y))/\pi \quad (22)$$

$$2 * (sin(x + y) + cos(x)) \quad (23)$$

2. Debugging Equation-Based Models

The choice of techniques for implementation of a debugger depends on where and for what it is intended to be used. Translation and optimization of large application models can be very time-consuming so it would be good if the approach has such a low overhead that it can be enabled by default. It would also be good if error messages from runtime could use the debug information from the translation and optimization stages to give more understandable and informative messages to the user (see Figure 1).

Figure 1. Using Information from the Translation in Subsequent Phases



A technique that is commonly used for debugging is tracing (sometimes called logging depending on context). Some programs that output a trace in order to help in the debugging of C applications are `strace` and `valgrind`.

`strace` (`truss` on some UNIX systems) logs all system calls and signals that the attached process receives. `valgrind` is a suite of tools for debugging (and profiling) programs. Its default tool, `MEMCHECK`, helps C programmers that use pointers in unintended ways. It is capable of reporting unreachable blocks of data, double free, accessing recently free'd data, accessing non-allocated data and calling standard C functions with input that has undefined result. Both `strace` and `valgrind` display the information to the user by streaming text.

The simplest way of implementing tracing is to print a message to the terminal or file in order to log the operations that you perform. The problem here is that if you roll back an operation, the log-file will still contain the operation that was rolled back. You also need to post-process the data if you require the operations to be grouped by equation.

A more elegant technique is to treat operations as metadata on equations, variables or equation systems. Other metadata that should already be propagated from source code to runtime include the name of the component that an equation is part of, which line and column that the equation originates from, and more. Whenever we perform an operation, we store the operation kind and input/output inside the equation as a list of operations. If the structure used to store equations is persistent this also works if the tool needs to roll back execution to an earlier state.

The cost of adding this metadata is a constant runtime factor from storing a new head in the list. The memory cost depends a lot on the compiler itself. If garbage collection of reference counting is used, the only cost is a small amount to describe the operation (typically an integer and some pointers to the expressions involved in the operation). If these expressions now referenced by being stored as metadata would normally become garbage and subsequently be deallocated, the memory usage increases, but this is a small amount even for large models[1].

2.1 Bookkeeping of Operations

2.1.1 Variable Substitution

We will consider the elimination of variable aliasing and variables with known values (constants) as the same operation that can be done in a single phase. It can be performed as a fixed-point algorithm where you collect substitutions and record if any change was made (stop if no substitution is performed or no new substitution can be collected). For each alias or known variable, merge the operations stored in the simple equation $x = y$ before removing it from the equation system. For each successful substitution, record it in the list of operations for the equation.

The history of the variable a in the equation system (24) could be represented as a more detailed version (25) instead of the shorter (26) depending on the order in which the substitutions were performed.

$$a = b, b = -c, c = 4.5 \tag{24}$$

$$a = b \Rightarrow a = -c \Rightarrow a = -4.5 \tag{25}$$

$$a = b \Rightarrow a = -4.5 \tag{26}$$

In equation systems that originate from a Modelica model we prefer to see a substitution as a single operation rather than a longer chain of operations (chains of 50 cascading substitutions are not unheard of and makes it hard to get an overview of the operations performed on the equation, even though sometimes all the steps are necessary to understand the reason for the final substitution).

We also plan to collect sets of aliases and select a single variable (doing everything in one operation) in order to make substitutions more efficient. However, alias elimination may still cascade due to simplification rules (27), which means that you need a work-around for substitutions performed in a non-optimal order.

$$a = b - c + d \Rightarrow a = b - b + d \Rightarrow a = d \tag{27}$$

Thus, we compare the previous operation with the new one and if we detect a link in the chain, we store this relation. When displaying the operations of an equation system, it is then possible to expand and collapse the chain depending on the user's needs.

2.1.2 Equation Solving

Some equations are only valid for a certain range of input. When solving an equation like (28), you assert that the divisor is non-zero and eliminate it in order to solve for x. We record a list of the assertions made (and their sources for traceability). An assertion may be removed if we later determine that it always holds or if it overlaps with another assertion (29).

$$x/y = 1 \Rightarrow x = y \quad (y \neq 0) \tag{28}$$

$$y \neq 0, 46.0 < y < 173.3 \Rightarrow 46.0 < y < 173.3 \tag{29}$$

2.1.3 Expression Simplification

Tracking changes to an expression is easy if you have a working fixed-point algorithm for expression simplification (record a simplification operation if the simplification algorithm says that the expression changed). However, if your simplification algorithm oscillates (as in 30) it is hard to use it as a fixed-point algorithm.

$$2 * x \Rightarrow x * 2 \Rightarrow 2 * x \tag{30}$$

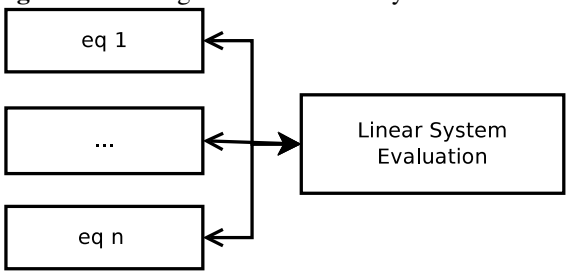
The simple solution is to use an algorithm that is fixed-point, or conservative (reporting no change made when performing changes that may cause oscillating behaviour). Finding where this behaviour occurs is not hard to find for a compiler developer (simply print an error message after 10 iterations). If it is hard to detect if a change has actually occurred (due to changing data representation to use more advanced techniques), you may need to compare the input and output expression in order to determine if the operation should be recorded. While comparing large expressions may be expensive, it is often possible to let the simplification routine keep track of any changes at a smaller cost.

[1] In the OpenModelica Compiler, the memory overhead for such tracing is roughly 40MB in a model with 30,000 equations. Most of that is shared with other data structures.

2.1.4 Equation System Simplification

It is possible to store these operations as pointers to a shared and more global operation or as many individual copies of the same operation. It is preferable to store this as a single global operation (see Figure 2) since the only cost is only some indirection when reading the data. It is also recommended to store reverse pointers (or indices) from the global operation back to each individual operation as well, so that reverse lookup can be performed at a low cost.

Figure 2. Sharing Result of Linear System Evaluation



As the tool we are using performs only limited simplification of these strongly connected components, we are currently limited to only recording evaluation of constant linear systems. As more of these optimizations, e.g. solving for y in (31), are added to the compiler, they will also need to be traced and support added for them in the debugger.

$$\begin{pmatrix} 1 & 1 & 2 \\ 1 & i & 1 \\ & -i & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 15 \\ 18 \\ 18 \end{pmatrix} \tag{31}$$

2.1.5 Differentiation

Whenever we perform symbolic differentiation in an expression, e.g. to expand known derivatives (32), we record this operation in the equation. We currently do not eliminate this state variable as in (33), but if we did the operation would also be recorded.

$$der(x) = der(time) \Rightarrow der(x) = 1.0 \tag{32}$$

$$der(x) = 1.0 \Rightarrow x = time + (x_{start} - time_{start}) \tag{33}$$

2.1.6 Index reduction

For the index reduction algorithm, we record any substitutions made, add source information to the new variable, and the operations performed on the affected equations. As an example for the dummy derivatives algorithm, this includes differentiation of the Cartesian coordinates (x, y) of a pendulum with length L (34) into (35) and (36). After the index reduction is complete, further optimizations such as variable substitution (37), are performed to reduce the complexity of the complete system.

$$x^2 + y^2 = L^2 \tag{34}$$

$$der(x^2 + y^2) \Rightarrow 2 * (der(x) * x + der(y) * y) \tag{35}$$

$$der(L^2) \Rightarrow 0.0 \tag{36}$$

$$2 * (der(x) * x + der(y) * y) \Rightarrow 2 * (u * x + v * y) \tag{37}$$

2.1.7 Function inlining

Since inlining functions may cause a new function call to be added to the expression, we inline functions until a fixed point is reached (with a maximum depth to avoid problems with recursive functions). We also simplify expressions in order to reduce the size of the final expression. When we have completed inlining calls in an equation, we record this is an inline operation with the expression before and after.

2.2 Presentation of Operations

Until now we have focused on collecting operations as data structured in the equation system. What can we do with this information? During the translation phase we can use it directly to present information to the user. Assuming that the data is well structured, it is possible to store it in a static database (e.g. SQL) or simply as structured data (e.g. XML). That way the data can be accessed by various applications and presented in different ways according to the user needs for all of them. Our OpenModelica prototype only outputs text at present; this is sufficient for a proof of concept as the output is human-readable, but it is not suitable for user tools that require structured data in order to be efficient.

The number of operations stored for each equation varies widely. The reason is that when you replace a known variable x with for example the number 0.0, you may start removing subexpressions. You then end up with a chain of operations that loops over variable substitutions and expression simplification. The number of operations performed may scale with the total number of variables in the equation system if you do not limit the number of iterations that the optimizer may take [8]. This makes some synthetic models very hard to debug. The example model in Listing 1 performs $1 + 2 + ... + N$ substitutions and simplifications in order to deduce that $a[1] = a[2] = ... = a[n]$.

Listing 1. Alias Model with Poor Scaling

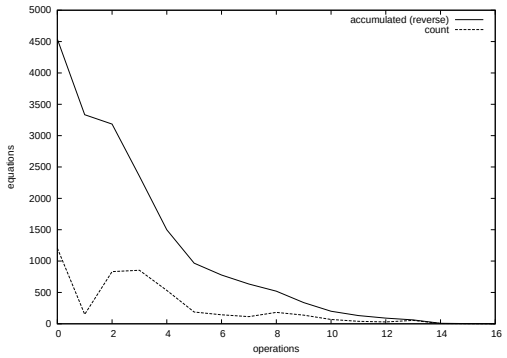
```
model AliasClass_N
  constant Integer N=60;
  Real a[N];
equation
  der(a[1]) = 1.0;
  a[2] = a[1];
  for i in 3:N loop
    a[i] = i*a[i-1]-sum(a[j] for j in
       1:i-1);
  end for;
end AliasClass_N;
```

Using a real-world example, the Engine1a model[2], the majority of equations have less than 10 operations (Figure 3), which is a manageable number to go through when

[2] Modelica.Mechanics.MultiBody.Examples.Loops.Engine1a from MSL 3.1 [15]

you need to debug your model and to know which equations are problematic.

Figure 3. The number of symbolic operations performed on equations in the Engine1a model



2.3 Runtime

In order to produce better error messages during runtime, it would be beneficial to be able to trace the source of the problem. We use the toy example in Listing 2 to show the information that the augmented runtime can display when an error occurs. The user should be presented with an error message from the solver (linear, nonlinear, ODE or algebraic does not matter). Here, the displayed error comes from the algebraic part of the solver. It clearly shows that $log(0.0)$ is not defined and the source of the error in the concrete syntax (the Modelica code that the user may influence) as well as the name of the component (which may be used as a link by a graphical editor to quickly switch view to the diagram view of this component). We also display the symbolic transformations performed on the equation, which will help us debug some problems with the model.

Listing 2. Runtime Error

```
Error: At t=0.5, block1.u = 0.0 is not
    in the domain of log (>0)
Source equation: [Math.mo
    :2490:9-2490:33] y = log(u)
Source component: block1 (MyModel
    Modelica.Blocks.Math.Log)
Flattened equation: block1.y = log(
    block1.u)
Manipulated equation: y = log(u)
  <Operations>
    variable substitution: log(block1.u
        ) = log(u)
  <Depending on equations (from BLT)>
    u <:link>
```

The equation that threw an assertion also depends on other variables. We display these as a link (which opens up a view with the information in Listing 3), enabling a user to debug the input of the equation. Using this view, we quickly realize that v is a constant, which causes the expression $v * (1.5 - v) - time$ to become 0 at some point in time. If v was time-variant, it is possible that the equation could always be calculated.

Listing 3. Runtime Error Dependency

```
Source equation: [MyModel.mo:X:Y-Z:A] u
    = p-time
Source component: <top> (MyModel)
Flattened equation: u = v*(1.5-v)-time
Manipulated equation: u = 0.5-time
  <Expand operations>
    simplify: 0.5*(1.5-0.5)-time =>
        0.5-time
    variable substitution: v*(1.5-v)-
        time => 0.5*(1.5-0.5)-time
```

If the user interface is intuitive enough (i.e. it is possible to follow links), this should give a user all the information he needs to debug his model. All possible error sources (including but not limited to solvers not converging, domain errors, singular equations, assertions) should be able to display such error messages and open up a browser/debugger of the optimized equation system.

This information can be easily made available in generated code by loading the database containing the transformation traces and annotating the source code with the key to the correct row in the table.

3. Conclusions and Further Work

A prototype design and implementation for tracing symbolic transformations and operations in a compiler (the OpenModelica Compiler [11]) for an EOO language (Modelica [14]) has been constructed. It has an overhead in the order of 0.01%. The rest is simply writing the database to file, which is relatively cheap compared to generation and compilation of the generated C code. The implementation also considers that some operations, e.g. cascading alias elimination, should be considered as an expandable operation that combines a simple overview and a detailed view. Further, a design for constructing a debugger using this data has been provided. This design can be used for static debugging, runtime debugging and to provide understandable runtime errors.

The runtime system of OpenModelica [10] simulations needs to be extended to take advantage of the collected information. It currently uses a mixture of text-files and compiled C sources to set parameters, start values, variable names and source information. The choice of using SQL or XML to store the description of the symbolic operations is a difficult one to make since many other tools rely on being able to set parameters in the OpenModelica text-based init-file (this is scheduled to change once OpenModelica supports the FMI interface [3], which uses XML files for initialization of data). While XML could also be used together with an template written in XSLT to display the information in a regular web browser, the overhead of writing and parsing huge XML files (hundreds of MB). We plan to implement our database using SQL as it is a more practical choice given that the database is usually never

needed (if the simulation succeeds and gives the correct result noone needs the debugger). The savings in storage and computation time should make up for the flexibility XML would give us.

A graphical debugger/browser for the transformations also needs to be designed as the current prototype only relies on searching in text-based files. Without this, a regular user will not be capable of taking advantage of the trace. At the time of this writing, the trace is rather cryptic and only available from the command-line, but an advanced user could still take advantage of the current trace (Listing 4) as it is more readable than the source code (Listings 5 and 6).

Listing 4. OpenModelica Trace (Snipped for Brevity)

```
nonlinear: x,y
  residual: xloc[0] - xloc[1] * time;
    operations (3):
      subst:
        x - y * time
        =>
        xloc[0] - xloc[1] * time
      simplify:
        y * (time * 1.0)
        =>
        y * time
      subst:
        y * (time * z)
        =>
        y * (time * 1.0)
  residual: xloc[1] - sin(xloc[0] *
     time);
    operations (1):
      subst:
        y - sin(x * time)
        =>
        xloc[1] - sin(xloc[0] * time)
```

Listing 5. OpenModelica ODE code fragment

```
start_nonlinear_system(2);
nls_x[0] = extraPolate($Px);
nls_xold[0] = $P$old$Px;
nls_x[1] = extraPolate($Py);
nls_xold[1] = $P$old$Py;
solve_nonlinear_system(residualFunc2);
$Px = nls_x[0];
$Py = nls_x[1];
end_nonlinear_system();
```

Listing 6. OpenModelica residualFunc2

```
res[0] = (xloc[0] - (xloc[1] * time));
tmp1008 = sin((xloc[0] * time));
res[1] = (xloc[1] - tmp1008);
```

This work focused mainly on operations performed on individual equations. By analyzing the BLT matrix of the equation system, a user could additionally be presented with equations that need to be solved before or after the selected equation. With more bookkeeping it should even be possible to show how the relationships change during the optimization process.

We also did not consider operations performed on algorithmic code or optimizations that can be performed on the hybrid parts of the system, e.g. dead code elimination of unreachable discrete events.

References

[1] Ed Anderson et al. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1999.

[2] Uri Ascher and Linda Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics, 1998.

[3] Torsten Blochwitz et al. The functional mockup interface for tool independent exchange of simulation models. In Modelica'2011 [16].

[4] Jean Bovet and Terence Parr. ANTLRWorks: an ANTLR grammar development environment. *Software: Practice and Experience*, 38:1305--1332, 2008.

[5] Willi Braun, Lennart Ochel, and Bernhard Bachmann. Symbolically derived Jacobians using automatic differentiation - enhancement of the OpenModelica compiler. In Modelica'2011 [16].

[6] Peter Bunus. *Debugging and Structural Analysis of Declarative Equation-Based Languages*. Licentiate thesis No 964, Linköping University, Department of Computer and Information Science, 2002.

[7] Conal Elliott. Beautiful differentiation. In *International Conference on Functional Programming (ICFP)*, 2009.

[8] Jens Frenkel, Christian Schubert, Günter Kunze, Peter Fritzson, and Adrian Pop. Towards a benchmark suite for modelica compilers: Large models. In Modelica'2011 [16].

[9] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, February 2004.

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

[11] Peter Fritzson et al. Openmodelica 1.7.0, April 2011.

[12] Michaela Huhn, Martin Sjölund, Wuzhu Chen, Christan Schulze, and Peter Fritzson. Tool support for Modelica real-time models. In Modelica'2011 [16].

[13] Sven Erik Mattsson and Gustaf Söderlind. Index reduction in differential algebraic equations using dummy derivatives. *Siam Journal on Scientific Computing*, 14:677--692, May 1993.

[14] Modelica Association. The Modelica Language Specification version 3.2, 2010.

[15] Modelica Association. Modelica Standard Library version 3.1, 2010.

[16] *Proceedings of the 8th International Modelica Conference*. Linköping University Electronic Press, March 2011.

[17] Roberto Parrotto, Johan Åkesson, and Francesco Casella. An XML representation of DAE systems obtained from continuous-time Modelica models. In *Proceedings of the 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pages 91--98. Linköping University Electronic Press, October 2010.

[18] Adrian Pop and Peter Fritzson. Towards run-time debugging of equation-based object-oriented languages. In *Proceedings of the 48th Scandinavian Conference on Simulation and Modeling (SIMS)*, October 2007.

[19] Richard Stallman, Roland Pesch, Stan Shebs, et al. *Debugging with GDB*. Free Software Foundation, 2011.

Modelica code generation from ModelicaML state machines extended by asynchronous communication

Uwe Pohlmann[1] and Matthias Tichy[2]

[1]Software Engineering Group, Department of Computer Science, University of Paderborn, Germany,
upohl@uni-paderborn.de
[2]Organic Computing, Department of Computer Science, University of Augsburg, Germany,
tichy@informatik.uni-augsburg.de

Abstract

Innovation in cyber-physical systems is today largely driven by embedded software. Thus, appropriate approaches have to be employed to handle the complexity that results from the multi-discipline nature of these innovative cyber-physical systems. Modelica as modeling language specifically targets these multi-discipline systems. The UML profile ModelicaML combines the graphical notation of the UML with the sound formal modeling provided by Modelica. ModelicaML currently does not support modeling asynchronous communication which is increasingly required when cyber-physical systems have to coordinate their behavior. In this paper, we present our approach for Modelica code generation from ModelicaML state machines which have been extended by asynchronous communication. We illustrate our approach by an extended two tanks system that contains two distributed controllers which coordinate themselves by message exchange.

Keywords UML 2.2, State Machine, ModelicaML, Modelica, State Graph2, MechatronicUML

1. Introduction

Today's cyber-physical systems such as aircrafts, spacecrafts, or automobiles are large and very complex. Engineers can design them virtually using digital computers before any physical prototypes are built. The development of virtual models of complex systems requires a close collaboration between engineers from different disciplines. To describe the integration of mechanical and electronic components in consumer products the word mechatronics originated in Japan around 1970. Mechatronics has come to mean multidisciplinary systems engineering and is the synergistic integration of physical systems, electronics, controls, and computers through the design process [5].

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zürich, Switzerland.
 Proceedings available at:
http://www.ep.liu.se/ecp/056/
EOOLT 2011 website:
http://www.eoolt.org/2011/

As all elements of a cyber-physical system interact with each other, engineers cannot engineer them independently. Instead, a tight integration in mechatronic design is the key element as complexity has been transferred from mechanical domain to electronic and computer software domains. "Mechatronics is the best practice for synthesis by engineers driven by the needs of industry and human beings" [5].

Modelica is an object-oriented, declarative, multi-domain modeling language for describing and simulating hybrid models. Such models can represent physical behavior, the exchange of energy, signals or other continuous-time interactions between system components as well as reactive, discrete-time behavior. Modelica uses the hybrid differential algebraic equation formalism as a sound mathematical representation. Furthermore, mature compilation and simulation environments for Modelica exist.

ModelicaML is a UML profile which extends the UML with concepts of Modelica and enables the modeler to specify advanced constructs like requirements for Modelica. The motivation of ModelicaML is to integrate Modelica and UML. UML's strength in graphical and descriptive high-level modeling is combined with Modelica's formal executable models for analyses and trade studies. Therefore, we use Modelica also as an action language for UML models. ModelicaML does not only target modelers who are familiar with UML. Modelica modelers will also benefit from using ModelicaML/UML for editing and maintaining Modelica models, because graphical modeling promises to be more effective and efficient than textual representation. The strength and efficiency of UML for system modeling and simulation has been proven in recent years. Common understanding of models for parties involved in development of systems results in high-quality models. Further, the combination of discrete-time and continuous-time simulation gives modelers a great benefit. The concrete ModelicaML profile documentation can be found in the technical report [16]. Pop et al. defined earlier versions of ModelicaML [14, 13].

Modelica and ModelicaML mainly focus on tightly coupled systems. In these systems the components (mechanical, electrical, embedded control, etc.) from different dis-

ciplines are tightly coupled for an optimal system performance [18]. But today's systems are increasingly distributed and form systems of systems. They typically coordinate via message passing. Statecharts respectively state machines are an appropriate modeling formalism for the specification of the coordination of these distributed systems. Currently, ModelicaML does allow state machines but it does not consider sending and receiving messages at transitions of the state machines.

In this paper, we present an extension of ModelicaML state machines for the specification of sending and receiving messages based on a running example. We present the syntax as well as the semantics of this extension. Finally, we explain the translation of these extended state machines to plain Modelica code.

We present the running example in the following section. In section 3, we present the message extensions to the ModelicaML state machines including the code generation to plain Modelica. We show the simulation results of our example in Section 5. After a discussion of related work in Section 6, we conclude in Section 7 and give an overview of future work.

2. Example

An example for a hybrid mechatronic system is a rainwater system with two tanks as shown in Figure 1. The use case for the example is that water flows into *tank 1* when it is raining. The tank collects all the rain until it is full. If the tank is full the additive water flows into the rain drainage. The second tank stands on a lower floor. It is a closed tank with no drainage. Periodically, water is taken from *tank 2*. Therefore, *tank 2* needs always a certain amount of water which it can get from *tank 1*.

Every tank has a sensor which measures the current level of water in the tank. Each tank has one controller. Both controllers communicate with each other via messages. Each controller has a message box which stores incoming messages. Each controller gets the current value of the level in its tank from the sensor. The controller of *tank 1* regulates the position of the valve. This valve blocks the pipe between the two tanks. When the valve is open, the water from *tank 1* flows into *tank 2*.

The controller of *tank 2* has the master role in the communication protocol when both controllers communicate with each other. Accordingly, the left controller has the slave role. The master controller asks the slave controller via messages to send some water. The slave controller commits or rejects this request. If *tank 2* has got enough water, the master controller asks the slave controller to close the valve of *tank 1* again.

3. State Machines with Message Semantics

In many cases discrete controllers have to interact with each other to achieve a common goal. Further, they are often physically separated and arranged in different locations. Therefore, they cannot access the same memory and communicate via shared variables. For this reason they have to use (parameterized)-messages to interact with each other.

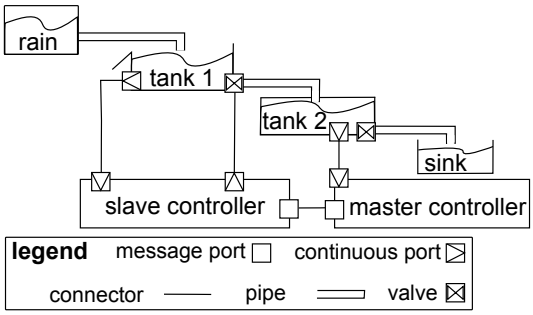


Figure 1. Two Tanks System (cf. [7, p. 391])

The communication needs a protocol as a formal description. We formalize protocols via state machines. Additionally, state machines are a good form to describe discrete behavior of a system.

Figure 2 shows the state machine which represents the master role of the communication protocol between the two controllers and Figure 3 shows the state machine which represents the slave role of the communication protocol between the two controllers. Figure 4 shows the state machines *protocolMasterControl* and *protocolSlaveControl* of both controllers which make the decisions which affect the further execution of the protocol behavior. A possi-

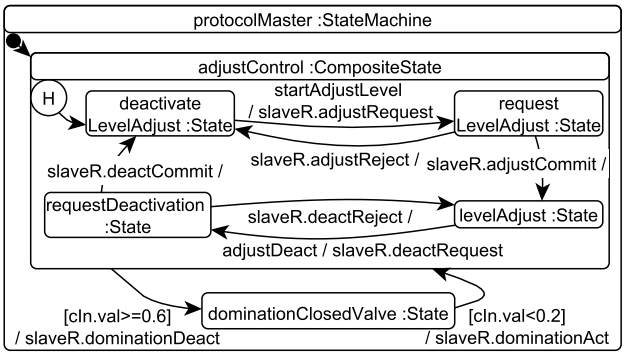


Figure 2. Protocol Behavior of the Master Controller

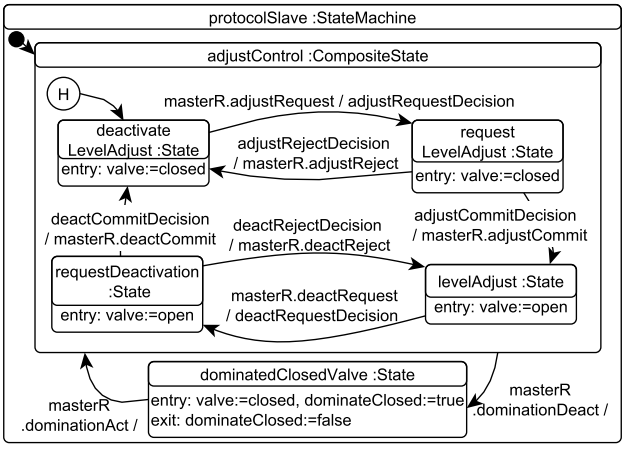


Figure 3. Protocol Behavior of the Slave Controller

ble communication scenario and the resulting sequence of messages are shown in the sequence diagram in Figure 5.

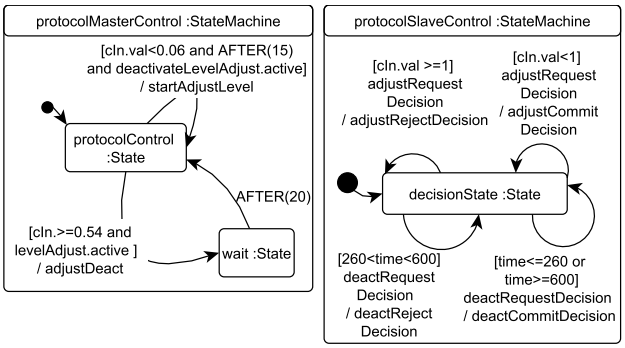


Figure 4. Protocol Control Behavior of the Controllers

For each message that is sent in this example scenario a directed edge is drawn from the sender to the receiver state machine. The *protocolMasterControl* state machine sends the message *startAdjustLevel* to the *protocolMaster* state machine when the level of water in *tank 2* is below a certain level and a minimum of 15 seconds is over. This message starts the protocol behavior. The *protocolMaster* state machine sends the message *adjustRequest* to the slave controller. The slave controller can agree or reject to open its valve. The slave controller opens the valve of *tank 1*, if the state machine *protocolSlaveControl* agreed by sending the message *adjustCommitDecision*. Then water flows from *tank 1* to *tank 2*. The master can ask the slave controller to close the valve if it is in the state *levelAdjust*. The slave controller can reject or commit this proposal.

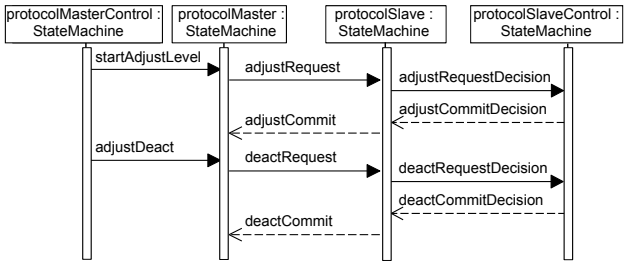


Figure 5. Communication Scenario

To show a more complex state machine with hierarchy and history we include a domination state to the state machines. In this state the master controller dominates the slave controller and forces the slave controller to close its valve if the amount of water of *tank 2* is too high. In real systems this may not be the best solution to model such a behavior. To dominate the slave controller the master controller changes to the state *dominationClosedValve* and forces the slave controller to close the valve of *tank 1*. Before the state change occurs the history state stores the most recent active state of the composite state *adjustControl*. The message *dominationDeact* informs the slave controller about the state change.

The master controller reactivates the most recent active state of the composite state *adjustControl* when the amount of water is below a certain level. The slave controller also reactivates its most recent active state of *adjustControl* when it receives the message *dominationAct* from the master. The whole coordination protocol represents a reactive communication between both controllers.

In the next section we give a brief introduction of syntax and semantics of ModelicaML state machines. Afterwards, we describe the semantics for messages. Messages are currently not defined for ModelicaML state machines. After this, we show a mapping from ModelicaML state machines extended by messages to Modelica code.

3.1 ModelicaML State Machines

State machines mainly consist of states and transitions between states. States can contain regions to add hierarchy or orthogonal behavior. The hierarchy of states and regions builds a tree structure. A configuration of a state machine defines all active states at a point in time.

A state is entered and activated when an incoming transition fires and a state is exited and deactivated when an outgoing transition fires. A state can have an entry-action, which is executed when the state is entered, an exit-action, which is executed when a state is exited, and do-actions, which are executed as long as a state is active.

A transition is enabled when the boolean guard expression is true and, if required, the boolean message variable is true. If more than one transition is enabled the transitions are in conflict with each other. Only the transition with the highest priority of those transitions, which are in conflict with each other, fires. The transition with the lowest priority integer value has the highest priority.

3.2 Semantics Definition by Modelica

In this section we address the semantics of state machines with the target to map this semantics to Modelica language constructs. A special focus is set on the message semantics. We translate state machines into Modelica algorithmic codes. The state machine semantics definition is closely linked to the UML semantics definition. However, there are some issues in which the state machine semantics differs from UML. This is based on the target language Modelica.

The state machine semantics does not change the synchronous data flow principle of Modelica. The developer must explicitly model the real-time characteristics.

The UML defines the behavior of state machines as follows:

> "*Behavior is modeled as a traversal of a graph of state nodes interconnected by one or more joined transition arcs that are triggered by the dispatching of series of (event) occurrences.*" [19, p.564].

The state machines react immediately to each value change of a model variable because Modelica evaluates the algorithm section of the code continuously, namely, after having finished continuous time integration steps and at Modelica event iterations. For a description of the discrete/ continuous modeling/simulation in Modelica based on the synchronous data-flow principle see e. g. Otter et al. [10].

Schamai et al. discuss the execution semantics of ModelicaML state machines [17]. A more detailed definition of ModelicaML state machines syntax and semantics is given by Pohlmann [12].

3.3 Message Syntax and Semantics

A message implementation in Modelica exists in the DEVS approach [15]. We use this approach for sending and receiving messages between different state machines. We define messages by modeling them as operations of the owner class of the state machine. In the two tanks example the messages are defined as operations in the classes *DiscreteMasterController* and *DiscreteSlaveController* which are the model classes of the components *masterController* and *slaveController* as shown in Figure 6. Additionally, the Figure shows the composite structure of the whole system modeled with ModelicaML.

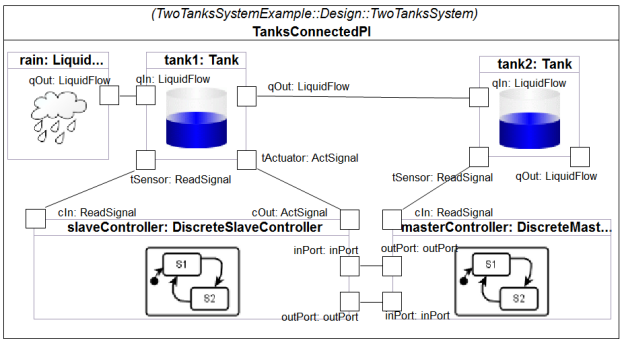


Figure 6. Component Structure

A good way to define messages is to model them as UML operations of the owner class of the state machine. In the two tanks example the messages are defined as operations in the class *BaseController*. The class *BaseController* is the abstract parent class of the model classes *DiscreteMasterCotroller* and *DiscreteSlaveController*. Figure 7 shows the inheritance relation of the involved classes and a part of the needed messages. The state machines of both child classes can use all defined messages. The syntax is the same as in UML for operations.

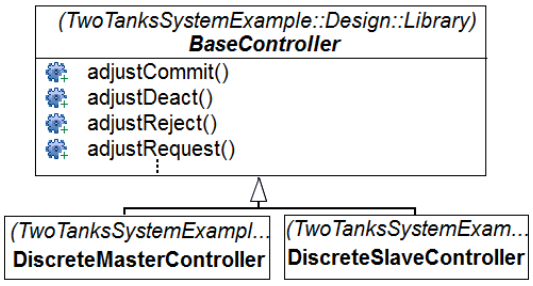


Figure 7. Message Definition of the Controller Classes

In the two tanks example messages could be sent from *slaveController* to *masterController* and vice versa. The destination of a message from *slaveController* to *masterController* is the *inPort* of *masterController*. The *outPort* of *slaveController* is connected via a connector to the *inPort* of *masterController*. In a similar way is the *outPort* of *masterController* connected to the *inPort* of *slaveController*.

In ModelicaML the causality of ports can be specified. They can be declared as input or output ports. In general, a port with causality *input* receives messages and via a port with causality *output* messages are sent. The transmission of messages is instantaneous when no behavior for the connector is defined. The content of a message is independent from its type. Accordingly, a message can contain parameters. Further, various messages can be received simultaneously over an *input*-port or sent via an *output*-port.

An output-port can only be connected to an input-port. An input-port is associated with the destination object. Each message port has its own message box, also called message pool. Additionally, each component has a component message pool which collects all messages from all of its message ports. This makes it easier to search for a certain message, because only the component message pool has to be searched and not all port message pools.

Figure 8 shows the internal representation of the message pools and the message flow via the port between the two controllers. Each component has the input-port message pool *inPortMessagePool* and the component message pool (*controllerMessagePool*).

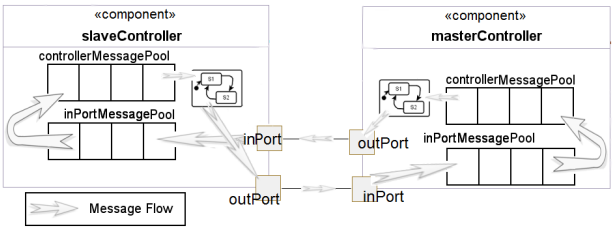


Figure 8. Internal Message Pool Representation and Message Flow

A component monitors the message pools by periodically sampling the status of the message pool. Before the system can react on a received message, the message pool is completely iterated and all received variables are read. For each message type one boolean variable exists. An available message is dispatched from the message pool and the corresponding variable becomes true, if it previously was false. If the message variable is true, the message is put back into the message pool. The message queue is ordered as a FIFO-queue. The execution semantics mostly does not depend on the order of messages in the message pool. It depends only on the order of messages in the message pool if messages from the same type are received via different ports at the same time. In this case we use an explicit order of the input-ports to get an explicit prioritization. We do not differ between messages in the message pool which we received from other components or messages which are raised within the same component. At one event iteration at a time instance it is not possible that multiple transitions react on the same message type. If there are multiple transitions they have to wait until the next event iteration.

Figure 9 shows an example. Both state machines are currently located in the marked states $A2$, $B2$. In the previous step the transition $A1 \rightarrow A2$ sent *message1* and simultaneously, the transition $B1 \rightarrow B2$ sent *message2*. The Figure shows the current state of the component message pools below the components. However, the relevant information

for the execution semantics is the priority order of the enabled outgoing transitions ($A2 \rightarrow A3$ [priority value = 2], $A2 \rightarrow A5$ [priority value = 3]) of state $A2$. Transition $A2 \rightarrow A3$ has the higher priority and fires. *Message2* is dispatched and state $A3$ becomes active.

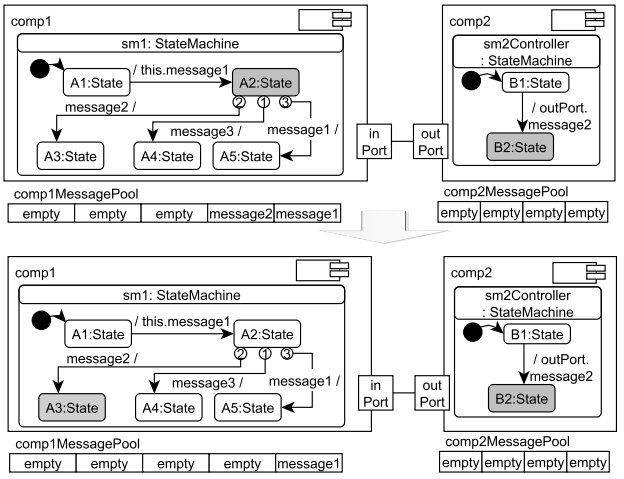


Figure 9. Message Execution Semantics

4. From State Machines to Modelica Code

The translation from ModelicaML models to Modelica consists of two steps. Firstly, the static semantics of the model is validated. The validation process automatically traverses the state machine graph. During this traversing a template checks the model for certain conditions such as if each composite state has at least one initial state. For each found error an error marker is set in the Eclipse problem view. The modeler can start the validation process at any time. Secondly, the modeler can start the code generation process if the model is error free. We use the template-based model-to-text approach of Acceleo[1] for the code generation[2].

4.1 Modelica Code for the State Machine Structure

The code generation of the static part of state machines generates a Modelica record structure for each state machine and its regions and states. A Modelica record is a class for specifying data without behavior. The behavior is located in an algorithm section of the state machine owner model class.

The state machine and each state have the variables *active*, *timeAtActivation*, *stime*, and *selfTransitionActivated*. The variable *active* indicates whether a state is active ($active = 1$) or inactive ($active = 0$). The variable *timeAtActivation* is set to the current simulation time when the state is activated. The variable *stime* is a local timer. The value is calculated by the statement $time - timeAtActivation$. The variable *selfTransitionActivated* is an auxiliary variable, which is needed to identify if a self-transition has been fired. Listing 1 shows the Modelica record definition of a simple state.

[1] http://www.acceleo.org

[2] The developed code generator for ModelicaML state machines without messages is a result of the master thesis of Pohlmann [12] which was done in a cooperation with Schamai from EADS Innovation Works. It is available at http://www.openmodelica.org/index.php/developer/tools/134.

Listing 1. Simple State Record Definition

```
record SimpleState
  Boolean active;
  Real timeAtActivation;
  Real stime;
  Boolean selfTransitionActivated;
end SimpleState;
```

The Modelica record definition of a state machine consists of several variables. The variable *startBehavior* is used to initialize the state machine behavior. Each region has the variable *numberOfActiveStates*. This variable is used to assert that in a region there is never more than one state active at the same time. Listing 2 shows the Modelica structure code for the *protocolMasterControl* state machine shown in the left part of Figure 4.

Listing 2. State Machine Record Definition

```
record masterController_SM_protocolMasterControl
  Boolean active;
  Real timeAtActivation;
  Real stime; // local timer.
  Boolean selfTransitionActivated;
  Boolean startBehavior;
  protocolMasterControl_Region_0 Region_0;
  // REGION records
    record protocolMasterControl_Region_0
    // SIMPLE STATES instantiation
    SimpleState protocolControl;
    SimpleState wait;
    InitialState Initial_0;
    Integer numberOfActiveStates;
   end protocolMasterControl_Region_0;
...
end masterController_SM_protocolMasterControl;
```

A special record is required if, instead of an initial-pseudostate, a *shallowHistory*-pseudostate is used like in Figure 2. The record of the composite state *adjustControl* must have a type with an enumeration of all states of the region. Further, the variable *lastActive* is an instance of this type. Listing 3 shows an example for such a record.

Listing 3. ShallowHistory Record Definition

```
record Region_0_HistoryState
  Boolean active;
  Real timeAtActivation;
  Real stime;
  Boolean selfTransitionActivated;
  type HistoryStateT = enumeration(
     deactivateLevelAdjust, requestLevelAdjust,
     levelAdjust, requestDeactivation);
     HistoryStateT lastActive;
end Region_0_HistoryState;
```

Besides the generation of the state machine structure, the structure for messages is implemented. A message is stored as a record. This record has at least the integer variable *msgType*. The *msgType* value must have a unique

value, because it is the identifier of the message type. Further, a message can have user defined attributes. Listing 4 shows an example of a message record. It has the integer variable *port* which encodes the port-id over which the message should be sent and the integer variable *msgType* which identifies the type of a message. Additionally, attributes which hold parameterisable values could be added.

Listing 4. Message Type Record Definition

```
replaceable record stdMessage
    Integer port;
    Integer msgType;
end stdEvent;
```

For the two tanks example the message type values are encoded as defined in Table 1.

10 = adjustRequest	11= adjustCommit
12 = adjustReject	13 = adjustDeact
14 = deactRequest	15 = deactCommit
16 = deactReject	17 = dominationDeact
18 = dominationAct	19 = startAdjust
20 = startAdjustLevel	21 = adjustRequestDecision
22 = adjustRejectDecision	23 = adjustCommitDecision
24 = deactRequestDecision	25 = deactRejectDecision
26 = deactCommitDecision	

Table 1. Message Type Encoding

Because of the limitations that Modelica cannot handle data structures with a dynamic size, such as a linked list, we implement the message mechanism like Sanz [15] by using Modelica external functions. The external function invokes a C-implementation that stores messages in the dynamic memory. The dynamic memory address of the component message pool is stored in the variable *cmpMsgPoolAdr*. Additionally, for each message port the variables *inputMsgPoolAdr*, *outputMsgPoolAdr* are used. They store the dynamic memory location of the port message pools. The variables *numIn*, *numOut*, *numreceived* are auxiliary variables. For each message a boolean variable exists which has the name of the message. Listing 5 shows the code which is needed for the messages and pools of *component1*.

Listing 5. Message Representations

```
//Number of Ports
parameter Integer numIn = 1
parameter Integer numOut = 1
//Memory address of the pools
Integer cmpMsgPoolAdr;
Integer inputMsgPoolAdr[numIn];
Integer outputMsgPoolAdr[numOut];
// Number of received messages
Integer numreceived;
// Message occurrence variables
 Boolean adjustCommit;
 Boolean adjustReject;
 Boolean adjustDeact;
 Boolean adjustRequest;
 Boolean deactRequest;
 Boolean deactCommit;
 Boolean deactReject;
 ...
```

4.2 Modelica Code for State Machine Behavior

The order of the generated algorithmic code is very important for the semantics of the state machines. The order of the resulting Modelica algorithm code is defined as pseudocode by the Algorithms 4.1, 4.2, 4.3, and 4.4.

At the beginning of the Modelica algorithm code, the initialization of the state machine is stated. Afterwards, the code for message handling is stated, if required. Accordingly, the region codes are generated in the order of their execution order which is defined by a priority value.

Algorithm 4.1: STATEMACHINEBEHAVIORCODE($StateMachine$)

```
initializeStateMachine
messageHandlingStatements
for each Region.sortRegion()
  do {REGIONBEHAVIORCODE(Region)
```

Algorithm 4.2 shows the structure of the region behavior code. The region behavior code starts with the local time management. This local time management sets and calculates the variables *timeAtActivation* and *sTime*. Afterwards, the history nodes, if present, are set to the currently active states. Now the transition code is generated for the region. Accordingly, the code for the do-actions is generated. At the end of the region behavior code the region behavior code of composite states and submachine states is stated. The code generation invokes the Algorithm 4.2 REGIONBEHAVIORCODE recursively at this point.

Algorithm 4.2: REGIONBEHAVIORCODE($Region$)

```
set local Time Behavior
set History Node to Current Active State
TRANSITIONBEHAVIORCODE(Region)
execute do code
for each CompositeState
  do { for each Region.sortRegions()
       do {REGIONBEHAVIORCODE(Region)
for each SubMachineState
  do { for each Region.sortRegions()
       do {REGIONBEHAVIORCODE(Region)
```

Algorithm 4.3 shows the structure of the transition behavior code. The code starts with the behavior code for the initial- and shallowHistory-pseudostates. These states are auxiliary states and are directly processed in favor. The behavior code for each state is nested within an if-clause. This clause is only processed if the parent state is still active. Therefore, it is ensured that transitions can never fire if the parent is not active.

Algorithm 4.3: TRANSITIONBEHAVIORCODE($Region$)

```
initialBehaviorCode
shallowHistoryBehaviorCode
if (parent is still active)
  then { for each State
         do { if (state is pre active)
              then {TRANSITONCODE(State)
```

Algorithm 4.4 shows the structure of a simple transition code. The concrete execution of a compound transition depends on the pseudostates involved in the chain of transitions from one state to another. This example directly connects two states. The transition with the highest priority is in the if-clause. Then the other transitions follow in the order of their priority in the else-if-clauses. If the guard is empty the condition is set to true. A trigger is optional. Entry-, and exit-actions are only generated if available.

Algorithm 4.4: TRANSITIONCODE(*State*)

comment: First Transition (Highest Priority)

if guard [and trigger]

then
- execute exit-action
- deactivate active state
- execute effect
- activate transition target state
- execute entry-action

comment: Second Transition (Lower Priority)

else if guard [and trigger]

then
- execute exit-action
- deactivate active state
- ...

In the following sections we describe the Modelica code for particular elements.

State Machine Initialization State machines are initialized when the simulation is started. The initialization sets all initial- or history-pseudostates of all regions to true.

Further, the message pools for the component and all discrete input-ports are created. Listing 6 shows the algorithmic code which creates the needed message pools initially. Listing 7 shows the invoked Modelica function which calls the corresponding *C*-Code <*events.c*>[3] of a message pool from Sanz.

Listing 6. Initialization of Message Pools

```
for i in 1:numIn loop
    inputMsgPoolAdr[i] := CreatePool();
end for;
cmpMsgPoolAdr := CreatePool();
end if;
```

Listing 7. Modelica Function which creates a new pool for messages in the dynamic memory

```
function CreatePool
output Integer q;
external "C" q =  QCreate();
  annotation (Include="#include <events.c>");
end CreatePool;
```

Message and MessageTrigger A message event is the dispatch of an asynchronous message instance. A boolean message trigger represents the receipt of this instance. A message can be created and sent by invoking a special *sendMessage* Modelica-function by a transition action. The message function is similar to the *sendEvent*-function of Sanz [15]. The *sendMessage* function is shown in listing 8.

Listing 8. Sends a message to a pool

```
function sendMessage
  input Integer poolAdr;
  input stdMessage e;
  output Integer out;
external "C" out =
        QAdd(poolAdr,e.port,e.msgType,e.value,0);
  annotation (Include="#include <events.c>");
end sendMessage;
```

At the beginning of the state machine behavior code the message handling code is stated. This code has the function to handle incoming messages and to put them from the different input-ports into the message pool of the component. Further, it handles the occurrence of messages in the message pool. If a message occurs and the corresponding variable is false, it sets the variable to true.

Listing 9 shows an example of the message handling. In the first for loop all messages from all input-ports are collected in the component message pool. The address of the memory location of the message pool is stored in the variable *cmpMsgPoolAdr*. Afterwards, the component message pool is searched for messages by the statement `message := getMessage(cmpMsgPoolAdr);`. The value of the variable *message.msgType* is used to check if a certain message is in the pool. If a message is in the pool then the corresponding message variable is set to true. For example if a message of `message.msgType=10` is in the pool and the variable *adjustRequest* is false, then the variable is set to true. Otherwise the message is put back into the message pool by the statement `sendMessage(cmpMsgPoolAdr,message)`.

Listing 9. messageHandlingStatements

```
for i in 1:numIn loop
    numreceived := numMessages(inputMsgPoolAdr[i]);
    for j in 1:numreceived loop
        message := getMessage(inputMsgPoolAdr[i]);
        message.port := i;
        sendMessage(cmpMsgPoolAdr,event);
    end for;
end for;
numreceived := numMessages(cmpMsgPoolAdr);
for j in 1:numreceived loop
    message := getMessage(cmpMsgPoolAdr);
    if message.msgType == 10
        and adjustRequest == false) then
        adjustRequest := true;
    ...
    else
        sendMessage(cmpMsgPoolAdr,message);
    end if;
...
```

A transition, which wants to react on the occurrence of a specific message, uses the boolean message variable as reference in its guard. When the message is available the guard becomes true and the transition fires and the transition action is executed. The transition action

[3] http://www.euclides.dia.uned.es/DESLib/Files/DESLib_1.2.zip

resets the message variable back to false. For example in Figure 2 the transition from $requestLevelAdjust \rightarrow deactivateLevelAdjust$ has as guard the variable *adjustReject* and the transition action statement `adjustReject:=false`. At the moment when the message *adjustReject* arrives from *protocolSlave* the variable is set to true by the message handling. The transition fires and resets the variable back to false, so that the next time when the message is available it could be set to true.

Local Time Behavior Each state has the time variables *timeAtActivation* and *stime*. The variable *timeAtActivation* is set to the current time when a state is entered. The variable *stime* is set in each integration step to the difference of the current simulation time and the *timeAtActivation*. If a state is not active the local timer *stime* is set to zero.

Initial/ShallowHistory-Pseudostate Behavior If the *initial*-pseudostate is active it is deactivated and the target of the outgoing transition is activated. If the *shallowHistory*-pseudostate is active the most recent active state is activated. If no most recent active state exists the target of the outgoing transition is activated.

Simple Transition Behavior A simple transition is a transition which directly connects two states. It is activated if the source state is pre-active. The Modelica pre-function is used to ensure that a state is active at the beginning of the event iteration. Therefore, it is not possible that a state becomes active and not active during the same iteration of an algorithm section.

For example if the state *deactivateLevelAdjust* from the state machine *protocolMaster* (see Figure 2) is active, the message *startAdjustLevel* has already arrived and therefore the boolean variable *startAdjustLevel* is true. Then the transition fires and sends the message *adjustRequest* to the state machine *protocolSlave* (Figure 3) of component *slaveController*. Listing 10 shows the resulting Modelica code.

Listing 10. Simple Transition Behavior

```
if pre(protocolMaster.Region_0
  .adjustControl.active) then
if pre(... .adjustControl.
    deactivateLevelAdjust.active) then
 if  startAdjustLevel then //guard
    //...; exit behavior
    ... .deactivateLevelAdjust.active := false;
    //start effect behavior
    startAdjustLevel:=false;
    message.msgType:=10;
    sendMessage(pre(cmpMsgPoolAdr,message);
    //end effect behavior
    ... .requestLevelAdjust.active := true;
     // ... ; entry behavior
  end if;
end if;
end if;
```

Highlevel Transition Behavior Highlevel transitions are outgoing transitions of composite or submachine states. When a highlevel transition deactivates these hierarchical states it has to be ensured that all active nested substates are deactivated before the composite-/ submachine is deactivated. Consider the state machine in Figure 2. Suppose, the state *adjustControl* is activate and the guard `cIn.val>=0.6` is true. Then the currently active state *deactivateLevelAdjust*, *requestLevelAdjust*, *levelAdjust* or *requestDeactivation* is deactivated. Afterwards, the state *adjustControl* itself is deactivated. Accordingly, the highlevel transition $adjustControl \rightarrow dominationClosedValve$ is taken and the message *dominationDeact* with message type identifier *17* is sent to the *protocolSlave* state machine. Finally, state *dominationClosedValve* is activated. Listing 11 shows the corresponding Modelica code.

Listing 11. Highlevel Transition Behavior

```
if pre(protocolMaster.Region_0
                .adjustControl.active) then
  if(cIn.val>=0.6) then
    if (...deactivateLevelAdjust.active) then
      ...deactivateLevelAdjust.active:=false;
    end if;
    if (...requestLevelAdjust.active) then
      ...requestLevelAdjust.active:=false;
    end if;
    ...
    ...adjustControl.active := false;
    message.msgType:=17;
    sendMessage(pre(outputMsgPoolAdr[1]),message);
    ...dominationClosedValve.active := true;
  end if;
end if;
```

5. Evaluation

We modeled the rainwater two tanks system as shown in Figure 1 with ModelicaML. The resulting composite structure of the system is shown in Figure 6. We modeled the sketched state machines of Figures 2, 3, and 4 with ModelicaML state machines and embedded them as behavior of the model classes *DiscreteMasterController* and *DiscreteSlaveController*. Further, we extended this model with our Modelica-Functions, like *sendsMessage* as shown in Listing 8. Further, we added Modelica code for the message passing and variables for our message types to the ModelicaML model. We generated from that ModelicaML model the whole Modelica code. The model and the generated code is available online at[4]. We simulated it with Dymola 7.4.

Figure 10 shows a part of the simulation results. The variable *rain.qOut.lflow* in the upper diagram shows the amount of rain which flows into *tank 1*. The inflow of *tank 1* is not controllable by the slave controller of *tank 1*. At simulation $time = 16$ the rain decreases. The variable *tank1.qOut.lflow* shows the amount of water which flows from *tank 1* into *tank 2* when the valve is opened. The valve is opened at simulation $time = 15$. Then $0.6\frac{m^3}{s}$ water leave *tank 1* for 24.3 seconds. The diagram in the middle shows the resulting water level in both tanks. When the valve is opened the water level of *tank 1* decreases and the water level of *tank 2* increases. In our example *tank*

[4] `http://www.cs.uni-paderborn.de/fileadmin/Informatik/FG-Schaefer/Personen/upohl/downloads/TwoTanksSystemExample.zip`

2 is bigger than *tank 1* and therefore the amount of water in *tank 2* increases slower than the decrease of water in *tank 1*. The lower diagram shows the state *levelAdjust* of the master controller. At $time = 15$ the state *levelAdjust* become active, because the self-transition of state *protocolControl* in state machine *protocolMasterControl* fires (see Figure 4). This starts the communication between both controllers to open the valve. At $time = 39.3$ the state *levelAdjust* is deactivated and the valve is closed.

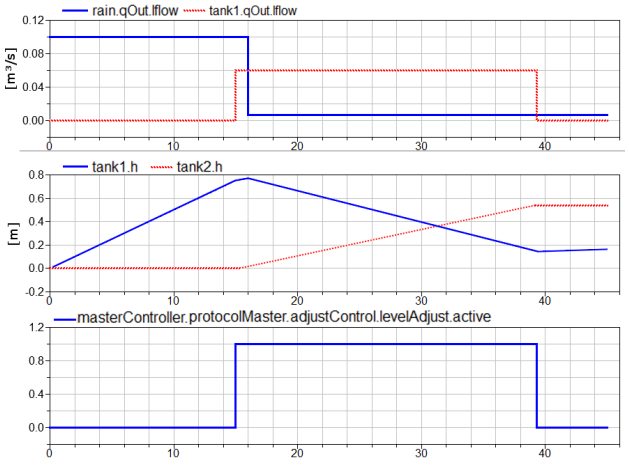


Figure 10. Simulation Results

The sequence diagram in Figure 11 shows the communication that appeared within the simulation. The messages coincide with our predicted scenario (see Figure 5). For each message that is sent a directed edge is drawn from the sender to the receiver state machine. Additionally, the time when the message is sent is annotated below the edge. When a message is drawn below another message and has the same time of sending it is sent after the upper message. Currently, ModelicaML only supports the specification of timing behavior in a rudimentary way. Therefore, we do not specify discrete timing behavior in our protocol. As a result it is difficult to grasp the timing behavior of the discrete controller and the communication protocol. This can be seen within the sequence diagram, because several messages that are sent one after another have the same time of sending.

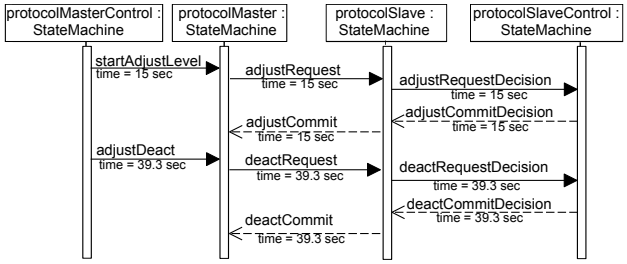


Figure 11. Communication Simulation with Concrete Time Annotation

6. Related Work

6.1 MechatronicUML

MechatronicUML [2, 9, 8, 4] is an approach for the model-driven development and verification of mechatronic real-time systems. The MechatronicUML refines the UML in order to make it applicable to mechatronic real-time systems. The component-based architecture of mechatronic systems is modeled using discrete and continuous components. Their discrete behavior is modeled using real-time statecharts, an extension of UML state machines with constructs from timed automata [1]. Continuous behavior is modeled with the help of the CAMeL-View [3] tool which allows the object-oriented modeling of different parts of mechatronic systems like multi-body system dynamics, control technology, and hydraulics. The MechatronicUML focuses on the real-time coordination between mechatronic systems and supports its formal verification with respect to safety properties. The presented extension of ModelicaML with message exchange has been based on the MechatronicUML. However, it enables the seamless modeling of mechatronic systems using a single formalism and tool in contrast to the MechatronicUML.

6.2 StateGraph2

The Modelica StateGraph2 library [11] is a free Modelica library providing components to model discrete events, reactive and hybrid systems with deterministic hierarchical state diagrams. It utilizes Modelica as action language. Via special blocks actions can be defined in a graphical way depending on the active step. StateGraph does not support a comprehensive set of state machines as defined in the UML and reused in ModelicaML. Larger StateGraph models are not easy to grasp as they are graphically more complex than UML state machines. Additionally, in contrast to our approach, StateGraph does not support message exchange.

6.3 SimulationX

The SimulationX modeling and simulation tool does directly support the specification of a subset of UML state machines [6]. The subset is rather restricted as they do not support orthogonal states. Orthogonal states are an important feature as they enable the modeling of parallel activities and, thus, can greatly reduce the model complexity. They also provide no mechanism like submachines to group a state machine into multiple parts and to reduce the visual complexity and to reuse once defined state machines. Furthermore, they do not support asynchronous message exchange.

7. Conclusion and Future Work

In this paper, we have presented an extension to ModelicaML for the specification of message exchange. We have illustrated our extension by a rainwater two tanks system using two controllers which exchange messages for their coordination. Furthermore, we presented how we translate the ModelicaML models to plain Modelica code and a helper C-function and finally, showed a simulation run. Our translation to Modelica code has been implemented as

plugins for the eclipse case tool Papyrus[5]. We are working on finishing the code generation with respect to messages.

Currently, we work on a new version of our translation directly to the StateGraph2 library. We want to combine StateGraph2 with algorithmic code for constructs which depend on a particular order of execution. Translating to StateGraph2 has the advantage that the resulting models are structured similarly to the ModelicaML state machines and are, thus, easier to understand by the developer than the generated Modelica code. Though, we have to extend the StateGraph2 library to support message sending and receiving for that translation.

Furthermore, we want to add more syntactical constructs to ModelicaML state machines for the better specification of temporal behavior. Specifically, clocks, time guards and invariants as used in timed automata will be added to ModelicaML state machines. Finally, we want to translate simulation runs done in a Modelica tool back to ModelicaML. For example, the states and transitions which are taken during a simulation run can be appropriately visualized using sequence diagrams.

Acknowledgments

This work was developed in the project 'ENTIME: Entwurfstechnik Intelligente Mechatronik' (Design Methods for Intelligent Mechatronic Systems). The project ENTIME is funded by the state of North Rhine-Westphalia (NRW), Germany and the EUROPEAN UNION, European Regional Development Fund, 'Investing in your future'.

Matthias Tichy is currently on leave from the Software Engineering Group at the University of Paderborn.

References

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] Steffen Becker, Stefan Dziwok, Thomas Gewering, Christian Heinzemann, Uwe Pohlmann, Claudia Priesterjahn, Wilhelm Schäfer, Oliver Sudmann, and Matthias Tichy. Mechatronicuml - syntax and semantics. Technical Report tr-ri-11-325, Software Engineering Group, University of Paderborn, Heinz Nixdorf Institute, 2011.

[3] Sven Burmester, Holger Giese, Stefan Henkler, Martin Hirsch, Matthias Tichy, Alfonso Gambuzza, Eckehard Münch, and Henner Vöcking. Tool support for developing advanced mechatronic systems: Integrating the fujaba real-time tool suite with camel-view. In *Proceedings of ICSE*, pages 801–804, 2007.

[4] Sven Burmester, Holger Giese, and Matthias Tichy. Model-driven development of reconfigurable mechatronic systems with mechatronic uml. In *Proceedings of MDAFA*, pages 47–61, 2004.

[5] Kevin C. Craig. Mechatronic system design. In *Proceedings of the Motor, Drive & Automation Systems Conference*, 2009.

[6] Ulrich Donath, Jürgen Haufe, Torsten Blochwitz, and Thomas Neidhold. A new approach for modeling and verification of discrete control components within a Modelica environment. In *Proceedings of the 6th Modelica Conference, Bielefeld*, pages 269–276, 2008.

[7] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, 1st edition, 2004.

[8] Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver Oberschelp. Modular design and verification of component-based mechatronic systems with online-reconfiguration. In *Proceedings of 12th ACM SIGSOFT FSE*, pages 179–188, 2004.

[9] Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer, and Stephan Flake. Towards the compositional verification of real-time uml designs. In *Proceedings of 9th ESEC and 11th ACM SIGSOFT FSE*, pages 38–47. ACM Press, 2003.

[10] Martin Otter, Hilding Elmqvist, and Sven E. Mattsson. Hybrid modeling in Modelica based on the synchronous data flow principle. In *Proceedings of CACSD*, pages 151–157, 1999.

[11] Martin Otter, Martin Malmheden, Hilding Elmqvist, Sven E. Mattsson, and Charlotta Johnsson. A New Formalism for Modeling of Reactive and Hybrid Systems. In *Proceedings of the 7th Modelica Conference*, pages 364–377, 2009.

[12] Uwe Pohlmann. A uml based modeling language with operational semantics defined by modelica. Master thesis, University of Paderborn, Department of Computer Science, Software Engineering Group, 2010. Master Thesis.

[13] Adrian Pop, David Akhlevidiani, and Peter Fritzson. Integrated UML and modelica system modeling with ModelicaML in Eclipse. In *Proceedings of the 11th IASTED*, 2007.

[14] Adrian Pop, David Akhlevidiani, and Peter Fritzson. Towards unified system modeling with the ModelicaML UML profile. In *Proceedings of EOOLT*, pages 13–24, 2007.

[15] Victorino Sanz, Alfonso Urquia, and Sebastian Dormido. Introducing messages in modelica for facilitating discrete-event system modeling. In *Proceedings of EOOLT*, pages 83–93, 2008.

[16] Wladimir Schamai. Modelica modeling language (modelicaml) : A uml profile for modelica. Technical report, Linköping University, Department of Computer and Information Science, The Institute of Technology, 2009.

[17] Wladimir Schamai, Uwe Pohlmann, Peter Fritzson, Christiaan J.J. Paredis, Philipp Helle, and Carsten Strobel. Execution of uml state machines using modelica. In *Proceedings of EOOLT*, pages 1–10, 2010.

[18] Rajarishi Sinha, Vei-Chung Liang, Student Member, Christiaan J. J. Paredis, and Pradeep K. Khosla. Modeling and simulation methods for design of engineering systems. *Journal of Computing and Information Science in Engineering*, 1:84–91, 2001.

[19] Object Management Group UML. Unified modeling language, superstructure, v2.2. Technical report, 2009.

[5] `http://www.openmodelica.org/index.php/developer/tools/134`

Real-time Oriented Modeling Languages and Tools

Using Equation-Based Languages for Generating Embedded Code for Smart Building Applications

Gregory Provan[1]

[1]Computer Science Department, University College Cork, Cork, Ireland, g.provan@cs.ucc.ie

Abstract

While significant research has been done on applying equation-oriented object languages, such as Modelica, to the simulation of complex systems, much research remains to use such languages for generating application-specific embedded code. We describe a method for using a hybrid system language (as a reference model), from which we generate reduced-order models suitable for creating embedded code for tasks such as control and diagnostics. We apply our approach to the generation of embedded diagnostics code for the operation of heating, ventilation and air-conditioning (HVAC) for complex buildings.

Keywords Embedded systems, efficient code generation, model-driven development

1. Introduction

The design of buildings that can simultaneously reduce energy use and integrate multiple building services, such as heating, ventilation and air-conditioning (HVAC), lighting, security, and fire&safety, is receiving widespread attention. Optimised building performance can save significant amounts of energy and reduce greenhouse-gas emissions, and integrating different building operations can enhance such optimisation and also lead to cheaper building management solutions[1]. Equation-oriented object languages, such as Modelica [12], play a major role in such design.

The major focus of building optimization research, to date, has been the simulation of energy use, in order to design structures that minimize energy usage [25]. However, little research has focused on extending languages like Modelica to enable the design of the embedded code that is used to control the building services once the design process is completed. As a consequence, the design of building structures remains decoupled from the design of key embedded services such as HVAC controls, diagnostics and maintenance.

Some tools are now appearing to integrate design of structures and of embedded code. For example, Wetter [26] has proposed a tool that uses a Modelica building simulation library for energy simulation, in conjunction with MATLAB and Ptolemy for control simulation. Such a tool, although a significant advance over energy-simulation tools such as EnergyPlus[2], lacks suitable approaches for actually generating embedded-systems code, as it is focused on simulation. Furthermore, several embedded applications require information beyond just simulation behaviours of nominal conditions. For example, diagnostics requires knowledge of failure modes and their behaviours, and fire&safety systems require the ability to identify the effects of fire and smoke, in order to design escape routes.

Given the use of equation-based languages for system specification, we describe a framework based on model transformation for generating embedded code designed for specific applications, such as diagnostics or control. Our contributions are as follows.

- This article proposes a framework for embedded systems design that uses a high-fidelity model, called a reference model Φ_R, as the basis for (a) the simulation of a system during the design phases, and (b) the generation of embedded code for operation of the system. We adopt a hybrid systems language (defined using Modelica) that explicitly defines system operational and failure modes, thereby extending such languages to include mode-based specifications.
- We develop a model library for HVAC with mode-specifications and fidelity-level tailored to generating embedded code for control and diagnostics purposes.
- We describe model transformation techniques that can be used to transform Φ_R into a languages suitable for embedded diagnostics.
- We apply our approach to the area of energy-efficient buildings, and demonstrate how to generate reduced-order models for diagnostics code that result in computationally efficient embedded diagnostics code.

[1] See www.smart2020.org

[2] http://apps1.eere.energy.gov/buildings/energyplus/

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zürich, Switzerland.

 Proceedings available at: http://www.ep.liu.se/ecp/056/

EOOLT 2011 website: http://www.eoolt.org/2011/

2. Related Work

Our work is based on prior research in the area of building energy simulation and design, model-based code generation and reduced-order modeling.

Energy simulation: Building energy simulation tools [5] typically focus on whole-building simulation, which includes both the building envelope and the energy-delivery systems, like HVAC. Such tools provide users with important building performance indicators, such as energy usage and demand, energy costs, and parameters such as temperature, humidity, light-level, etc. These tools provide a variety of capabilities, but the majority (e.g., EnergyPlus, TRNSYS) can be viewed as black-box or grey-box tools, in that they hide the underlying energy diffusion equations from the user, as well as the actual simulation code.

Modelica is gaining increasing usage for building energy simulation, and unlike leading tools such as EnergyPlus or TRNSYS, exposes its fundamental equations within an object-oriented framework. Several libraries exist for HVAC simlation, e.g., [26]. However, for the purposes of generating embedded code, the existing model libraries are too detailed, and their use complicates the code generation process. As a consequence, we have developed simpler model components that suit our particular needs better.

Model-Based Code Generation: Model Driven Development (MDD) is a field in which significant efforts are being deployed, typically for pure software development from requirements through to embedded code. The Object Management Group, Inc. (OMG) has raised the level of interest in model transformation through standardization of modeling formalisms, e.g., UML or SysML, and of model-based code generation. Hundreds of tools exist for the purpose of requirements capture, modeling and code generation. However, industrial-strength, mature model-to-model transformation systems remain the subject of significant research. In particular, where we transform equation-based models to other models or code, e.g., [3, 13], relatively little research has been done.

In this article we apply standard model transformation methods [6] to a Modelica model (the target language), in order to generate a model in a more specific, lower-fidelity diagnosis model (the target language). Our innovation is not the model transformation process, but the specification of rules for a model-to-model transformation in which the models differ significantly in terms of fidelity, semantics, and purpose. In contrast, most model-to-model transformations in the literature are between models that are relatively similar.

There are several existing code generators for Modelica. The standard code-generation approach creates code (e.g., C code) for simulating the Modelica model [15]. Other generators create control code for subsets of the Modelica model, e.g., [8]. Our approach differs from these generators in that it uses a model transformation approach to generate code defined according to source and target metamodel specifications. Hence, instead of using flags to identify subsets of a Modelica model and compile them separately, e.g., as in [8], our approach can work on the full Modelica model.

Our approach is most closely related to the MetaModelica framework [11]. In this article we provide an example of using model transformation for generating embeddable diagnosis models from Modelica models for an HVAC system. We explicitly specify the transformation rules that map the Modelica objects into the diagnosis language objects. Note that we apply manually-generated rules in this article; in other work we are exploring automated methods to create these rules [2].

Reduced-order modeling: Model reduction is well established within the engineering literature as a means for creating models of appropriate fidelity and computational demands [1, 19]. Typically, model reduction applies mathematical operators to system-level models. These reduction operations include proper orthogonal decomposition (POD), centroidal Voronoi tessellation (CVT), and energy-based reductions.

Reducing a model Φ into another model Φ' entails accurately approximating a function for Φ by another function for Φ' that is more efficient according to some metric, e.g., that requires much less data to define. The reduced function may be explicitly defined, e.g., by a formula or pre-computed function, such as a simpler PDE or a coarse-grid approximation, an interpolant or a least-squares approximant. In other words, reduced-order modeling can be viewed as the use of data compression techniques. This article differs from engineering model-reduction in that it creates a reduced-order diagnosis model, but in an entirely different language (propositional logic) than that of the original HVAC model, which is expressed as a hybrid automaton [14].

3. Application Domain

3.1 System Schematic

We model an air-handling unit (AHU) as a system that supplies air to a single zone in a building (although in general, it can supply air to multiple, separately controlled zones), as shown in Figure 1. The AHU consists of a set of components which includes: a bypass mixer, a heating coil followed by a cooling coil, all housed in a single air duct. The bypass dampers are controlled to provide a mixture of outside air and return (re-circulated) air; we constrain the amount of outside air to vary between the lower and upper proportions μ_{min} and μ_{max} of the maximum airflow through the AHU. The air supplied to the zone(s), at temperature T_s, is processed through a variable air volume (VAV) box containing a damper and heating/cooling coils. We constrain the airflow to each zone (as regulated by a damper) to vary between the lower and upper proportions of the specified maximum flow.

3.2 AHU Control Strategy

The AHU controller aims to maintain the supply air temperature T_s at its set point T^* by controlling the heating coil, dampers, and cooling coil in sequence. We assume that we have a simplified system, in which we focus only

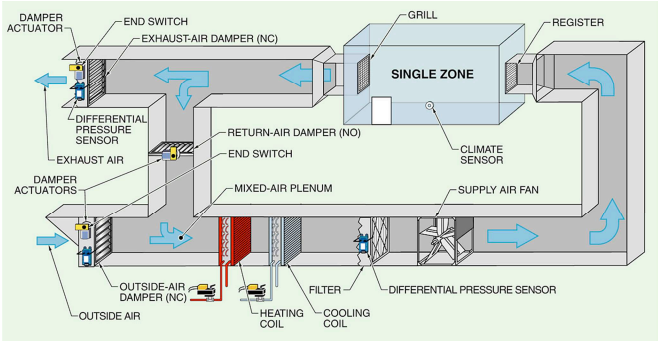


Figure 1. Schematic for Air Handling Unit with a single zone

on the supply air temperature T_s. In reality, we also need to control the zone parameters, e.g., temperature T_z, humidity W_z and CO_2 level C_z.

The AHU controller is split into controllers for air-flow, and temperature (heating/cooling).

1. Air-flow control: we must control the supply and return fans, in conjunction with the dampers, to ensure that the flow of air to the zone and through the AHU meets the relevant targets. A PID controller governs the mixing of return and outside air through setting the dampers for (a) exhaust air, (b) outside air and (c) recirculated air.
2. Air temperature control: Given air that has been already mixed (with return air and outside air), the temperature of this air must be modulated to achieve the supply air set-point s for the zone. We use separate PID controllers for the chiller and heater coils to achieve the zone's temperature set-point T_z.

The AHU can operate in multiple modes, the most important of which are as follows:

- Heating: the heating mode operates if the heating load exceeds the capacity of the return and outdoor air to achieve the required supply air temperature T_s.
- Cooling: the cooling mode operates if the cooling load exceeds the capacity of the return and outdoor air to achieve the required supply air temperature T_s.
- Economizing: the economizer mode operates if there is a cooling load and the outdoor air temperatures are low enough to satisfy this load. Extra outside air is brought in instead of running the cooling coil to cool the mix of return air and minimum outdoor air.

The AHU control system is a switching controller, and must be able to switch between its modes dependent on the set-points for temperature (T^*), humidity (W^*) and CO_2 (C^*), and on the current zone loads and outdoor air conditions. Within any mode, we need dynamical models to adequately capture the thermodynamics of flow of humidified air throughout the system.

4. System Architecture

This section describes the architecture for transforming the reference model into embedded code.

We define a *reference model*, Φ_R, as the basis from which we generate all other models and embedded code. We transform Φ_R into application-specific representations, e.g., simulation and diagnosis models, as shown in Figure 2. This figure shows how we map the reference model, Φ_R, into a model suitable for embedded-code generation; in our case we examine control and diagnosis models, Φ_C and Φ_D, respectively. Given one of these models, we then need to "compile" the model into an embeddable representation, based on the requirements of the target platform. For example, if the embedded platform is a SunSpot with particular CPU and memory restrictions, we can generate Java control code ξ_C that abides by the restrictions. In this article we focus on the transformation from reference model Φ_R to diagnosis model Φ_D.

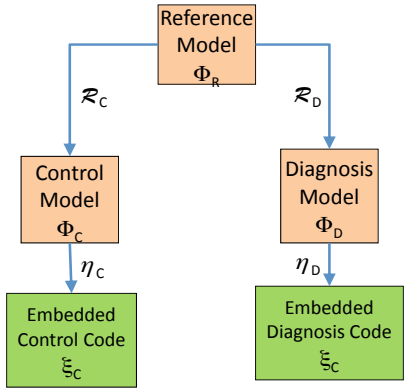


Figure 2. Mapping Reference Model to Embedded Code

Figure 3 shows a more detailed snapshot of the model transformation process. [3] Figure 3 shows that we must use

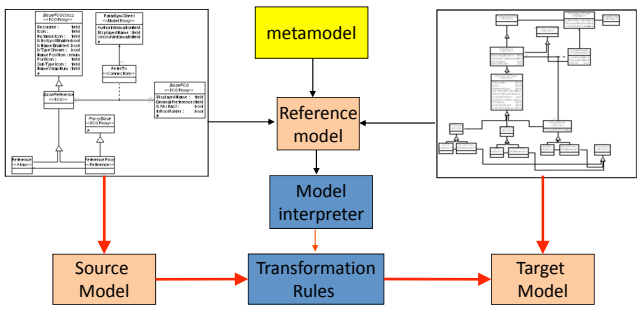


Figure 3. Architecture for Model and Embedded Code Generation

metamodels for the source and target models in order to apply transformation rules:

DEFINITION 1 (Model Transformation System). *A Model Transformation System consists of the tuple* $\langle \mathcal{M}_S, \mathcal{M}_C, \mathcal{R} \rangle$, *where* $\mathcal{M}_S$ *and* $\mathcal{M}_T$ *are the source and target metamodels, respectively, and* $\mathcal{R}$ *is the collection of transformation rules applied to* $\mathcal{M}_S$ *and* $\mathcal{M}_T$.

[3] More details of model transformation for HVAC systems can be found in [2].

There are many different theories for model transformation, as well as corresponding tools (see [17]); here we focus on describing the rules used for the model transformation, as applied to our HVAC domain for diagnostics.

In this article, the *reference model* Φ_R is defined using the Modelica language to specify hybrid automata [14]. The metamodel for this language is summarized in Figure 4. The *diagnosis model* Φ_D is defined using the Lydia

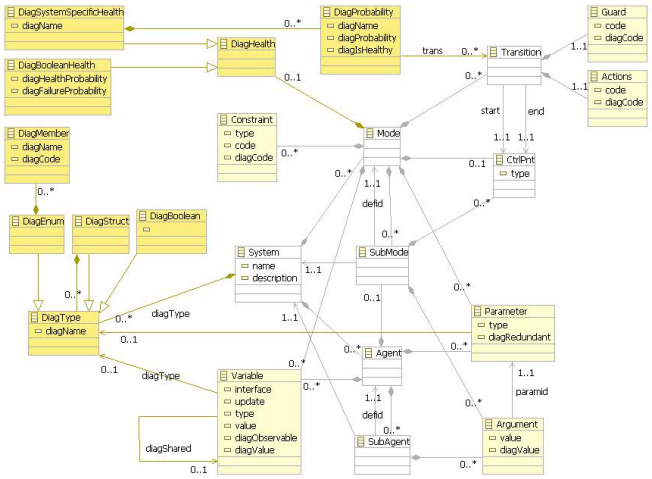


Figure 4. MetaModel for Reference Model

language [9]. The metamodel for this language is summarized in Figure 5.

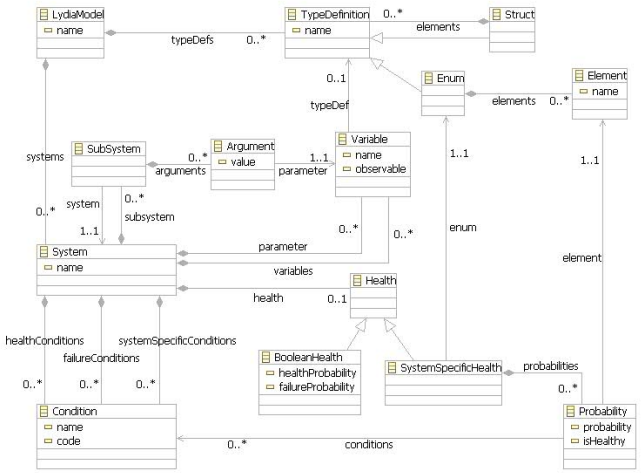


Figure 5. MetaModel for Diagnostics Model

5. Reference Model

This section outlines our notation for the reference model.

5.1 Component-Based Systems

In contrast to modeling and model-reduction approaches that work on monolithic models, we propose an object-oriented approach where we assume that a system can be represented as an inter-connected set of components [24]. Hence, we can define a component-based reference system as follows:

DEFINITION 2 (Component-Based Reference System). *A Reference System consists of the pair $\langle \mathcal{G}, \mathcal{C} \rangle$, where $\mathcal{G}$ is the system topology multi-graph in which each component is a node and each component connection is an edge-set, and $\mathcal{C}$ is the collection of components from which the system model can be composed.*

The multi-graph notation enables us to define any two components as having multiple connections between them. A multi-graph G is an ordered pair $G := (V, E)$ in which V is a set of vertices or nodes, and E is a multi-set of unordered pairs of vertices, called edges or connections.

A key aspect of a reference system is the notion of component [7]:

DEFINITION 3 (Component). *A component consists of the tuple $\langle \mathcal{I}, \mathcal{O}, \zeta \rangle$, where $\mathcal{I}$ and $\mathcal{O}$ are sets of input and output ports (variables), respectively, and ζ is a relation that specifies the transformation of input port values to output port values.*

5.2 Relations for Reference Model

We model mode changes as discrete, i.e., they activate and deactivate constituent equations of model components. Given a discrete set of system modes, the system can operate in a single mode at any time. For example, a hydraulic system with a valve stuck open will operate with equations in which the valve does not respond to actuator commands to close the valve. We can also model different sets of equations for the valve being opened and shut: when the valve is shut, it enforces zero pressure drop across it; when it is open, it enforces zero flow across it. Using this framework, we define a hybrid system [14] as one that operates in piecewise continuous modes, each of which can be modeled by ODEs and DAEs, or by PDEs (depending on the class of Hybrid System). Our simulation model is thus defined as a Hybrid System.

We explicitly model in the hybrid system a set of modes for normal and for failure states. We also assume that only a subset of events are observable, and in our models we denote these events as those relating to sensors and actuators changing state. We assume fault events are unobservable.

DEFINITION 4 (Hybrid System). *A hybrid system is defined as $\Phi_S = (\mathcal{H}, X, \Sigma, \mathcal{H}_0, E, f, G)$ where*

- *$\mathcal{H}$ is the set of discrete states (or modes) of the system,*
- *$X \subseteq \Re^n$ is the continuous state space,*
- *Σ is a finite set of transition labels or events,*
- *$\mathcal{H}_0 \subseteq \mathcal{H} \times X$ is the set of initial conditions,*
- *$E \subset \mathcal{H} \times \Sigma \times \mathcal{H}$ is the transition relation, which defines the set of (controlled and autonomous) discrete transitions,*
- *$f : \Re \times \mathcal{H} \times X$ is the flow condition for every mode defined by a differential equation,*
- *and $G : E \rightarrow 2^X \times \pi$ is a partial function that associates a guard condition (represented as a subset of X) with each autonomous transition, given a probability π.*

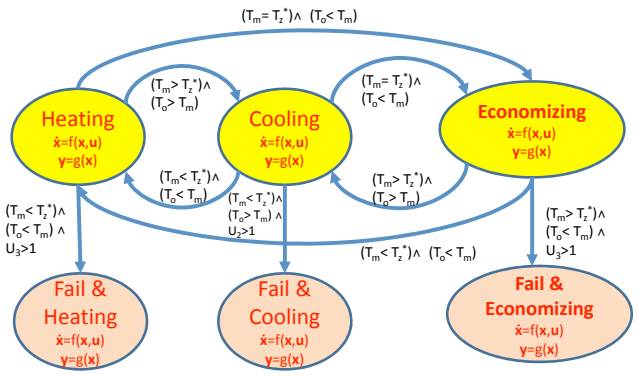


Figure 6. Hybrid Systems model for AHU

We define a set X of continuous-valued variables, and a set $\mathcal{H}$ of discrete-valued (mode) variables, and denote the system variables as $\mathcal{Z} = \boldsymbol{X} \cup \mathcal{H}$. Each variable $z_i \in \mathcal{Z}$ has a corresponding domain $\theta_i \in \Theta$.

Example: We model the hybrid system for an AHU as shown in Figure 6, with the modes, events and mode transitions specified. We see that we can transition between the nominal modes, but that the fault modes are absorbing, since we assume no repair. The fault modes are linked to nominal modes in terms of the effect of particular failed components on that mode; e.g., a fouled heating coil will affect the *heating* mode and not the other nominal modes.

We can define the generic state equations for control as follows:

DEFINITION 5 (State-space Equations $\mathcal{E}_S$). *We represent a simulation model as having a set $\mathcal{E}$ of dynamical equations over a set $\mathcal{Z}$ of variables, where $\mathcal{E}$ is given by:*

$$\dot{\boldsymbol{X}}(t) = f[\boldsymbol{X}(t), \boldsymbol{U}(t), \hbar(t)] \quad (1)$$

$$\boldsymbol{Y}(t) = g[\boldsymbol{X}(t), \boldsymbol{U}(t), \hbar(t)]. \quad (2)$$

For simplicity of exposition, we will suppress the temporal notation, e.g., state x rather than $x(t)$.

Example: In the literature, an AHU simulation model consists of a set of equations defining normal-mode behaviours, with no specification of modes [26]. Here, we explicitly define the system modes and their entailed equations. In our AHU model, we assume a control vector $\boldsymbol{U} = (u_1, u_2, u_3)$, where u_1 is the return-air mass flow rate, u_2 is the chiller pump flow rate, and u_3 is the heating coil rate. We also define our three primary continuous-valued state variables as $\boldsymbol{X} = (x_1, x_2, x_3)$, where x_1 is the zone temperature T_z, x_2 is the zone humidity W_z, and x_3 is the supply temperature T_s.

5.3 System Modes

A system may be defined as operating in a particular mode, which has a characteristic set of equations. We define two types of system modes: operating and fault. An operating mode $\hbar_o \in \mathcal{H}_O$ is defined by a control setting $\tilde{\boldsymbol{U}}$ and observed state $\tilde{\boldsymbol{Y}}$, and presumes that all components are operating normally. A fault mode $\hbar_f \in \mathcal{H}_f$ specifies the health of the system components, i.e., whether each component is functioning in a nominal or degraded/faulty state. Each failure mode can take on the value of *OK* or one of multiple failure states.

For the system-level mode specification $\mathcal{H}$, we will need to define $\mathcal{H}$ as the cross-product of the operating mode $\mathcal{M}_O \in \mathcal{H}_O$ and the failure mode $\mathcal{M}_F \in \mathcal{H}_F$, i.e., $\mathcal{H}$= $\mathcal{H}_O \times \mathcal{H}_F$. This is because the dynamics of a failure depends on the particular operating mode in which that failure occurs. For example, an aircraft during takeoff or during high-altitude cruising faces very different airflow dynamics, and a flap fault will affect the plane differently in the two situations.

Each mode is characterized by a set of (a) set-points and (b) ambient conditions. In our AHU model, all modes share two set-points: zone temperature $T_z^*(e.g., 18^oC)$; and zone humidity $W_z^*(e.g., 0.6)$. The control settings for operating modes heating, cooling and economizing are denoted by $(\bar{u}_1, 0, \bar{u}_3)$, $(\bar{u}_1, \bar{u}_2, 0)$, and $(\bar{u}_1, 0, 0)$, respectively, where $\bar{u}_i$ denotes $u_i > 0$.

In typical circumstances, the ambient conditions force the use of a particular control setting that characterises an operating mode. For example, when the desired supply air temperature T_S is higher than both the outside air temperature T_O and the return air temperature T_r, then we must be in heating mode. If we assume that $\dot{m}_s$ is fixed and constant, then the modes and ambient conditions drive many other system set-points, e.g., damper set-points. For example, we can compute the control settings for the dampers based on the temperatures in the AHU: the damper setting to introduce outside air is set such that the proportion of outside air, μ_A, is given by $\mu_A = \frac{T_m - T_r}{T_o - T_r}$; similarly, the dampers are set such that the proportion of CO_2, μ_C, is given by $\mu_C = \frac{C_R - C_S}{C_R - C_O}$.

5.4 Reduced-Order HVAC Reference Model

This section describes how we generate a physics-based reduced-order model given the fundamental equations for thermodynamics. The basic principle that we adopt is the First Law of Thermodynamics, which defines conservation of energy (and indirectly mass). The heat balance equation is $\Delta E = H - W$, where the change in internal heat ΔE is the difference between the heat H input and the energy W extracted from the system.

In an HVAC system, we use either water or air to transfer energy to (heating) or extract energy from (cooling) a zone. A standard equation specifying heat transfer through a substance is Fourier's equation, which is a 3D PDE with time.

$$\rho C_p \frac{dT}{dt} = k \left[\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \right], \quad (3)$$

where k is thermal conductivity.

We simplify this to an atemporal, 1D representation, as follows.

$$\rho C_p \frac{dT}{dt} = \dot{Q} = kA \frac{\Delta T}{\Delta x} = \frac{kA}{\ell}(T_1 - T_2), \quad (4)$$

where A is cross-section (area), ℓ is material thickness, and T_1/T_2 are the high/low temperatures.

We can apply this simplification to heat and mass transfer in HVAC flow systems as follows. Given supply and

return air temperatures T_s and T_r, respectively, heat transfer into a zone is given by $\dot{Q} = \rho\dot{V}C_p(T_s - T_r)$ for heating and $\dot{Q} = \rho\dot{V}C_p(T_r - T_s)$ for cooling, where $\dot{V}$ is the volumetric flow rate.

We can generalise this approach to a situation where we assume energy and mass conservation across the interchange of multiple fluid streams. In this case, we assume steady and adiabatic conditions, and that no frictional losses occur across the mixing regions. We adopt a standard approach, the Hardy–Cross method [10], for evaluating the air flow rate and the pressure drops across the components of a duct network. We model the VAV air distribution subsystem's duct-work as a serial-parallel hydraulic network, in which each component is defined as a flow resistance. We assume that the air distribution duct-work is hydraulically balanced.

For example, in an AHU where we mix air masses from the return $\hbar_r$ and outside, $\hbar_o$, to create a mixed mass $\hbar_m$, we have energy and mass balance equations as follows:

$$\dot{m}_r C_{pa} T_r + \dot{m}_o C_{pa} T_o = \dot{m}_m C_{pa} T_m \tag{5}$$

$$\dot{m}_r + \dot{m}_o = \dot{m}_m. \tag{6}$$

We can compute the mixing temperature T_m and mixing humidity W_m from the input streams as follows:

$$T_m = \frac{\dot{m}_r T_r + \dot{m}_o T_o}{\dot{m}_r + \dot{m}_o} \tag{7}$$

$$W_m = \frac{\dot{m}_r W_r + \dot{m}_o W_o}{\dot{m}_r + \dot{m}_o} \tag{8}$$

From equation 7, we can derive

$$T_m = T_r \frac{\dot{m}_r}{\dot{m}_r + \dot{m}_o} + T_o \frac{m_o}{\dot{m}_r + \dot{m}_o} = \mu T_r + (1 - \mu) T_o, \tag{9}$$

where $\mu = \frac{\dot{m}_r}{\dot{m}_r + \dot{m}_o}$ is the return air fraction of the total air mass flow-rate. W_m can be deduced as

$$W_m = \mu W_r + (1 - \mu) W_o. \tag{10}$$

We can also apply energy balance across the portion of the AHU from the mixing box to the supply airflow into the zone. If we denote the heat of the airflow in the mixing box and supply as $\dot{Q}_m$ and $\dot{Q}_s$ respectively, as the supply fan and heating coil inject heat denoted by H_{sf} and H_{hc} respectively, then by energy balance we must have

$$\dot{Q}_s = \dot{Q}_m + H_{sf} + H_{hc}.$$

If we have a constant flow-rate $\dot{m}_s$ in this portion of the AHU, and there is a temperature increase of ΔT_{sf} and ΔT_{hc} at the supply fan and heating coil, respectively, then we must have

$$\dot{m}_s C_p T_s = \dot{m}_m C_p T_m + \dot{m}_s C_p \Delta T_{sf} + \dot{m}_s C_p \Delta T_{hc},$$

from which we can deduce, if $\dot{m}_s = \dot{m}_m$,

$$T_s = T_m + \Delta T_{sf} + \Delta T_{hc} \tag{11}$$

In an analogous fashion, we can define the energy balance when we have a temperature increase of ΔT_{cc} due to a cooling coil, to give

$$T_s = T_m + \Delta T_{sf} - \Delta T_{cc}. \tag{12}$$

6. Diagnosis Model-Generation Process

We transform a reference model Φ_S to a diagnosis model Φ_D using model transformation methods [17], which map the (source) metamodel for Φ_S into the (target) metamodel for Φ_D using a set of transformation rules. Here, we focus on the HVAC diagnostics application domain, and provide domain-specific examples of the transformation rather than a formal analysis. We have addressed other applications of this approach, e.g., *control code* generation for building lighting systems, elsewhere [16].

6.1 Diagnosis Model

The purpose of a diagnostics model is to be able to distinguish the mode of the system given an observation. Hence the level of model fidelity is only that which is sufficient to perform such mode-identification. As a consequence, a diagnostics model is typically more abstract than the corresponding reference model.

We adopt a diagnosis model analogous to the reference model, except that we have a restricted form for the variable domains and equations. We assume that each domain is discrete, and that each equation is propositional.

We represent a diagnosis model for an artifact as a propositional formula over a set $\mathcal{Z}$ of discrete-valued variables [20].

DEFINITION 6 (Diagnostic Model). *A diagnostic model Φ_D is defined as the triple $\Phi_D = \langle \mathcal{E}_D, \mathcal{H}_f, \boldsymbol{y}_D \rangle$, where $\mathcal{E}_D$ is a propositional theory over a set of variables $\mathcal{Z}_D$, $\mathcal{H}_f \subseteq \mathcal{Z}$, $\boldsymbol{y}_D \subseteq \mathcal{Z}_D$, $\mathcal{H}_f$ is the set of component failure-mode variables, and $\boldsymbol{y}_D$ is the set of observable variables.*

6.2 Diagnosis Model Generation

We model HVAC systems using both the nominal and failure modes of the system components. HVAC systems are stiff, i.e., they have a mixture and fast and slow processes. We can simplify such stiff systems by abstracting the fast, continuous transients into discontinuous changes between slow states; this process converts the models into mixed continuous/discrete, hybrid, models. We also model the transition from a nominal to a failure state as a discontinuous change.

In this work, we assume that the system is in steady-state conditions, i.e., it is not switching between operating modes. Hence we only model and map steady-state nominal conditions and transitions from an operating mode to a fault mode. This steady-state analysis is the approach considered in most approaches, e.g., APAR [22].

To distinguish reference and diagnosis models, we use superscript R for reference, and superscript D for diagnosis. We apply model transformation, which uses the metamodel Υ_R for Φ_R and the metamodel Υ_D for Φ_D, together with a set $\mathcal{R}$ of transformation rules, to transform Φ_R into

Φ_D, i.e., $\Upsilon_R \xrightarrow{\mathcal{R}} \Upsilon_D$. The rules $\mathcal{R}$ transform the model entities for Φ_R and Φ_D as follows:

- map variables: $z_i^R \rightarrow z_i^D$
- map domains: $\theta_i^R \rightarrow \theta_i^D$
- map equations: $\mathcal{E}_i^R \rightarrow \mathcal{E}_i^D$
- map guard conditions G into diagnosis equation conditions.

Variable Mapping: We assume a 1:1 mapping for variables, i.e., we do not abstract away any variables in Φ_R.
Domain Mapping: For many variables, we assume that we know the nominal operating value, given a particular mode. For some variables, such as the damper position, the best that we can know is the most-likely range of its value. If we know the nominal value (or set-point x), as well as the minimum and maximum values x_{min}, x_{max} respectively, then we can perform a precise mapping from a continuous-valued variable $X_R \in \mathcal{Z}_R$ to its counterpart $X_D \in \mathcal{Z}_D$ in a diagnosis model. In other words, if X_R has domain $[x_{min}, x_{max}]$ and nominal set-point x^*, we can map it to a discrete-valued variable X_D with domain $\{\xi_{min}, \xi^-, \xi, \xi^+, \xi_{max}\}$, where $\xi^- \in (x_{min}, x)$ and $\xi^+ \in (x, x_{max})$.[4] This is shown in Figure 7.

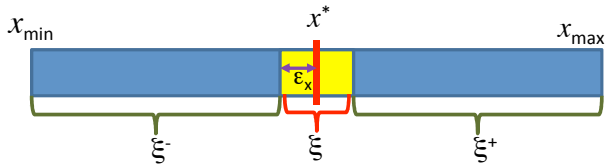


Figure 7. Discrete quantization of continuous-valued variable domain. We subdivide the domain of x into three ranges, corresponding to nomimal (ξ), sub-nominal (ξ^-), and super-nominal (ξ^+).

Equation Mapping: We map equations based on the equation type. Section 6.3 provides a subset of equation mappings, in order to exemplify the approach. We address three main equation types:

Temporal Arithmetic Relations Given relation $X_1(t) = X_2(t+\tau)$, we map X_1 and X_2 to propositions denoting different times, i.e., X_1^t and $X_2^{t+\tau}$; we then define a propositional formula based on the temporal precedence, i.e., $X_1^t \Rightarrow X_2^{t+\tau}$.

Atemporal Arithmetic Relations Given $X_i = \alpha f(X_j, U)$, we map this to a proposition of the form $[X_i = \xi] \Rightarrow [\alpha f(X_j, U) = \xi]$ for the relevant discrete domain values of X_i, X_j.

Differential-equation Relations For a relation of the form $\dot{X} = \alpha f(X, U)$, we enforce our steady-state assumption to set $\dot{X} = 0$, from which we obtain $\alpha f(X, U) = 0$.

[4] In practice, we will need to calibrate this discretization, using machine learning to identify the nominal operating range.

Guard Conditions: If the guard condition G is satisfied and we execute a transition to a diagnosis mode $\hbar$, then the resultant equations in the diagnosis model will be of the form $(G \wedge \hbar) \wedge W$, where W is a proposition formed from the equations in Φ_R for mode $\hbar$.

6.3 Component-Specific Mapping

We adopt a component-based modeling approach, in which we define a system-level model from an interconnected set of components. We assume that by automatically generating diagnosis components from reference components, we can then define a system-level diagnosis model by composing these diagnosis components.

This section describes a subset of the mappings defined for the AHU system components. We classify our model reduction process into two types:

1. Sensor/Actuator device reduction
2. Thermodynamics-Based reduction

6.3.1 Fault Modes Covered

We adopt a component-based modelling approach, so we list the fault modes on a component basis, as shown in Table 1. We adopt most of the same component faults as done in the APAR model [22], with some modifications.

Component	Failure-Mode
sensor	no-signal drift-high drift-low
damper	stuck-open stuck-closed leakage
duct/pipe	clogged leakage
valve	stuck-open stuck-closed leakage
fan	fail
coil	fouled

Table 1. Failure Modes for AHU

6.3.2 Sensor/Actuator Device Transformation

This class of reduction focuses on generating a tractable diagnostics-oriented representation of the physical sensor/actuator devices. The primary principle is that the real-world setting equals the device setting when the device is functioning normally; for example, a real temperature equals the sensor-measured temperature.[5]

Sensor Mapping: We can represent an ideal, fault-free sensor using the reference equation

$$X_s(t+\tau) = X_m(t),$$

where the subscript s denotes the sensed value and m denotes the measured (true) value, t is a time index, and τ is

[5] Here, we focus on device functionality, and we will deal with calibration and internal parameter estimation issues in future work.

a time offset. This equation notes that the sensed value for variable X, when the sensor is nominal, should be identical to the true value, given a temporal offset due to the sensing process.

The device mode values for $\hbar_S$ we consider are: {OK, no-data, fail-low}. In the case of a sensor with actual and measured values S and $\hat{S}$, respectively, we perform fault isolation when a residual has magnitude δ_S, which is a device-dependent offset.

For sensors, we apply the rules defined earlier, assuming that $X_m(t)$ has a nominal value, x, and an offset ϵ_x around x that represents the sensor error or allowable variability.

From a diagnostic perspective, we are only interested in knowing is we are at the nominal value or not. Hence we map X_m's domain to a discrete-valued domain as follows: $[x_{min}, x-\epsilon_x] \rightarrow \xi^-$; $[x-\epsilon_x, x+\epsilon_x] \rightarrow \xi$; $[x+\epsilon_x, x_{max}] \rightarrow \xi^-$. By including the *nil* value to record the case when the sensor produces no output, we obtain the domain {*nil*,ξ^-, ξ, ξ^+}.

We map the equations as follows. We denote the failure-modes, the reference model equations (given $\delta > \epsilon_x$), and the diagnosis model equations in Table 2.

Actuator Mapping: We can represent an ideal, fault-free actuator using the reference equation

$$X_m(t+\tau) = X_c(t),$$

where the subscript m denotes the measured (actual) value and c denotes the commanded (desired) value, t is a time index, and τ is a time offset. This equation notes that the physically-assigned value for variable X, when the actuator is nominal, should be identical to the commanded value, given a temporal offset due to the actuation process. Analogous to a sensor, we assume that an actuator has domain $[x_{min}, x_{max}]$. For example, an actuator for a valve has values that can range from fully-open to fully-closed. The device mode values for $\hbar_A$ we consider are: {OK, no-data, stuck-open, stuck-closed}.

We map the equations as follows. We denote the failure-modes, the reference model equations (given $\delta > \epsilon_x$), and the diagnosis model equations in Table 2.

6.3.3 Thermodynamics-Based Transformation

This class of reduction focuses on generating a tractable diagnostics-oriented representation of the fundamental thermodynamic heat and mass conservation equations.

Pipe/Duct: Given a pipe/duct, the rate of change of temperature given inlet and outlet temperatures of T_i and T_o is [4]:

$$\frac{dT_o}{dt} = \frac{(h_i + h_o)m_a C_p}{h_i M_c C_c}(T_i - T_o).$$

We abstract away the temporal elements of this equation, and define a lumped-parameter relation for the outlet temperature as $T_o = T_i(1-\gamma(h, C, m))$, where $\gamma(h, m, C)$ is a function parameterised by the relevant coefficients for heat transfer h, specific heat C, and the mass m. For simplicity of exposition, we will drop the parameters for γ.

We introduce the following failure modes for the pipe/duct: $\hbar_d$ = {OK, clogged, leakage}, where clogged denotes no throughput flow, and leakage denotes some loss of fluid flow. Given these mode values and the lumped-parameter relation, we can generate diagnostic equations for temperature changes along a pipe/duct, as in Table 2, where $\lambda \in (0, 1)$ is a leakage parameter. We can define an analogous set of equations for mass flowrate m.

Variable-orifice valve (or damper): We now define our model for a simple variable-orifice valve (or damper for air-flow). For a valve with inlet/outlet flowrates of $\dot{m}_i, \dot{m}_o$ and position $\pi \in (\pi_{min}, \pi_{max})$, we have $\dot{m}_o = f(\dot{m}_i, \pi)$. For simplicity, we assume that $\dot{m}_o = \dot{m}_i$ when $\pi = \pi_{max}$, and $\dot{m}_o = 0$ when $\pi = \pi_{min}$. Together, this information is sufficient to develop a set of diagnostic equations given the failure modes $\hbar_v$ = { OK, clogged, leakage}, in Table 2, where $\lambda \in (0, 1)$ is a leakage parameter.

Fan/Pump: We model a fan/pump as a device that increases the outflow pressure based on a pump constant η: $P_o = f(P_i, \eta)$. Using this simplified equation, together with given the failure modes $\hbar_f$ = { OK, fail, VSD-fail}, where fail denotes total failure and VSD-fail denotes anomalous variable-speed drive (VSD), we can define diagnostic equations analogous to those for the damper.

6.3.4 Flow Composition/Decomposition

The mixing box component explicitly computes mixed flow-rates, temperatures and humidity based on incoming air-flows. Equations 6, 9 and 10 specify the flow-rate, temperature and humidity relationships, respectively. We can map these relations directly to diagnosis-model relations by introducing failure-modes defining causes for inconsistencies in either the mix-ratio μ, i.e., blocked ducts or other upstream blockages (e.g., damper faults), or in the temperatures of incoming flows, T_R and T_o.

7. Deployment and Validation

The large buildings for which we are targeting our code-generation system typically use Building Management Systems to deploy the control and diagnostics code [21]. As such, we assume that we have a PC-style processor on which control and diagnostics are computed in a centralized manner. We are also investigating the generation of distributed control code that can be deployed on a variety of wireless networks. For example, we have shown how to automate the generation of control code for SunSpot nodes [16].

We are in the process of validating our approach, both in terms of computational feasibility and correctness. We have performed a preliminary comparison of the diagnostics results with those of a well-known commercially-deployed tool, the APAR rule-based system for HVAC [23]. Our comparison indicates that the auto-generated rules isolate faults with accuracy similar to that of APAR, using simulated building data [18]. However, validation on real-world data is necessary to constitute a proper validation.

8. Conclusions

This article has described a model-driven approach for generating embedded diagnosis code using the equation-based

	Mode	Reference Model Equation	Diagnosis Model Equation
Sensed variable X	OK	$X_s(t) = X_m(t+\tau)$	$(\hbar = OK) \Leftrightarrow [(X_m^t = \xi) \Rightarrow (X_s^{t+} = \xi)]$
	drift-high	$X_s(t) = X_m(t+\tau) + \delta$	$(\hbar = high) \Leftrightarrow [(X_s^t = \xi) \Rightarrow (X_s^{t+} = \xi^+)]$
	drift-low	$X_s(t) = X_m(t+\tau) - \delta$	$(\hbar = low) \Leftrightarrow [(X_m^t = \xi) \Rightarrow (X_s^{t+} = \xi^-)]$
	no-signal	$X_s(t) =$*nil*	$(\hbar = no - signal) \Leftrightarrow (X_s^{t+} =$ *nil*$)$
Actuator U	OK	$U_m(t) = U_c(t+\tau)$	$(\hbar = OK) \Leftrightarrow [(U_c^t = \xi) \Rightarrow (U_m^{t+} = \xi)]$
	stuck-open	$U_m(t) = U_c(t+\tau) + \delta$	$(\hbar = open) \Leftrightarrow [(U_c^t = \xi) \Rightarrow (U_m^{t+} = \xi^+)]$
	stuck-closed	$U_m(t) = U_c(t+\tau) - \delta$	$(\hbar = closed) \Leftrightarrow [(U_c^t = \xi) \Rightarrow (U_m^{t+} = \xi^-)]$
	no-data	$U_m(t) =$*nil*	$(\hbar = no - data) \Leftrightarrow (U_m^{t+} =$ *nil*$)$
Pipe/Duct	OK	$T_o(t) = (1-\gamma)T_i(t)$	$(\hbar_d = OK) \Leftrightarrow [(T_i^t = \xi) \Rightarrow (T_o^t = \xi)]$
	leakage	$T_o(t) = (1-\gamma)(1-\lambda)T_i(t)$	$(\hbar_d = leakage) \Leftrightarrow [(T_i^t = \xi) \Rightarrow (T_o^t = \xi^-)]$
	OK	$\dot{m}_o(t) = (1-\gamma)\dot{m}_i(t)$	$(\hbar_d = OK) \Leftrightarrow [(\dot{m}_i^t = \xi) \Rightarrow (\dot{m}_o^t = \xi)]$
	clogged	$\dot{m}_o(t) = 0$	$(\hbar_d = clogged) \Leftrightarrow (\dot{m}_o^t = 0)$
	leakage	$\dot{m}_o(t) = (1-\gamma)(1-\lambda)\dot{m}_i(t)$	$(\hbar_d = leakage) \Leftrightarrow [(\dot{m}_i^t = \xi) \Rightarrow (\dot{m}_o^t = \xi^-)]$
Damper	OK	$\dot{m}_o(t) = f(\dot{m}_i(t), \pi(t))$	$(\hbar_v = OK) \Leftrightarrow [(\dot{m}_i^t = \xi) \Rightarrow (\dot{m}_o^t = \xi)]$
	clogged	$\dot{m}_o(t) = 0$	$(\hbar_v = clogged) \Leftrightarrow [(\dot{m}_o^t = 0)$
	leakage	$\dot{m}_o(t) = f(\dot{m}_i(t), \pi(t))(1-\lambda)$	$(\hbar_v = leakage) \Leftrightarrow [(\dot{m}_i^t = \xi) \Rightarrow ((\dot{m}_o^t = \xi^-)]$

Table 2. Equation Mappings for selected AHU Components

object language Modelica as a source representation. We applied our approach to the generation of qualitative diagnostics code for HVAC applications, showing the transformation rules for this domain.

We have done a preliminary validation of the feasibility and correctness of our approach, showing that the auto-generated diagnostics compare favorably with those of a well-known commercial diagnostics system. We are extending this work to incorporate control-code generation and deployment on a distributed wireless network. This HVAC domain will complement the building lighting system control system previously reported [16]. We also plan to apply this approach to real buildings.

Acknowledgments

This work was funded by Science Foundation Ireland (SFI) grant 06-SRC-I1091.

References

[1] AC Antoulas, DC Sorensen, and S. Gugercin. A survey of model reduction methods for large-scale systems. In *Structured matrices in mathematics, computer science, and engineering*, volume 280, page 193. Amer. Mathematical Society, 2001.

[2] M. Behrens and G. Provan. Temporal model-based diagnostics generation for hvac control systems. *Proc. Itnl. Conf. on Theory and Practice of Model Transformations (ICMT)*, pages 31–44, 2010.

[3] M. Behrens, G. Provan, M. Boubekeur, and A. Mady. Model-driven diagnostics generation for industrial automation. In *7th IEEE International Conference on Industrial Informatics (INDIN)*, pages 708–714. IEEE, 2009.

[4] DR Clark, CW Hurley, and CR Hill. Dynamic models for HVAC system components. *ASHRAE transactions*, 91(1):737–751, 1985.

[5] D.B. Crawley, J.W. Hand, M. Kummert, and B.T. Griffith. Contrasting the capabilities of building energy performance simulation programs. *Building and Environment*, 43(4):661–673, 2008.

[6] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[7] B. Denckla and P.J. Mosterman. Formalizing causal block diagrams for modeling a class of hybrid dynamic systems. In *Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC'05. 44th IEEE Conference on*, pages 4193–4198. IEEE, 2005.

[8] H. Elmqvist, M. Otter, D. Henriksson, B. Thiele, and S.E. Mattsson. Modelica for embedded systems. In *Proceedings of the 7th International Modelica Conference*, pages 354–363. Linköping University Electronic Press, 2009.

[9] A. Feldman, J. Pietersma, and A. van Gemund. All roads lead to fault diagnosis: Model-based reasoning with lydia. *Proc. BNAICŠ06*, 2006.

[10] J.A. Fox. *An introduction to engineering fluid mechanics*. McGraw-Hill, 1974.

[11] P. Fritzson, A. Pop, and M. Sjölund. Towards modelica 4 meta-programming and language modeling with meta-modelica 2.0. Technical Report Computer and Information Science, ISSN 1654-7233, 2011.

[12] P.A. Fritzson. *Principles of object-oriented modeling and simulation with Modelica 2.1*. Wiley-IEEE Press, 2004.

[13] Z. Hemel, L. Kats, and E. Visser. Code generation by model transformation. *Theory and Practice of Model Transformations*, pages 183–198, 2008.

[14] T.A. Henzinger. The theory of hybrid automata. In *Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 278–292. IEEE, 1996.

[15] J. Larsson and P. Fritzson. A modelica-based format for flexible code generation and causal model transformation. In *5th International Modelica Conference, 2006*. Vienna: Arsenal, 2006.

[16] A. Mady, M. Boubekeur, and G. Provan. Compositional model-driven design of embedded code for energy-efficient buildings. In *7th IEEE International Conference on Industrial Informatics (INDIN)*, pages 250–255. IEEE, 2009.

[17] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[18] G. Provan. Generating Reduced-Order Diagnosis Models for HVAC Systems. In *Itnl. Workshop on Principles of Diagnosis*, Murnau, Germany, October 2011.

[19] T. Reis and T. Stykel. A survey on model reduction of coupled systems. *Model Order Reduction: Theory, Research Aspects and Applications*, pages 133–155, 2008.

[20] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[21] T. Salsbury. A survey of control technologies in the building automation industry. In *Proc. of the 16th IFAC World Congress*, pages 331–341, 2005.

[22] J. Schein and S.T. Bushby. A hierarchical rule-based fault detection and diagnostic method for HVAC systems. *HVAC&R Research*, 12(1):111–125, 2006.

[23] J. Schein, S.T. Bushby, N.S. Castro, and J.M. House. A rule-based fault detection method for air handling units. *Energy and buildings*, 38(12):1485–1492, 2006.

[24] J.L. Stein and L.S. Louca. A component-based modeling approach for system design: Theory and implementation. *Ann Arbor*, 1001:48109–2125.

[25] C. Turner, M. Frankel, and U.S.G.B. Council. *Energy performance of LEED for new construction buildings*. New Buildings Institute, 2008.

[26] M. Wetter. Modelica Library for Building Heating, Ventilation and Air-Conditioning Systems. In *Proc. IBPSA Conference*, pages 652–659, 2010.

Comodeling Revisited: Execution of Behavior Trees in Modelica

Toby Myers[1] Wladimir Schamai[2] Peter Fritzson[3]

[1]Institute of Intelligent and Integrated Systems, Griffith University, Australia, toby.myers@griffithuni.edu.au
[2]EADS Innovation Works, Germany, wladimir.schamai@eads.net
[3]PELAB- Programming Environment Lab, IDA, Linköping University, Sweden, peter.fritzson@liu.se

Abstract

Large-scale systems increasingly consist of a mixture of co-dependent software and hardware. The differing nature of software and hardware means that they are often modeled separately and with different approaches. Comodeling is a design strategy that allows hardware/software integration issues to be identified, investigated and resolved in the early stages of development. Previous work described a comodeling approach that integrates Behavior Engineering with Modelica. This paper revisits this approach and introduces a new means of integration that natively executes Behavior Trees in Modelica rather than utilizing external functions. This enhanced integration has several benefits. Firstly, it makes comodeling easier to apply as the comodel is captured solely in Modelica. Secondly, it makes the ability to execute Behavior Trees widely available. Finally, it opens the possibility to use comodeling with other complementary approaches such as the virtual verification of system designs against system requirements.

Keywords: Comodeling, Behavior Engineering, Behavior Trees, Modelica, Model Driven Engineering

1. Introduction

The increasingly co-dependent nature of software and hardware in large-scale systems causes a software/hardware integration problem. During the early stages of development, the requirements used to develop a software specification often lack the quantified or temporal information that is necessary when focusing on software/hardware integration. Also early on in development, the hardware details must be specified, such as the requirements for the sensors, actuators and architecture on which to deploy the software. There is a risk of incompatibility if the software and hardware specifications contain contradicting assumptions about how integration will occur. Even if the software and hardware specifications are compatible, it is possible that a software/hardware combination with an alternative form of integration exists that would be more advantageous.

In previous work we introduced a design strategy called comodeling [3],[4] that lets developers systematically investigate and compare different software and hardware partitions to meet a system's constraints earlier in the design process, when integration problems are easier and cheaper to resolve. Our comodeling approach used multiview modeling to separately model the system's software and hardware aspects by integrating Behavior Engineering (BE) with Modelica. BE is a system and software engineering methodology that supports the engineering of large-scale dependable software intensive systems from out of its requirements. Modelica is an equation-based, object-oriented mathematical modeling language suited to modeling complex physical systems in multiple domains. BE is initially used to formalize and integrate the natural language requirements to create an executable specification. This specification is then combined with a Modelica model of the hardware and environment to create an integrated comodel.

One limitation of our comodeling approach was that the BE and Modelica models that together formed a comodel were executed separately and integrated using external functions. In this paper, we propose a new means of integration that enables a Behavior Tree (BT) to be natively executed in Modelica. This enhanced integration has several benefits. Firstly, it makes comodeling easier to apply as the comodel is captured solely in Modelica. The BT can now directly affect the acausal equations used to model the hardware and environmental components. Secondly, it makes the ability to execute Behavior Trees widely available. Finally, it opens the possibility to use comodeling with other complementary approaches such as the virtual verification of system designs against system requirements.

The remainder of this paper is structured as follows. In Section 2 we first give some background on Modelica and BE as well as briefly revisiting the automated train protection system case study we previously used to demonstrate comodeling. In Section 3 we discuss our new approach that integrates BE and Modelica by representing BTs in Modelica. Section 4 discusses some of the design decisions taken by our approach, in

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, September, 2011, ETH Zürich, Switzerland.
 Proceedings available at: http://www.ep.liu.se/ecp/056/
EOOLT 2011 website: http://www.eoolt.org/2011/

particular why we chose to use code generation rather than develop a Modelica library. Section 5 discusses how our new integration approach allows comodeling to be complemented by other approaches such as vVDR. In section 6 we present related work before concluding in section 7.

2. Background

2.1 Introduction to Modelica

Modelica [5][6][7] is an open standard for system architecture and mathematical modeling. It is envisioned as the major next generation language for modeling and simulation of applications composed of complex physical systems.

The equation-based, object-oriented, and component-based properties allow easy reuse and configuration of model components, without manual reprogramming in contrast to today's widespread technology, which is mostly block/flow-oriented modeling or hand-programming.

The language allows defining models in a declarative manner, modularly and hierarchically and combining of various formalisms expressible in the more general Modelica formalism.

The multi-domain capability of Modelica allows combining of systems containing mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process-oriented components within the same application model. In brief, Modelica has improvements in several important areas:

Object-oriented mathematical modeling. This technique makes it possible to create model components, which are employed to support hierarchical structuring, reuse, and evolution of large and complex models covering multiple technology domains.

Physical modeling of multiple application domains. Model components can correspond to physical objects in the real world, in contrast to established techniques that require conversion to "signal" blocks with fixed input/output causality. That is, as opposed to block- oriented modeling, the structure of a Modelica model naturally corresponds to the structure of the physical system.

Acausal modeling. Modeling is based on equations instead of assignment statements as in traditional input/output block abstractions. Direct use of equations significantly increases re-usability of model components, since components adapt to the data flow context for which they are used.

Several tools implement the Modelica language, ranging from open-source products such as OpenModelica [8], to commercial products like Dymola [9] and MathModelica [10].

2.2 Introduction to Behavior Trees

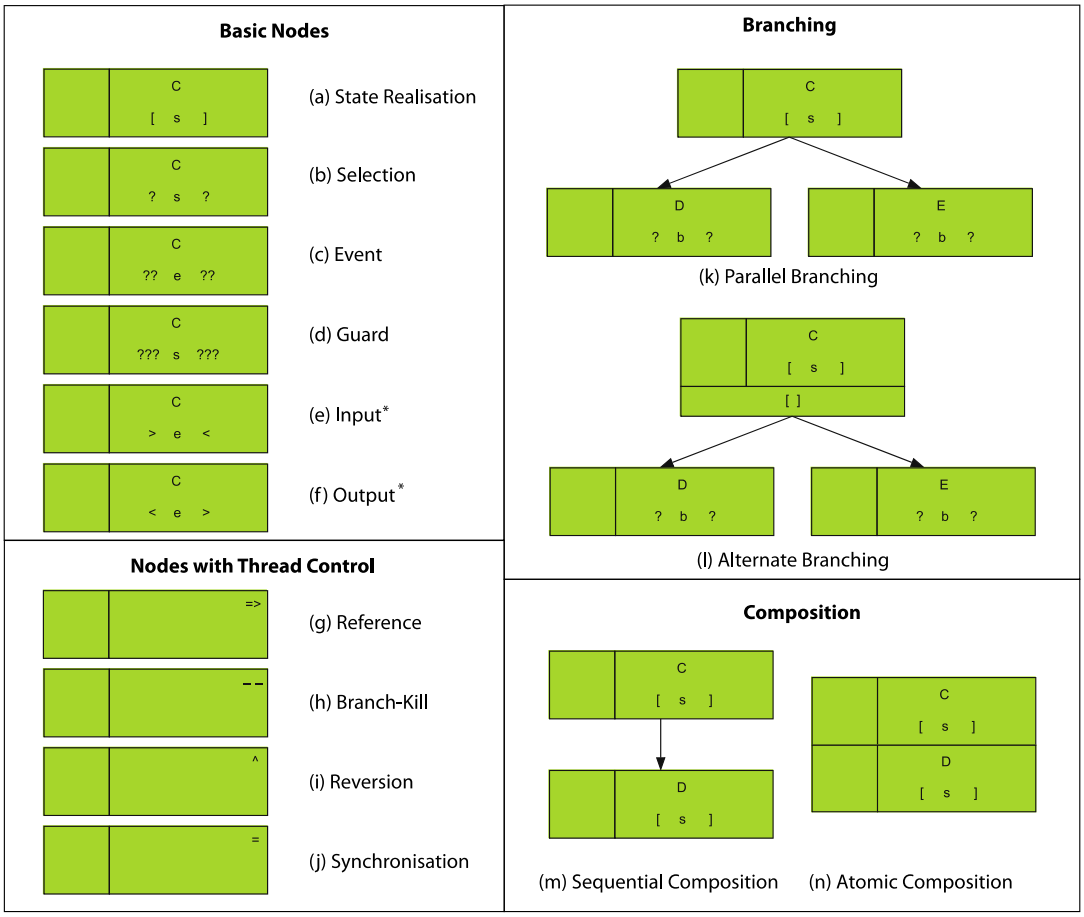
Prior to introducing BTs, it is necessary first to also have an understanding of the overall BE methodology. This is because BE is a tight interlinking of the Behavior Modeling Process (BMP) and the Behavior Modeling Language (BML). The BML consists of three integrated views: Behavior Trees, Structure Trees and Composition Trees [11],[12]. The Behavior Modeling Process (BMP) uses these views in four stages: Requirements Translation, Fitness for purpose test, Specification, and Design.

The combination of the BML and the BMP results in different uses of the views at each stage. When using BTs for example, the first stage of the BMP translates each requirement into a Requirement Behavior Tree (RBT). These are then integrated to form a single Integrated Behavior Tree (IBT), which is transformed into an executable specification in a Model Behavior Tree (MBT) and finally design decisions are applied to create a Design Behavior Tree (DBT).

Of the three views, BTs are the most relevant to the transformation to state machines as they capture dynamic behavior. A BT is a formal, tree-like graphical form that represents the behavior of individual or networks of entities which realize and change states, create and break relations, make decisions, respond to and cause events, and interact by exchanging information and passing control [11].

(A) Tag; (B) Component Name; (C) Behavior;
(D) Behavior Type; (E) Operator; (F) Label;
(G) Traceability Link; (H) Traceability Status

Figure 1. Elements of a BT Node



(a) State Realisation: Component realises the described behavior; (b) Selection: Allow thread to continue if condition is true; (c) Event: Wait until event is received; (d) Guard: Wait until condition is true; (e) Input Event: Receive message*; (f) Output Event: Generate message*; (g) Reference: Behave as the destination tree; (h) Branch-Kill: Terminate all behavior associated with the destination tree; (i) Reversion: Behave as the destination tree. All sibling behavior is terminated; (j) Synchronisation: Wait for other participating nodes; (k) Parallel Branching: Pass control to both child nodes; (l) Alternate Branching: Pass control to only one of the child nodes. If multiple choices are possible make a non-deterministic choice; (m) Sequential Composition: The behavior of concurrent nodes may be interleaved between these two nodes; (n) Atomic Composition: No interleaving can occur between these two nodes. *Note: single characters mean receive/send message internally from/to the system, double characters mean receive/send message from/to the environment.

Figure 2. Summary of the Core Elements of the Behavior Tree Notation

Figure 1 displays the contents of a BT node. The general form of a BT node consists of a main part (B-D) describing the name of the component and the behavior it exhibits qualified by a behavior type. The main part of the BT node may also have an optional operator (E) and a label (F). Each BT node also has a tag (A) which contains information to trace the node to the natural language requirements from which it was originally translated. The traceability link consists of an identifier (G) which is used to link the BT node to any associated requirements. The traceability status (H) indicates the status of this link using a set of values.

A summary of the core elements of the BT notation is shown in Figure 2. BTs are defined by a formal semantics defined in CSPsigma, an extension of CSP that can capture state based information [13].

2.3 Example: Automated Train Protection (ATP)

A key benefit of the BT notation is to provide a formal path from natural language requirements to a model of the software. To briefly demonstrate this, we will show the first two stages of the BMP using two requirements of an Automated Train Protection (ATP) system [3]. The ATP system automates the train's response to several track-side signals by sensing each signal and monitoring

the driver's reaction. If the driver fails to act appropriately, the ATP system takes control of the train and responds as required.

Table 1 shows two requirements of the ATP system. The translation of these two requirements into their corresponding RBTs is shown in Figure 4(a). Consider the RBT of requirement 6 (RBT6) with reference to the system requirements.

The first two nodes show the ATP controller receiving a value and a selection to determine if the value is a caution signal. The second node has a '+' in the tag to indicate this behavior is implied from the requirements as they do not explicitly state it is necessary to check the signal is a caution signal.

The next node shows that the Alarm is enabled, and captures that there is a relation between the Alarm and the Driver's Cab. Capturing the information about the Driver's Cab ensures that the original intent of the requirements is retained. The next BT node assumes that it is implied that the ATP Controller is responsible for observing whether the speed of the train is decreasing.

The final two BT nodes of RBT6 describe the relation between the ATP Controller and the Braking System, and the Braking System realising the activated state. Figure 4(b) also shows how integration of the two requirements proceeds using the integration point, ATP > value <.

The DBT shown in Figure 4(a) can then be used in conjunction with a Modelica model to create a comodel. This comodel can be used to determine appropriate quantified and temporal values to augment the existing requirements. For example, comodeling can be used for R6 to determine how often the speed of the train must be checked and how much of a change in speed is sufficient for the speed of the train to be considered decreasing.

R5	If a proceed signal is returned to the ATP controller then no action is taken with respect to the train's brakes.
R6	If a caution signal is returned to the ATP controller then the alarm is enabled within the driver's cab. Furthermore, once the alarm has been enabled, if the speed of the train is not observed to be decreasing then the ATP controller activates the train's braking system.

Table 1. Requirements R5 and R6 of the ATP system

3. Representing Behavior Trees in Modelica

The integration of BE and Modelica to perform comodeling was previously performed using external functions in Modelica. These external functions interacted with the Behavior Run-time Environment (BRE) where the Behavior Trees were executed. In this new approach, Modelica models are automatically generated from behavior trees captured in TextBE [14], a textual editor for capturing BE models. The generated Modelica model represents BTs using an algorithmic section that is linked to a class for each component in the BT.

Each branch of the BT is represented by an integer variable. The nodes of each branch are the represented as integer values the branch variable. This simple representation provides two benefits. Firstly, the one to one mapping of a node to the value of a branch variable makes it easy to follow the trace through the BT from the simulation output. Secondly, the branch variables simplify the representation of more complex BT constructs such as alternative branching, reversion, synchronization and branch-kill.

Each component in the BT has an integer variable, *state*, that records the current state of the component. This variable is updated using enumerated values that map an integer value to the string associated with each behavior of that component.

Several design decisions were made to simplify the execution of the BTs in Modelica. Firstly, only one node may be active at any one time. Secondly, a node is active for a user-specified delay. Finally, the execution of the BTs has been constrained to ensure fairness when multiple branches are executed in parallel.

Figure 3 shows the results of a simulation of a BT translated into Modelica that demonstrates state realisations, selections, sequential composition, alternative branching and reversion. The Modelica code and the translated BT are shown in Figure 5. The described transformation has also been included as part of the TextBE tool, together with a set of examples available at www.behaviorengineering.org.

3.1 Basic Nodes

State Realisation – A state realisation updates the *state* variable of the associated component to the enumerated value of the behavior of the BT node.

```
c.state := Integer(c.states.s);
```

Selection – A selection performs an equality check on the *state* variable of the associated component, comparing it to the enumerated value of the behavior of the BT node. Depending on the result of this equality check, the flow of control either continues or is terminated.

```
if c.state == c.state_s then
  ... // Continue flow of control
else
  ... // Terminate branch
end if;
```

Guard – A guard is similar to a selection, with the exception that the else branch is not included to ensure that the guard is continually re-evaluated until true.

```
if c.state == c.state_s then
  ... // Continue flow of control
end if;
```

Input – Inputs and outputs are implemented as boolean variables. When the variable is true, the input is active.

Events last for one cycle, to ensure that if an internal output is active in one branch, it can be received by an internal output in another branch. Inputs are represented with an equality check that is true if the associated event becomes active.

```
if e2 then
  ... // Continue flow of control
end if;
```

Output – Each output is associated with a real variable. When the boolean variable associated with the event is set to true, the real variable is set to the current time plus the value of the user-specified delay. An if statement then ensures that the output is set to false one cycle after they are activated.

```
e2 := true;
e2Delay := startTime + (delay/2);
...
if startTime > e2Delay then
  e2 := false;
end if;
```

3.2 Branching and Composition

Sequential Composition – Updates the value of the branch variable.

```
if branch1 == 1 then
  ... // Node behavior
  branch1 := 2;
elseif ...
```

Parallel Branching – Clears the current branch value and sets the branch value of the child branches to their first node.

```
if branch1 == 1 then
  ... // Node behavior
  branch1 := 0;
  branch2 := 1;
  branch3 := 1;
elseif ...
```

Alternative Branching – As per parallel branch, but when the first node of any of the child branches is activated all the sibling branches are terminated.

```
if branch2 == 1 then
  ... // Node behavior
  branch2 := 2;
  branch3 := 0;
end if;
```

Atomic Composition – Adds further constraint to all sibling branches of atomic composed nodes that flow of control cannot continue if the branch values of the atomic composed nodes are active.

```
if branch3 == 1 and not(branch2==1 or
branch2==2) then
  ... // Node behavior
  branch3 := 2;
```

3.3 Operators

Reference – Clears the current branch value and sets the branch value of the destination node.

```
if branch1 == 3 then
  branch3 := 0;
  branch2 := 2;
```

Reversion – Clears the current branch value and all sibling parallel branches and sets the branch value of the destination node.

```
if branch1 == 3 then
  branch2 := 0;
  branch3 := 0;
  branch1 := 1;
```

Branch-Kill – Clears the branch value of the destination node and the branch value of any of its descendants.

```
if branch2 == 2 then
  branch3 := 0;
  ... // Continue flow of control
```

Synchronization – One synchronisation node checks when the branch value of all nodes is set correctly and sets a boolean variable to true. All other synchronisation nodes wait until this Boolean variable is true.

```
if branch3 == 2 and branch2 == 3 then
  sync1 := true;
  ... // Node behavior
  ... // Continue flow of control

if branch2 == 3 and sync1 then
  ... // Continue flow of control
```

4. Libraries versus Code Generation

The approach taken to represent behavior trees in Modelica was heavily influenced by previous work involving the execution of UML state machines using Modelica [1]. This work raised two points of discussion: whether to use the declarative (equations) or imperative (algorithms) constructs of Modelica and whether to create a Modelica library or create a code generator.

The imperative portions of Modelica were found to be necessary to represent a comprehensive set of state machine concepts in Modelica such as inter-level transitions; entry, do, and exit actions of states, and; fork and synchronization of parallel regions. Algorithmic code was necessary for these concepts because a particular sequence of operations was required. A similar situation occurred with the translation of behavior trees to Modelica for concepts involving concurrent behavior such as alternative branching, atomic composition, reversion, reference, branch-kill and synchronization. We were unable to find a means to represent these concepts using equations but their representation in algorithmic sections was reasonably straightforward.

The second point of discussion involved whether the representation should be implemented using a Modelica library or if model transformation should be used to

generate the Modelica representation from a dedicated modeling editor. As with state machines, we chose to implement behavior trees in Modelica using the later approach.

This implementation choice is a specific instance of the two choices faced whenever any new domain-specific language (DSL) is created [19]: language invention and language exploitation. Language invention involves developing a new language from scratch. Language exploitation creates a DSL by extending an existing general purpose language (GPL), a language that uses generalized concepts that cut across multiple modeling dimensions.

Language invention is best applied when existing languages cannot be used to capture domain-specific concepts. Language invention creates DSLs that provide the most benefit to users but that also requires the development and maintenance of completely new tools such as an editor, compiler, debugger and simulator. Language invention is most suited to DSLs with a focus on syntax over semantics, which semantics often implicitly defined by model compilers or generators [20].

DSLs created by language exploitation can leverage any existing technology developed by the chosen GPL to simplify the implementation and mapping between multiple DSLs. It also allows the utilization of existing editors and the potential to benefit from a user's familiarity with the existing GPL. The ultimate success of language exploitation, however, depends on the effectiveness of the chosen GPL at capturing the concepts of the DSL.

If a model library approach were to be taken, editors would have to be used that were not specifically designed for visual languages such as state machines and behavior trees. It is likely that compromises would also have to be made to accommodate the representation of state machines or behavior trees in a Modelica library.

It is for these reasons that both the state machine and the behavior tree mappings to Modelica used a language invention approach for capturing and visualizing their models. This allows editors that were specifically designed for that DSL to be used. For model execution, however, a language exploitation approach is taken where the DSL is transformed into a Modelica representation. This approach allows the advantages of Modelica to be leveraged whilst also maintaining the advantages of having an editor tailored to the particular DSL. It also makes it possible to integrate BE with ModelicaML and vVDR as discussed in the following section.

5. Complementing Comodeling with vVDR

Comodeling, as has been outlined previously, provides a number of benefits. It leverages BE to follow a process that helps to ensure that system requirements are correct, consistent, complete, and unambiguous. It also determines the quantified and temporal information required for software/hardware interactions in the early stages of development. The effect different software and hardware partitions have on the timing, performance and complexity of individual components and on the integrated system's behavior as a whole can then be determined by simulating different comodels.

The representation of BTs natively in Modelica, however, makes it easier to integrate comodeling, and BE in general, with other approaches such as vVDR.

5.1 Introduction to vVDR

vVDR is a method for a virtual Verification of system Design alternatives against system Requirements. The application of this method is illustrated in [2] using ModelicaML. ModelicaML is a UML- and Modelica-based language. It supports all textual Modelica constructs and, in addition, supports an adopted version of the UML state machines and activity diagrams for behavior modeling as well as UML class diagram and composition diagram for structure modeling. This enables engineers to use the simulation power (i.e., solving of hybrid DAEs) of Modelica by using standardized graphical notations for creating the system models.

In the vVDR method, each requirement, or more general any analysis question, is formalized as a model which evaluates the violation or fulfilment of a requirement, or more generally provides the answer to the stated analysis question.

In case of a requirement, the formalization is done by relating measurable properties, that are addressed in the requirement statement, to each other in order to express when this requirements is violated.

Then requirements models and a system design alternative model are instantiated and bound to each other in a test model. The test model also includes the test scenario, which provides stimuli for the system model. Finally, the models are translated into Modelica code, the test model is simulated and the results are analysed.

The vVDR method and its implementation in ModelicaML focus on enabling the analysis and verification of different design alternatives against the same set of requirements. It is designed to automate the composition of test model and to facilitate a flexible way to reuse formalized requirements models, test scenario models and system design alternative models in various combinations.

For example, the same test scenario model can be used to verify different system design alternatives against the same set of requirements. Similarly, the same requirements can be used for verifications that are driven by different test scenario models.

The vVDR approach does not integrate individual requirements and, hence, does not provide means to determine inconsistencies between requirements. Inconsistencies between requirements can only be detected when requirement violations are detected during simulations.

In contrast, the BE methodology provides means to ensure the consistency of the specification. Any inconsistency or incompleteness detected in the

specification is resolved and corrected so that that each of the individual requirements does not conflict with any other requirement in the specification.

This is the starting point for the vVDR that assumes a set of requirements that do not conflict with each other. The advantage of using the vVDR approach is that it enables the verification of a system model against a subset of requirements that are of interest for a particular analysis. This is useful for the assessment of design alternatives by focusing on specific aspects as well as for regression testing when some requirements have changed or design model has evolved.

5.2 Integrating vVDR with BE and Comodeling

There are two applications that could be investigated for integrating vVDR with BE and Comodeling.

The first application is to integrate BE with vVDR by using a Model Behavior Tree as a source for the generation of both vVDR requirements violation monitors and test cases. Test cases would be used to drive a design alternative in order to determine if it fulfils the requirements by evaluating the generated requirements violation monitors.

The second application is to augment the existing comodeling approach with vVDR. vVDR would not need to provide violation monitors, as the comodels have been built from out of the requirements. Instead vVDR could provide monitors that evaluate the performance of different comodels to find the best candidate that fulfils a set of criteria.

6. Related Work

We are not aware of any other work that represents BTs in Modelica. Schamai et al have defined an approach to transform state machines into Modelica using code generation [1]. Modelica libraries also exist for state machines, e.g., Stategraph [15] and Petri nets [16]. Myers et al [17] have also created a transformation for converting BTs into executable UML state machines.

Wendland [18] has outlined an approach to augment the BMP to capture testing information and generate test cases from BTs. This could be integrated with vVDR and comodeling.

7. Conclusion

This paper presents an approach for executing BTs using Modelica in order to support comodeling and simulation of system requirements, hardware and software. Modelica is used as action language and as execution technology. By using Modelica as action language in BTs the mathematical modeling and simulation power of Modelica is leveraged. For example, Modelica equations or algorithm code is used for capturing of encapsulated behavior (e.g. inside state realization) in BTs. This enables the modeling of continuous-time, discrete-time or event-based behavior including the solving of differential algebraic equations and events handling. This way logical requirements or software models can be simulated together with physical system behavior models using the same technology.

The model-transformation rules, described in section 3, are implemented as in the code generator that can be used in TextBE[14] as a prototype. The decision for implementing a code generator instead of developing a Modelica library for BTs is discussed in section 4. The Modelica code, that is generated from BTs models, can then be loaded and simulated using OpenModelica tools [8]. This combination is the first publically available runtime execution environment for Behavior Trees models. Moreover, using Modelica as a common modeling and simulation technology enables complementing BE methodology with other approaches, such as the vVDR method implementation in ModelicaML[2], as discussed in section 5.

Acknowledgements

This research was supported under Australian Research Council Linkage Projects funding scheme (project number LP0989363) and by VINNOVA the Swedish Governmental Agency for Innovation Systems in the OPENPROD project.

References

[1] Wladimir Schamai, Uwe Pohlmann, Peter Fritzson, Christian J.J. Paredis, Philipp Helle, Carsten Strobel. Execution of UML State Machines Using Modelica In *Proceedings of the 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, (EOOLT 2010), Published by Linkoping University Electronic Press, www.ep.liu.se, In conjunction with MODELS 2010, Oslo, Norway, Oct 3, 2010.

[2] Wladimir Schamai, Philipp Helle, Peter Fritzson, Christian Paredis. Virtual Verication of System Designs against System Requirements, In *Proc. of 3rd International Workshop on Model Based Architecting and Construction of Embedded Systems* (ACES 2010). In conjunction with MODELS 2010. Oslo, Norway, Oct 4, 2010.

[3] Toby Myers, Geoff Dromey, Peter Fritzson. Comodeling: From Requirements to an Integrated Software/Hardware Model, *IEEE Computer* 44(4) pp.62-70, April 2011 doi: 10.1109/MC.2010.270.

[4] Toby Myers. *The Foundations for a Scaleable Methodology for Systems Design*, PhD Thesis, School of Computer and Information Technology, Griffith University, Australia, 2010.

[5] Modelica Association. Modelica: A Unified Object-Oriented Language for Physical Systems Modeling: Language Specification Version 3.0, Sept 2007. http://www.modelica.org.

[6] Michael Tiller. Introduction to Physical Modeling with Modelica. Kluwer Academic Publishers, 2001.

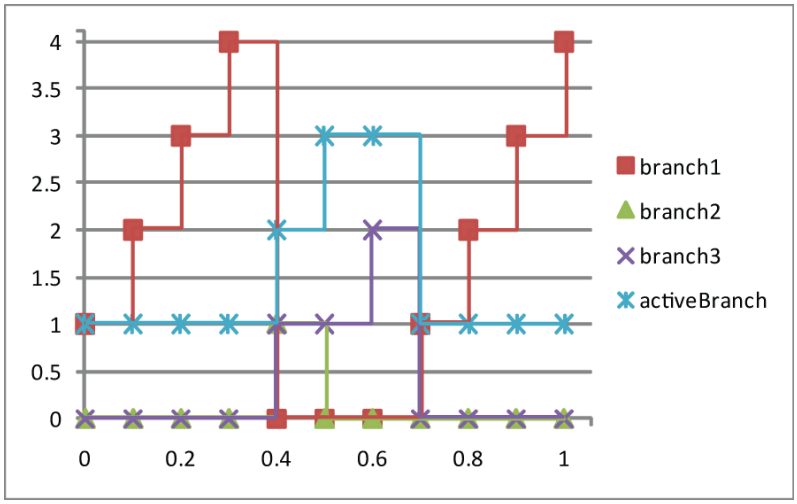
[7] Peter Fritzson. Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Wiley-IEEE Press, 2004.

[8] Open Source Modelica Consortium. OpenModelica. http://www.openmodelica.org.

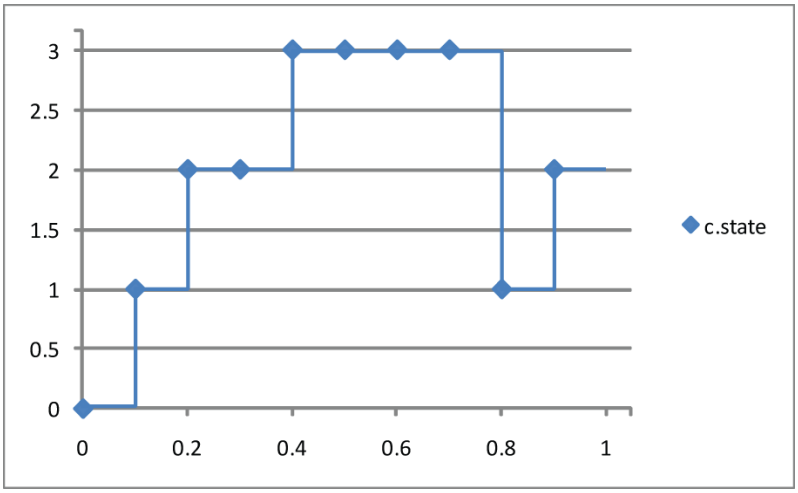
[9] Dynasim. Dymola. http://dynasim.com.

[10] MathCore. Mathmodelica. http://www.mathcore.com.

[11] Geoff Dromey. From requirements to design: Formalizing the key steps. in Proc. Conf. on Software Engineering and Formal Methods (SEFM). IEEE Computer Society, pp. 2-13, 2003.
[12] Geoff Dromey. Climbing over the "no silver bullet" brick wall. IEEE Software, vol. 23, pp. 120, 118-119, 2006.
[13] Robert Colvin and Ian Hayes. A semantics for Behavior Trees using CSP with specification commands. *Science of Computer Programming*, In Press, Corrected Proof, Available online 9 December 2010.
[14] Toby Myers. TextBE: A Textual Editor for Behavior Engineering. *Proceedings of the 3rd Improving Systems and Software Engineering Conference* (ISSEC), Sydney, Australia, 2-5 August 2011 (Accepted).
[15] Martin Otter, Martin Malmheden, Hilding Elmqvist, Sven Erik Mattsson, and Charlotta Johnsson. A New Formalism for Modeling of Reactive and Hybrid Systems. In *Proceedings of the 7th International Modelica Conference*, Como, Italy. September 20-22, 2009.
[16] Sabrina Proß and Bernhard Bachmann. A Petri Net Library for Modeling Hybrid Systems in OpenModelica. *Proceedings of the 7th International Modelica Conference*, Como, Italy, 20-22 September 2009.
[17] Toby Myers, M. Wendland, S. Kim, Peter Lindsay. From Requirements to Executable UML State Machines: A Formal Path using Behavior Engineering and M2M Transformations, *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, Wellington, New Zealand, 16-21 October 2011 (submitted).
[18] M. Wendland, I. Schieferdecker, A. Vouffo-Feudjio. Requirements driven testing with behavior trees. In *Proceedings of the ICST Workshop Requirements and Validation, Verification & Testing (ReVVerT 2011)*. 2011. Accepted.
[19] M. Mernik, J. Heering and A. M. Sloane. When and how to develop domain-specific languages, *ACM Computing Surveys (CSUR),* vol. 37(4), 2005. pp. 316-344.
[20] J. Heering and M. Mernik, Domain-specific languages in perspective, Tech. rep., CWI, sEN-E0702. 2007.



(a)



(b)

Figure 3. Simulation Plots of Selection Example Translation (a) branch and activeBranch variables (b) c.state variable

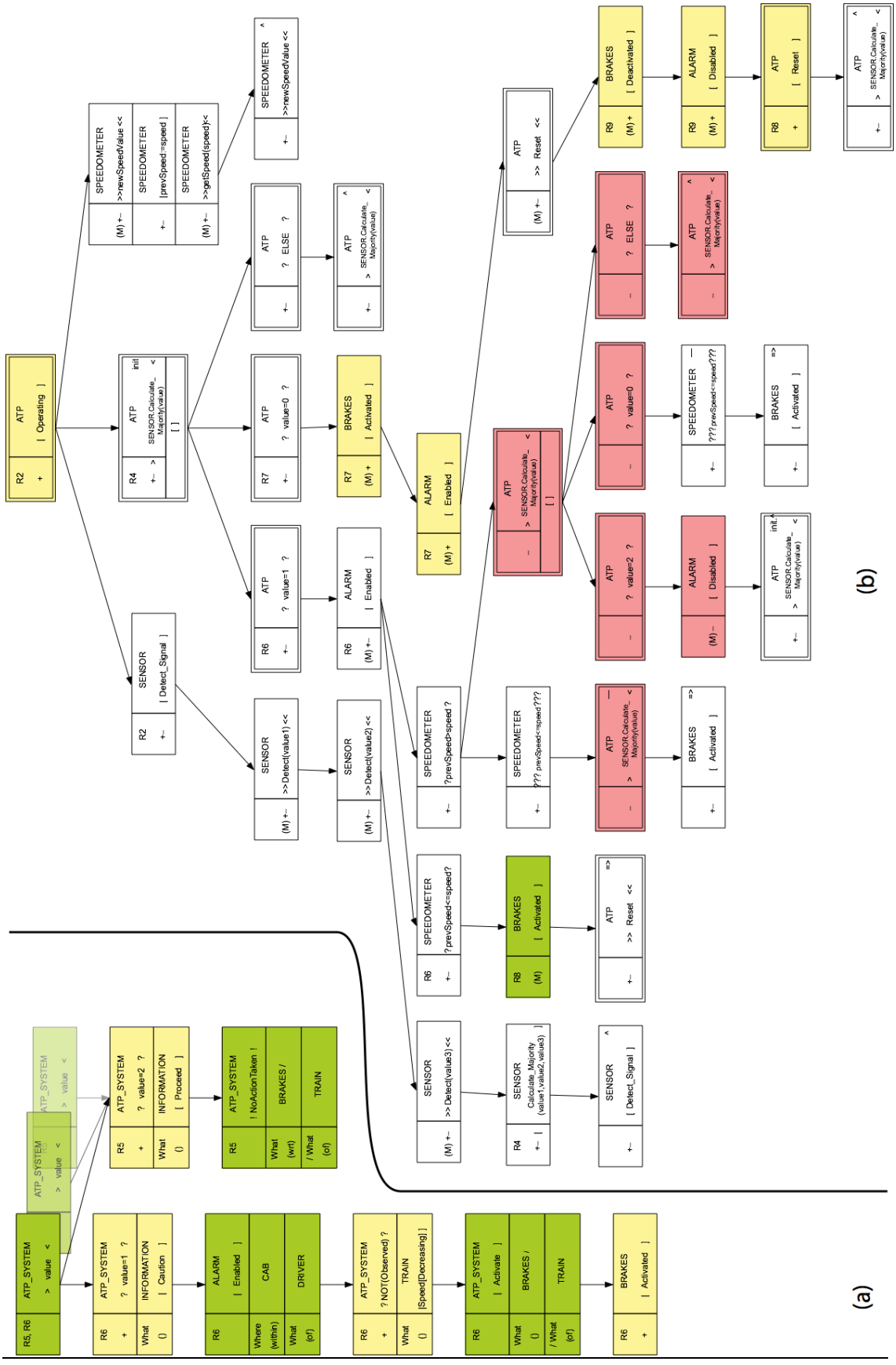


Figure 4. Behavior Engineering model of the Automated Train Protection System (a) Integration of Requirements R5 and R6 (b) Design Behavior Tree

```
model Selection
 Integer branch1(start=1),branch2,branch3;
 Integer activeBranch(start=1),
 lastActiveBranch;
 Boolean next(start=true);
 C c;
 Real startTime;
 constant Real delay = 0.1;

 algorithm
 when time>startTime + delay then
  while branch1==pre(branch1) and
 branch2==pre(branch2) and
 branch3==pre(branch3) and next loop
   if activeBranch==1 then
    if branch1==1 then
     c.state := Integer(c.states.s);
     branch1:=2;
    elseif branch1==2 then
     c.state := Integer(c.states.t);
     branch1 := 3;
    elseif branch1==3 then
     if c.state==c.state_t then
      branch1 := 4;
     else
      branch1 := 0;
     end if;
    elseif branch1==4 then
     c.state := Integer(c.states.u);
     branch1 := 0;
     branch2 := 1;
     branch3 := 1;
    end if;
   elseif activeBranch==2 then
    if branch2==1 then
     if c.state==c.state_s then
      branch2 := 2;
      branch3 := 0;
     else
      branch2 := 0;
     end if;
    elseif branch2==2 then
     c.state := Integer(c.states.v);
     branch2 := 0;
    end if;
   elseif activeBranch==3 then
    if branch3==1 then
     if branch2==0 then
      branch3 := 2;
     end if;
    elseif branch3==2 then
     branch3 := 0;
     branch1 := 1;
    end if;
   end if;

   activeBranch := activeBranch + 1;
   if activeBranch==4 then
    activeBranch := 1;
   end if;
   if activeBranch==1 and branch1==0 then
    activeBranch := activeBranch + 1;
   end if;
   if activeBranch==2 and branch2==0 then
    activeBranch := activeBranch + 1;
   end if;
   if activeBranch==3 and branch3==0 then
    activeBranch := 1;
   end if;
   if activeBranch==1 and branch1==0 then
    activeBranch := activeBranch + 1;
   end if;
   if activeBranch==2 and branch2==0 then
    activeBranch := activeBranch + 1;
   end if;
   if activeBranch==lastActiveBranch then
    next := false;
   end if;
  end while;
  startTime := time;
  lastActiveBranch := activeBranch;
  next := true;
 end when;
end Selection;

class C
 type states = enumeration(s,t,u,v);
 constant Integer state_s =
Integer(states.s);
 constant Integer state_t =
Integer(states.t);
 Integer state;
end C;
```

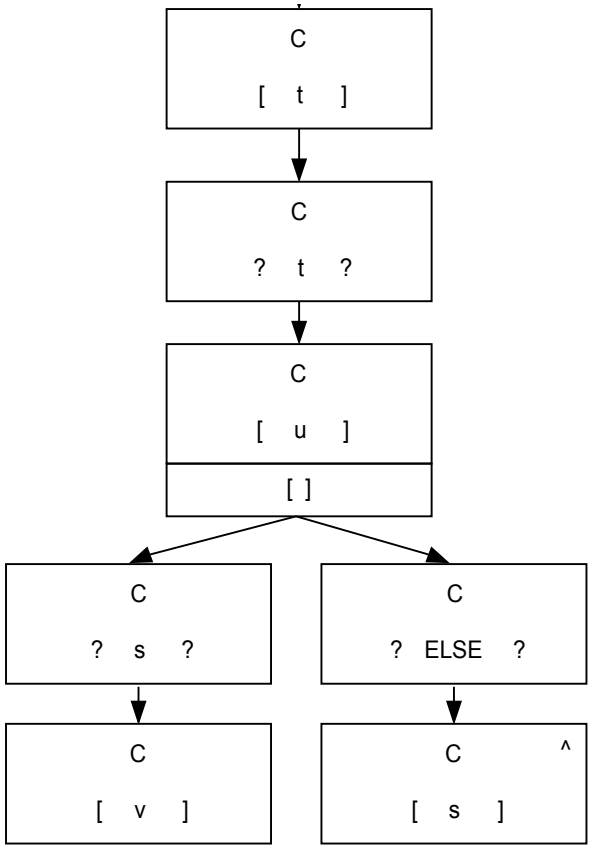


Figure 5. Example Translation of a Behavior Tree into Modelica

Exploiting OpenMP in the Initial Section of Modelica Models (Work in Progress)

Javier Bonilla[1] Luis J. Yebra[1] Sebastián Dormido[2]

[1]PSA-CIEMAT, Plataforma Solar de Almería - Centro de Investigaciones Energéticas, MedioAmbientales y Tecnológicas, Almería, Spain, {javier.bonilla,luis.yebra}@psa.es

[2]UNED, Universidad Nacional de Educación a Distancia, Madrid, Spain, sdormido@dia.uned.es

Abstract

This paper presents a practical case where parallelization for multi-core processors can be exploited in Modelica models using OpenMP. Although this parallelization is applied to a particular case and to a particular section in the code, the parallel implementation is straightforward and a gain in speed of around 11% is obtained. The particular section in the code is the initial section and the particular case is to consider that the initial section is time consuming and the operations are independent from each other. The particular case is tested in a real dynamic model during the simulation and calibration processes. The most appealing feature of this method is that it is straightforward to implement and that it could be easily adopted by equation-based modelling languages. On the other hand, the process is not automatically performed and the modeller needs to have a minimum knowledge about parallel computing.

Keywords Modelica, OpenMP, parallelization, initial section, multi-core processors

1. Introduction

Nowadays, modern equation-based object-oriented (EOO) modelling languages are continuously increasing their expressiveness to describe and model complex systems. However, there is an important drawback of having large and complex system models, the required computational effort to simulate is very high.

Commonly, EOO models are compiled as single-threaded executables not taking advantage of the newest multi-core processors available even on desktop computers. Moreover, and considering the particular case of Modelica [13], the extension of the language to consider multi-core processors is not easy due to the flattening of the equation-based object-oriented Modelica code into C code.

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zürich, Switzerland.

 Proceedings available at: http://www.ep.liu.se/ecp/056/

EOOLT 2011 website: http://www.eoolt.org/2011/

With regard to parallelization in EOO modelling languages it is worth mentioning [3, 4, 12, 19]. This paper does not pretend to be a meticulous study about parallelization in EOO modelling languages, it just describes a straightforward parallelization method in the initial section of the resulting C source code obtained from the Modelica model, and not considering the parallelization in the numerical integration process.

2. OpenMP

OpenMP (Open Multi-Processing) [2] is an application programming interface (API) for multi-platform (Unix and Microsoft Windows platforms) shared-memory parallel programming in C/C++ and Fortran. It provides a simple and flexible interface to develop parallel applications by means of a set of compiler directives, library routines, and environment variables. Its applicability ranges from desktop computers to clusters and supercomputers.

OpenMP is maintained by the OpenMP Architecture Review Board (ARB). The first OpenMP API specifications was released in 1997 for Fortran 1.0 whereas for the C/C++ languages was released the following year, in 1998. The OpenMP version 2.0 was released in 2000 and 2002 for Fortran and C/C++ respectively. In 2005, both versions were unified in version 2.5. Version 3.0 was released in 2008. The current version, version 3.1, has been recently released in June, 2011.

The key concept in OpenMP is multithreading, in this method of parallelization the master thread forks a series of slave threads for particular parallel tasks, the run-time environment is responsible of allocating the threads to different processors or cores. Figure 1 illustrates this concept, where the master thread forks 2 additional threads for tasks I, 3 additional threads for task II and 1 additional thread for task III.

The core elements of the OpenMP API are summarized and briefly explained in the following list.

- **Parallel control directives.** They control the flow execution of the program (i.e. *parallel* directive).
- **Work sharing.** They distribute the execution of instruction among the processors or cores (i.e. *parallel for* or *section* structures).

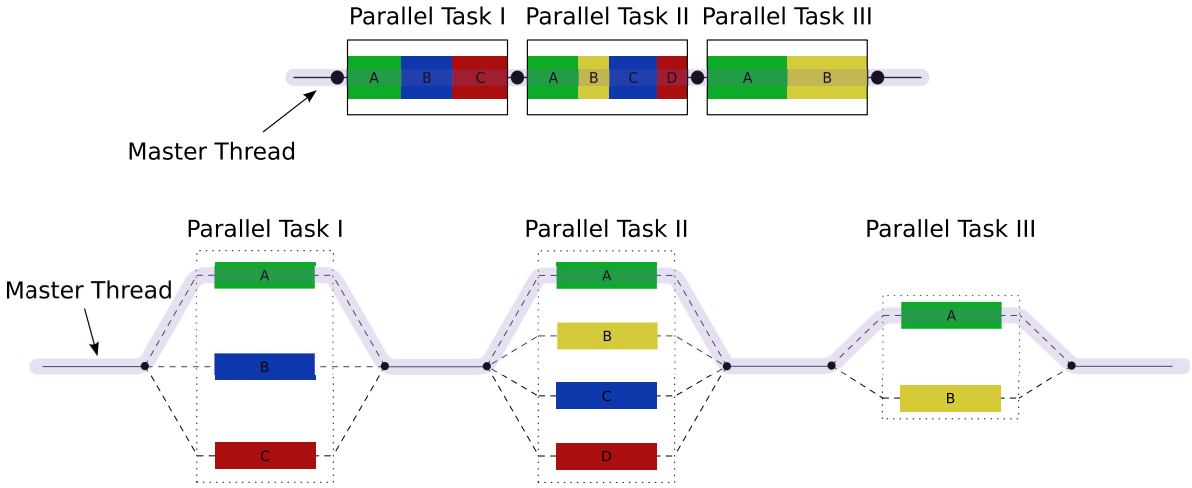


Figure 1. OpenMP multithreading concept [1].

- **Data environment.** It is defined by the scope variables, *shared* and *private*.
- **Synchronization.** It is controlled by the *critical*, *atomic* and *barrier* directives.
- **Run-time functions.** Set and provide run-time information (i.e. *omp_set_num_thread()* and *omp_get_num_thread()* which set and obtain the maximum number of threads that can be allocated).

3. Real System Under Study

This section explains briefly the facility under study, the developed Modelica model and the defined simulator scheme to facilitate the simulation, calibration and validation processes.

3.1 DISS facility: Direct Steam Generation Parabolic-Trough Solar Thermal Power Plant

The real system under study is the CIEMAT-PSA (Centro de Investigaciones Energéticas Medioambientales y Tecnológicas - Plataforma Solar de Almería, a Spanish government research and test center) DISS (DIrect Solar Steam) test facility, a parabolic-trough solar thermal power plant. A general view of this facility is shown in Figure 2.

The parabolic-trough technology is one of the several different solar thermal concentrating technologies available. Parabolic-Trough Collectors (PTCs) are solar concentrators which convert the direct solar radiation into thermal energy, heating a heat transfer fluid (HTF) up to around 675 K. Their high working temperature makes PTCs suitable for supplying heat to industrial processes, replacing traditional fossil fuels [11] [18].

The HTF used in the DISS test facility is the two-phase-flow steam-water, which circulates in three different states, subcooled liquid, steam-water mixture and superheated steam.The aim of the DISS facility is to develop a new generation of solar thermal power plants using parabolic-trough collectors to produce high pressure steam in the absorber tubes, thus eliminating the oil commonly used as a heat transfer medium between the solar field and the conventional power block. This kind of technology is known as Direct Steam Generation (DSG), it increases overall system efficiency while reducing investment costs.

3.2 Modelica model

A DISS solar thermal power plant equation-based object-oriented dynamic model was developed to study the behavior of the real plant [17]. An EOO methodology was chosen in order to describe the system as a set of equations which are acausal, maintaining it mathematical meaning. Moreover, a object-oriented methodology allows to define basic models which can be reused to develop new complex models without additional effort [5]. This methodology allows to develop reusable and easy to maintain components.

The modelling language chosen was Modelica. Modelica is developed and maintained by the Modelica Association. Modelica is a suitable modelling language to develop complex mathematical models of physical systems. Modelica also has useful libraries to develop thermo-hydraulic systems. Modelica Media [14] is a thermodynamic properties computation library, which follows IAPWS (the International Association for the Properties of Water and Steam) recommendations in its latest IF97 formulation, (Industrial Formulation 1997) [10]. This formulation is optimised for short computing times and low CPU load. Furthermore, the ThermoFluid library [16, 8] provides a framework and basic components for modelling thermo-hydraulic and pro-

Figure 2. General view of the DISS test facility owned by Plataforma Solar de Almería (CIEMAT).

cess systems in Modelica. The Integrated Development Environment (IDE) chosen, which supports the Modelica language, has been Dymola [7].

Figure 3 shows the DISS test facility Modelica component diagram which has 11 PTC components. The model inputs are: the ambient temperature ($Tamb$), the solar radiation (Rad), regarding the HTF, the inlet temperature ($inletTemp$), the inlet pressure ($inletPres$) and the mass flow ($mdot\_ws$) of the fluid, the three last inputs are also provided for the last PTC injector.

3.3 Simulator scheme

With the aim of facilitating the settlement of the initial conditions for the simulation, the calibration and validation processes, a simulator scheme was developed. Figure 4 shows the DISS facility simulator scheme which is not only the model itself but a series of tools to facilitate the simulation.

Among the developed tools in the simulator scheme, it is worth mentioning the following.

- A web application to easily access the real data from the data acquisition system (DAS) with the aim of comparing the real data with the simulation results and also to obtain the initial conditions for the dynamic simulation.
- A computer program to convert the real data obtained from the web application to a input trajectory file used in the Dymola IDE.
- A Modelica library to retrieve the initial values from the trajectory file and solve the initial condition problem. This Modelica library reads the input trajectory file to set values to certain parameters which are used to solve the initial condition problem. These reading operations are time consuming and they can be easily parallelized.

The calibration process has been performed using Matlab/Simulink [15]. Dymola includes mechanisms to export Modelica models to Simulink in a easy and direct way using a Simulink block where parameters and inputs can be defined. The Matlab Genetic Algorithm Toolbox [6] has been used to calibrate the DISS model, a multi-objective approach was selected in order to minimize the absolute value of the percentage relative error between the real and simulated output temperatures for each PTC. Figure 5 shows the DISS Simulink block in Matlab. More information about calibration of Modelica models in Matlab together with a practical example can be found in [9].

4. Initial Section Parallelization

As previously explained in section 3.3, a Modelica library was developed to retrieve real data from the input trajectory file with the aim of setting several parameters to solve the initial condition problem. Reading from a text file is a time consuming operation. Some data must be read from this input file, including the initial inlet and outlet pressure, temperature, mass flow of the HTF, etc. These reading operations can be easily parallelized because they are independent from each other.

The procedure to parallelize the initial section in the resulting C code, obtained from the translation of the Modelica model by the Dymola tool, is the following.

1. In the resulting C source code generated by the Dymola tool (*dsmodel.c*), it has been included a reference to the OpenMP header (*omp.h*), as shown in Listing 1, in order to use in the source code, the OpenMP API, directives and functions.

```
1  #include <omp.h>
```

Listing 1. OpenMP header inclusion

2. Set the maximum number of threads to be executed using the *omp_set_num_threads()* function as shown in Listing 2. In our particular case this value was set considering the number of the processor cores, 4 cores.

```
1  constant int NCores = 4;
   omp_set_num_threads(NCores);
```

Listing 2. Setting the maximum number of threads

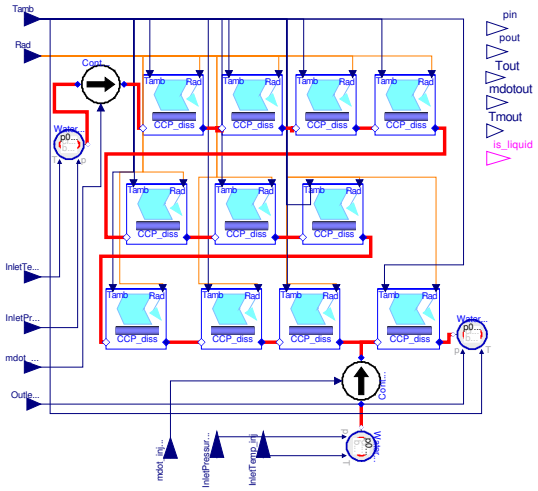


Figure 3. DISS test facility Modelica model.

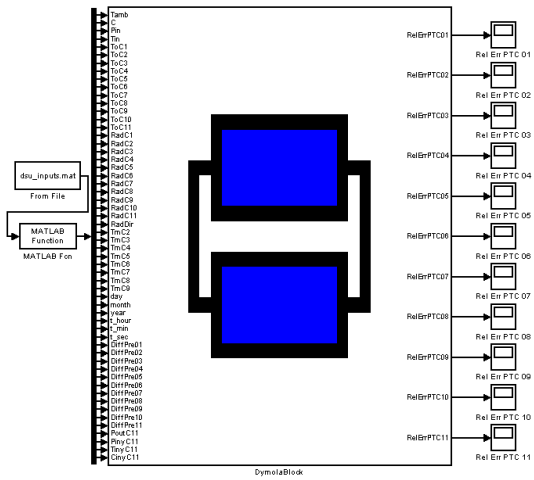


Figure 5. DISS Simulink block.

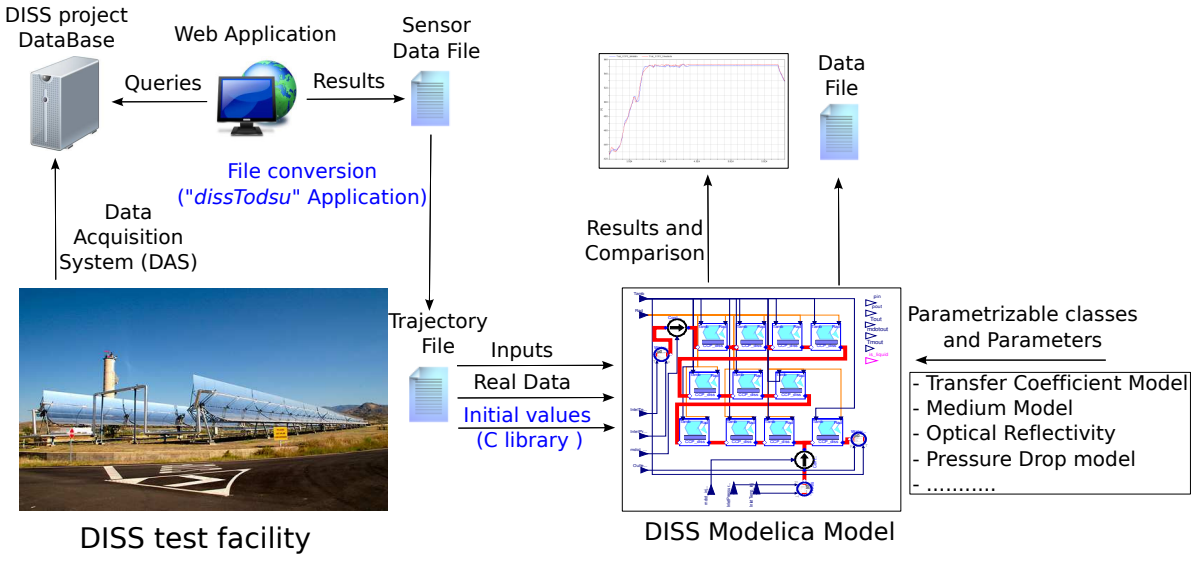


Figure 4. DISS simulator scheme.

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {
      ........
      ........
      T[0]=getIniValue(iniTime,"Tem PTC1");
      T[1]=getIniValue(iniTime,"Tem PTC2");
      ........
      ........
    }
    #pragma omp section
    {
      ........
      ........
      p[0]=getIniValue(iniTime,"Pres PTC1");
      p[1]=getIniValue(iniTime,"Pres PTC2");
      ........
      ........
    }
    #pragma omp section
    {
      ........
      ........
      r[0]=getIniValue(iniTime,"Rad PTC1");
      r[1]=getIniValue(iniTime,"Rad PTC2");
      ........
      ........
    }
    #pragma omp section
    {
      ........
      ........
      m[0]=getIniValue(iniTime,"Mflow PTC1");
      m[1]=getIniValue(iniTime,"Mflow PTC2");
      ........
      ........
    }
  }
}
```

Listing 3. OpenMP parallel sections

3. Locate the initial section/s in the *dsmodel.c* file which can be identified in the C source code by the *InitialSection* identifier.

   After that, the sentences in the initial section must be manually distributed between the different threads to balance the computational load between cores. For that purpose the OpenMP *sections* directives were used, the source code is shown in Listing 3.

   First, the *parallel* directive was used to indicate that the following sentences must be executed in parallel, then the *sections* and *section* directives were used to define 4 sections which correspond with 4 threads. The reading operations were equally distributed between the 4 sections, each one executed by a different thread in each available processor core.

4. Compile the *dsmodel.c* including the OpenMP library. For the GNU Compiler Collection (GCC) in Linux operating systems, only the *-fopenmp* modifier is necessary to compile the *dsmodel.c* source code with OpenMP support. For that purpose, a script file, *compile.sh*, was created to include the previously mentioned modifier in the compilation script. The default compilation script for the Dymola tool in Linux operating systems is *dsbuild.sh*.

The previously described procedure to parallelize the initial section of Modelica models must be done manually. It could be worth studying how this procedure could be performed directly in the Modelica code and not in the resulting C source code. However, this is not easy, only constant and parameter values can be paralellized with this approach. Obviously, time consuming operation must be involved in the computation of these values because in other case no performance improvement in simulation will be obtained. It must be also taken into consideration the dependencies between these values to properly distribute their calculation between the different threads.

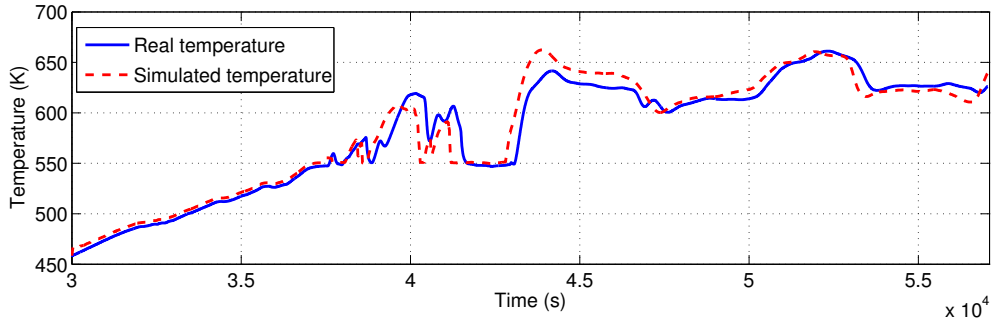


Figure 6. Real and simulated output DISS field temperature in the calibration process.

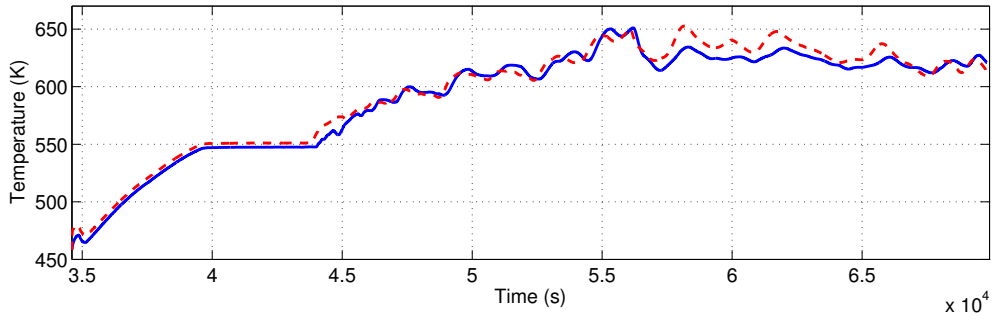


Figure 7. Real and simulated output DISS field temperature in the validation process.

5. Simulation Results

The tests presented in this section were performed using a Intel® Core™ i5 CPU M540, 2.53 GHz. Several tests using different input trajectory files from several operation days were performed and a mean execution time has been calculated. It is important to note that the parallelization is only considered in the initial section and not during the numerical integration.

The simulation statistics are summarized in Table 1. Although the gain in speed is not enormous, just a global speedup of 1.13 was obtained, the calibration process using genetic algorithms is time consuming and even a modest performance improvement is worthwhile. Moreover, if it is considered that the parallelization was only performed in the initial section and not in the whole simulation, the speedup obtained is considerably high, 2.80, because the mean initial section secuential and parallel execution times are 31.231 and 11.127 seconds respectively.

With respect to the calibration process using genetic algorithms, if 20 elements in the population and 100 iterations are considered, the calibration execution time is reduced in 4.5 hours. Although it keeps being time consuming, because it lasted 3 days and 4.5 hours in being performed.

As an example, Figures 6 and 7 show the real DISS test field output temperature (solid line) and the simulated DISS test field output temperature (dashed line) for two different operation days. Figure 6 corresponds to a day used in the calibration process whereas Figure 7 corresponds to a day used in the validation process. The x axis represents the time in seconds from the beginning of the day and the y axis shows the real and simulated DISS test field output temperatures in kelvin.

6. Conclusions and Future Work

The most remarkable conclusions are the following.

- OpenMP can be easily used to parallelize the initial section in the resulting C source code obtained from Modelica models when necessary.
- To take advantage of the parallelization in the initial section of Modelica models it is required time consuming calculations in the initial section.
- The proposed approach has been tested in a dynamic model, which has been validated by experimental data, obtaining a mean speedup of 1.13 in the whole simulation and a mean speedup of 2.80 in the initial section where the parallelization takes place.
- The gain in speed is worth not only in simulation but also in the calibration process where the simulation time is a critical aspect.

Kind of model	Original	Parallelized
Execution time (s)	170.727	151.415
Execution time speedup	1	1.13
Initialization section time (s)	31.231	11.127
Initialization section speedup	1	2.80
Calibration execution time	3 d 9 h	3 d 4.5 h

Table 1. Simulation statistics

As future work it would be useful to study and tackle the following open issues.

- Study how to include mechanisms to describe the use of the OpenMP API directly in the Modelica code instead of using it in the resulting C source code.
- Study how to take advantage of the OpenMP API during the whole simulation, specially in the numerical integration process, and not only in the initial section of Modelica models.
- Take advantage of a 13-node cluster and consider the parallelization not only in the simulation but also in the genetic algorithm calibration method by evaluating concurrently members of the population in each cluster's node.

Acknowledgments

This work has been financed by CIEMAT research centre and UNED own funds, and by the National Plan Project DPI2010-21589-C05-04 of the Spanish Ministry of Science and Innovation and FEDER funds. This support is gratefully acknowledged by the authors. This work has been also performed within the scope of the collaboration agreement between the Automatic Control Group of CIEMAT at Plataforma Solar de Almería, and the Computer Science and Automatic Control Department of UNED.

References

[1] OpenMP, wikipedia the free encyclopedia, `http://en.wikipedia.org/wiki/OpenMP`.

[2] The OpenMP Architecture Review Board ARB. The OpenMP API specification for parallel programming, `http://openmp.org/`.

[3] Peter Aronsson. *Automatic Parallelization of Equation-Based Simulation Programs*. Doctoral thesis No 1022, Linköping University, Department of Computer and Information Science, 2006.

[4] Peter Aronsson and Peter Fritzson. A task merging technique for parallelization of modelica models. In *Modelica conference, Hamburg, Germany*, 2005.

[5] F.E. Cellier. Object-oriented modeling: Means for dealing with system complexity. In *Proc. 15th Benelux Meeting on Systems and Control*, pages 53–64, Mierlo, The Netherlands, 1996.

[6] Andrew Chipperfield, Peter Fleming, Hartmut Pohlheim, and Carlos Fonseca. Genetic algorithm toolbox for use with matlab. Technical report, 1994.

[7] A.B. Dynasim. *Dymola 6.0 User Manual*, 2006. `http://www.dynasim.se`.

[8] J. Eborn. *On Model Libraries for Thermo-hydraulic Applications*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, March 2001.

[9] K. Hongesombut, Y. Mitani, and K. Tsuji. An incorporated use of genetic algorithm and a modelica library for simultaneous tuning of power system stabilizers. In Modelica Association, editor, *Modelica Conference 2002 Proceedings*, pages 89–98, Germany, March 18-19 2002. Modelica Association and Deutsches Zentrum für Luft- und Raumfahrt (DLR).

[10] IAPWS. Release on the IAPWS Industrial Formulation 1997 for the Termodynamic Properties of Water and Steam. Technical report, The International Association for the Properties of Water and Steam, Erlangen, Germany, September 1997.

[11] C.F. Kutshcer, R.L. Davenport, D.A. Dougherty, R.C. Gee, P.M. Masterson, and E. Kenneth. Design approaches for solar industrial process-heat system. Tech. rep. seri/tr-253-1356, Solar Energy Research Institute, Golden (Colorado), USA, 1982.

[12] Håkan Lundvall. *Automatic Parallelization using Pipelining for Equation-Based Simulation Languages*. Licentiate thesis No 1381, Linköping University, Department of Computer and Information Science, 2008.

[13] Modelica Association. Modelica specification 2.2.1, 2007.

[14] Modelica Association. Modelica standard library 2.2.1, 2007.

[15] The MathWorks, Inc. *MATLAB Documentation*. The MathWorks, Inc., 2009.

[16] Hubertus Tummescheit, Jonas Eborn, and Falko Wagner. Development of a modelica base library for modeling of thermo-hydraulic systems. In Modelica Association, editor, *Modelica 2000 Workshop Proceedings*, Lund, October 2000.

[17] L. J. Yebra. *Object-Oriented Modelling of Parabolic-Trough Collectors with Modelica (in Spanish)*. PhD thesis, Universidad Nacional de Educación a Distancia (UNED), Madrid, Spain, 2006.

[18] E. Zarza. *The Direct Steam Generation with Parabolic Collectors. The DISS project (in Spanish)*. PhD thesis, Escuela Superior de Ingenieros Industriales de Sevilla, Seville, Spain, Nov 2000.

[19] Per Östlund. Simulation of Modelica Models on the CUDA Architecture. Master's thesis, Linköping University, Department of Computer and Information Science, 2009.

Separate Compilation of Causalized Equations - Work in Progress

Christoph Höger
Technische Universität Berlin
christoph.hoeger@tu-berlin.de

Abstract

Separate Compilation is currently considered impossible for Modelica in practice, because several features of state-of-the-art Modelica compilers currently rely on global information. One prominent example for those features is causalization. This particular feature is very important for the generation of fast simulation code. In this work we show how a post-compilation causalization can fit into the operational semantics of a language like Modelica. We present the semantics of a causalizing language system are together with a prototype. This prototype shows that separately compiled models can form a causalized, thus fast, simulation program.

1. Introduction

Separate Compilation as defined in [5] is the process of creating code fragments that can be used in *any* (type-valid) context. This method is a standard approach in software engineering and allows both code re-usage and modularization. Those features are prominent design goals for Modelica [1], yet no current implementation currently implements a separate compilation model.

As we have already shown in [9], there is no principal reason, why Modelica cannot be separately compiled as a language. The only necessary difference is to interpret the process of flattening as the *operational semantics* of the language. Naturally this means that the process of flattening must then happen at runtime.

Although Modelica thus supports separate compilation, there are several obstacles when it comes to *efficient* code generation: Symbolic methods for the handling of systems of equations seem to naturally demand the presence of those equations in a uncompiled form.

One of those methods is *causalization* as described e.g. in [6]. Causalization can drastically enhance the simulation performance of models with sparse systems of equations. Therefore we consider it a must-have for a decent Modelica implementation. Since we also consider separate compilation a necessity, the question arises, how that particular symbolic method can be implemented *after* code generation. That question shall be answered.

The rest of the paper is organized as follows: We will first give a formal definition of causalization in the context of a tiny modeling language, TinyModelica. We will also present the operational semantics of that language. Subsequently we will discuss (on the basis of linear equations) how models in TinyModelica can be compiled separately and causalized afterwards. Finally we will present our current prototype implementation and demonstrate its performance compared to OpenModelica.

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zrich, Switzerland.

2. TinyModelica Syntax

For the purpose of this document, we will describe a small modeling language, called TinyModelica. We use this language to give a formal overview of the instantiation process of a model and its causalization.

TinyModelica is a very basic language for modeling continuous systems. It does not allow inheritance nor modifications. In fact, it does not even contain parameters or initial value definitions. Its only feature is the definition of models that may contain variables, equations and sub-models.

2.1 Syntax

The syntax of TinyModelica is defined below in EBNF form. We use the $*$ operator to denote repetitions of any (including zero) length and the $+$ operator for repetitions of length ≥ 1.

$$
\begin{array}{lcl}
\mathit{Statement} & ::= & \texttt{causalize} \\
 & | & \texttt{solve (}\mathit{Block}^{+}\texttt{)} \\
 & | & \mathit{Var} \\
 & | & \mathit{Eq} \\
 & | & \mathit{ModelDef} \\
 & | & \mathit{Statement}\ ;\ \mathit{Statement} \\
\mathit{ModelDef} & ::= & \texttt{model id}\ \mathit{Var}^{*}\ \mathit{Eq}^{*}\ \texttt{end} \\
\mathit{Eq} & ::= & \texttt{equation id}\ \mathit{Block}^{*} \\
\mathit{Var} & ::= & \texttt{var id :}\mathit{Name} \\
\mathit{Name} & ::= & \texttt{id}\ [.\ \mathit{Name}] \\
\mathit{Block} & ::= & \mathit{Name} := \mathit{Expr}
\end{array}
$$

A program contains a list of statements. Every model definition contains typed variables (including sub-models) and equations. Additionally, TinyModelica contains two kinds of special statements: `causalize` and `solve`. Their formal semantics will be discussed below.

One important difference between Modelica and TinyModelica is the syntax of equations. While the former one supports nearly mathematical notations (limited only by the restrictions of text editors), like $\dot{x} = f(y,t)$, TinyModelica's equations are defined by sets of *blocks*.

Every block is a variable name together with an expression. Informally, such a block represents *one solution* to an equation. This difference in presentation makes it possible to compile models separately.

Also note that our definition of equations contains a *name*. This is not only necessary for some of the semantics given below. It also seems natural to allow a modeler to at least name some equations. This ease the process of analyzing models as well as simplifying error reporting.

As a special restriction, TinyModelica does not contain a definition of its expression language. This is very important: We want to show, that separate compilation of TinyModelica models is possible. Since we do not give a definition of expressions, there is no way to use any symbolic information in the operational semantics.

This restriction also allows us to ignore important modeling aspects like integration or time events. We simply focus on algebraic equations and assume those features to be part of a decent runtime implementation (in fact, our prototype supports both).

3. TinyModelica Operational Semantics

The operational semantics of TinyModelica are rather simple. Since we do not cover features like inheritance of types, the rules given below should be easy to understand.

In the following, we will use a simple environment structure Γ. This environment can be seen as a partial function mapping names to real values.

$$\Gamma : Name \hookrightarrow \mathbb{R}$$

We also introduce another partial function $\mathcal{M}$, mapping names to model definitions. And $\mathcal{E}$ for the mapping of equations.

$$\mathcal{M} : Name \hookrightarrow Model$$

$$\mathcal{E} : Name \hookrightarrow Equation$$

The *states* of a TinyModelica program are tuples of three such functions:

$$State \subseteq \{(\mathcal{M}, \mathcal{E}, \Gamma) | \mathcal{M} : Name \hookrightarrow Model \ldots\}$$

Updates of the environment are written with the $\oplus$ operator:

$$(f \oplus (x \mapsto y))(z) = \begin{cases} y & z = x \\ f(z) & z \neq x \end{cases}$$

The operational semantics are given in the form of structural operational semantics. One simple example for such a rule would be the statement composition rule:

COMP

$$\frac{\langle (\mathcal{M}, \mathcal{E}, \Gamma)\ \ s\ \ \rightarrow (\mathcal{M}', \mathcal{E}', \Gamma') \qquad \langle (\mathcal{M}', \mathcal{E}', \Gamma')\ \ S\ \ \rightarrow (\mathcal{M}'', \mathcal{E}'', \Gamma'')}{\langle (\mathcal{M}, \mathcal{E}, \Gamma)\ \ s\ ;\ S\ \rangle \rightarrow (\mathcal{M}'', \mathcal{E}'', \Gamma'')}$$

This rule reads as follows: To evaluate a compositional statement $s\ ;\ S$, first s is evaluated. Afterwards S is evaluated in the resulting state.

3.1 Model Collection

The first rule of the operational semantics handles model collection. Essentially, every model in a program is loaded into the environment.

MODEL

$$\frac{}{(\langle \mathcal{M}, \mathcal{E}, \Gamma \rangle,\ \texttt{model}\ x \ldots \texttt{end}\ \ ) \rightarrow \langle \mathcal{M} \oplus (x \mapsto s), \mathcal{E}', \Gamma' \rangle}$$

Note that this just creates an environment $\mathcal{M}$, that can easily be calculated statically. This is an important property for separate compilation. In fact, any decent compiler will probably generate $\mathcal{M}$ as part of it's type-checking process.

3.2 Model Instantiation

Model instantiation is a recursive procedure. Every submodel (including variables) and equations are handled. During this procedure. The model-name is passed through (and attached as a prefix to subsequent sub-models).

INSTMODEL

$$\frac{\begin{array}{c} \mathcal{M}(t) = \ \texttt{model}\ N\ V\ E\ \texttt{end} \\ V = \ \texttt{var}\ x_1\texttt{:}t_1\ \ \ldots\ \ \texttt{var}\ x_n\texttt{:}t_n \\ E = \ \texttt{equation}\ e_1 B_1\ \ \ldots\ \ \texttt{equation}\ e_m B_m \\ V_{new} = \{\ \texttt{var}\ x.\ x_j\texttt{:}\ t_j\ \ | j = 1 \ldots n\} \\ E_{new} = \{\ \texttt{equation}\ x.\ e_i\ app(B_i, x)\ \ | i = 1 \ldots m\} \\ \langle (\mathcal{M}, \mathcal{E}, \Gamma), V_{new}\ E_{new} \rangle \rightarrow \langle \mathcal{M}', \mathcal{E}', \Gamma' \rangle \end{array}}{\langle (\mathcal{M}, \mathcal{E}, \Gamma),\ \ \texttt{var}\ x\texttt{:}t\ \rangle \rightarrow (\mathcal{M}', \mathcal{E}', \Gamma')}$$

This rule uses the *app* function, which basically renames all occurring variables in a set of blocks according to the given instance name:

$$app(\{\ \ n_i := e_i\ \ \}, x) = \{\ \ x.\ n_i := \ \ prefix(e_i, x)\}$$

We assume that *prefix* renames all variables in an expression accordingly. Thus the expression of every block can be seen as a function over the unknowns occurring in it.

The recursive instantiation process stops at `real` variables (which is simply a built-in type to denote unknowns in the model). Those variables are stored in the environment. They are identified with the (arbitrarily chosen) initial value 0.

LOADVAR

$$\frac{}{\langle (\mathcal{M}, \mathcal{E}, \Gamma),\ \ \texttt{var}\ x\texttt{:real}\ \ \rangle \rightarrow (\mathcal{M}, \mathcal{E}, \Gamma \oplus (x \mapsto 0)}$$

Equations are also loaded into the environment.

$$\textsc{LoadEq}\quad \frac{E = \texttt{ equation}\, x\ B}{\langle(\mathcal{M},\mathcal{E},\Gamma),E\rangle \rightarrow (\mathcal{M},\mathcal{E}\oplus(x\mapsto E),\Gamma)}$$

3.3 Causalization

With those simple rules in place, we can now define the process of causalization. Causalization is the process of converting acausal (algebraic) equations into an assignment that solves the equation.

The basic idea of that process is well known from school books. For instance if one would want to solve the equation

$$x + 3 = y - 5$$

for the variable x, knowing y, the corresponding assignment would be

$$x := y - 8$$

Computing such an expression is naturally much faster than solving the system of equations with an iterative method. The hierarchic structure of object oriented languages like Modelica makes it very likely that for many of the variables of a system, such an assignment can be found. So causalization is a very important aspect of any Compiler for such a language.

In the following we will formally define the process of causalization. Note that the general idea is not new (see, e.g. [6]). The most famous algorithm used in practice is probably that presented by Tarjan [13]. The novelty in our approach is that we do not need any symbolic information for the process (recap: we did not even define a language for such information). Therefore we can easily define causalization as part of the operational semantics being executed *after* compilation.

The first element of the process is the *occurrence* relation occ established between equations and variables: Informally, if a variable is used inside an equation both are in the occurrence relation.

$$(\ \texttt{equation id}\ B\ , x) \in occ \Leftrightarrow\ \ x := E\quad \in B$$

That relation is then used to build the *occurrence Graph* $G_{occ} = (E_{occ}, V_{occ})$ for a state $(\mathcal{M},\mathcal{E},\Gamma)$:

$$V_{occ}(\mathcal{M},\mathcal{E},\Gamma) = \text{dom}(\mathcal{E}) \cup \text{dom}(\Gamma)$$
$$E_{occ}(\mathcal{M},\mathcal{E},\Gamma) = \{(v,e) | \mathcal{E}(e)\ occ\ v\}$$

Note, that G_{occ} is bipartite (if $\text{dom}(\mathcal{E}) \cap \text{dom}(\Gamma) = \emptyset$, which we assume). G_{occ} can be used to check some properties for solvability, e.g. the existence of at least one equation for each variable. But since we do not deal with error handling for now, we assume only solvable systems of equations as input.

In a next step, the occurrence Graph is transformed into the *dependency graph*. For this task, we search a perfect match $M(\mathcal{M},\mathcal{E},\Gamma)$ in the *transposed* set of edges of G_{occ}. (In the following we will omit the parameter $(\mathcal{M},\mathcal{E},\Gamma)$ for brevity)

$$M \subseteq E_{occ}^T\ s.t. \qquad \forall v \in \text{dom}(\Gamma)\ \exists! e\ s.t.\ (e,v) \in M$$
$$\wedge \forall e \in \text{dom}(\mathcal{E})\ \exists! v\ s.t.\ (e,v) \in M$$

The existence of such a perfect match is again considered a property of the given system of equations (if it does not exist, there is a under- or over-determined subsystem).

The (directed) dependency graph G_{dep} is defined as follows:

$$G_{dep} = (V_{dep}, E_{dep}) = (V_{occ}, E_{occ} \cup M)$$

Because the edges in M are drawn from equations to unknowns, we can state an important property: Every name in V_{occ} is part of a strongly connected component with exactly 2 nodes.

Lemma 1. *Every Node in V_{occ} is part of a 2-element strongly connected component.*

$$x \in V_{occ} \Rightarrow \exists! y \in V_{occ} s.t. \{(x,y),(y,x)\} \subseteq E_{dep}$$

Proof.

$$\begin{aligned} x \in V_{occ} \Rightarrow \quad & x \in \text{dom}(\Gamma) \vee x \in \text{dom}(\mathcal{E}) \\ \Rightarrow \quad & \exists!(e,x) \in M \vee \exists!(x,v) \in M \\ \Rightarrow \quad & \exists(x,e) \in E_{occ} \vee \exists(v,x) \in E_{occ} \\ \Rightarrow \quad & \exists! y \in V_{occ} : \{(x,y),(y,x)\} \subseteq E_{dep} \end{aligned}$$

□

Such a strongly connected component (consisting of the name of an equation and the name of an unknown) can directly be identified with a block:

$$\begin{aligned} \mathcal{E}(e) = \quad & \texttt{equation}\, e \ldots v := Ex \ldots \\ \wedge \quad & \{(v,e),(e,v)\} \in E_{dep} \\ \Rightarrow \quad & block(e) = block(v) = \ \ v := Ex \end{aligned}$$

Note that $block$ yields exactly one result for every name, if the variable names of all blocks of an equation are pairwise unequal (another static property of the input program, that we (yet) do not care about). The fact that there is *at least one* block for every such pair in the graph follows directly from the construction of E_{occ}. Since we are interested in actually solving a system of equations, we need another kind of graph, the *calculation order graph*:

$$G_{calc} = (\{block(v) \mid v \in V_{occ}\}, \leq_{calc} \subseteq (Block \times Block))$$

Here $\leq_{calc}$ can be seen informally as a dependency between two blocks: $b_1 \leq_{calc} b_2$ means that b_2 depends on the variable calculated by b_1. :

$$\frac{(v,e) \in E_{occ} \qquad b_1 = block(v) \qquad b_2 = block(e) \qquad b_1 \neq b_2}{b_1 \leq_{calc} b_2}$$

For obvious reasons, such a dependency relation needs to be transitive:

$$\frac{b_1 \leq_{calc} b_2 \qquad b_2 \leq_{calc} b_3}{b_1 \leq_{calc} b_2}$$

It might occurr that $b_1 \leq_{calc} b_2 \wedge b_2 \leq_{calc} b_1$. For that reason we introduce $=_{calc}$ to express such a cyclic dependency:

$$\frac{b_1 \leq_{calc} b_2 \qquad b_2 \leq_{calc} b_1}{b_1 =_{calc} b_2}$$

Because $<_{calc}$ is transitive, we can easily lift it to a relation over sets of cyclic dependent blocks $\prec_{calc}$:

$$\begin{gathered} Q_{calc} = V_{calc}/_{=_{calc}} \\ \prec_{calc} \subseteq Q_{calc} \times Q_{calc} \\ B_1 \prec_{calc} B_2 \Leftrightarrow (b_1, b_2) \in B_1 \times B_2 \Rightarrow b_1 <_{calc} b_2 \end{gathered}$$

Lemma 2. *$T_{calc} = (Q_{calc}, \prec_{calc})$ forms a forest.*

Proof. Assume that $\exists B_1, B_2 \in Q_{calc}\ s.t.\ B_1 \prec_{calc} B_2 \wedge B_2 \prec_{calc} B_1 \wedge B_1 \neq B_2$

From the definition, it follows, that $(b_1, b_2) \in B_1 \times B_2 \Rightarrow b_1 \leq_{calc} b_2 \wedge b_2 \leq_{calc} b_1 \Rightarrow b_1 =_{calc} b_2$

But since $B_1, B_2 \in Q_{calc} = V_{calc}/_{=_{calc}}$, by the definition of the quotient set, it follows that $b_1 =_{calc} b_2 \Rightarrow b_2 \in B_1 \Rightarrow B_1 = B_2$. ↯

At that point we can now define the behavior of the `causalize` statement.

CAUSALIZE

$$\frac{\begin{gathered} T_{calc}(\mathcal{M}, \mathcal{E}, \Gamma) = (\{B_1 \dots B_n\}, \prec_{calc}) \\ B_1 \prec_{calc} \dots \prec_{calc} B_n \\ \langle (\mathcal{M}, \mathcal{E}, \Gamma),\ \texttt{solve}\ B_1;\ \dots \texttt{solve}\ B_n\ \rangle \rightarrow (\mathcal{M}', \mathcal{E}', \Gamma') \end{gathered}}{\langle (\mathcal{M}, \mathcal{E}, \Gamma),\ \texttt{causalize}\ \rangle \rightarrow (\mathcal{M}', \mathcal{E}', \Gamma')}$$

Causalization is now built into the operational semantics: First create the calculation tree and then simply walk it in order. More prominently, no symbolic information about expressions was used during this process. Yet we still have to discuss the properties of the `solve` statement.

3.4 Block evaluation

The only semantics left to discuss concern the `solve` statement. Up until now everything in TinyModelica's semantics did not yield actual calculation of simulation data. To fill this gap, we assume that the semantics of the (still undefined) expression language is given via two evaluation functions:

$$\begin{aligned} \mathcal{A}: &\quad State \times Expr \rightarrow \mathbb{R} \\ \mathcal{J}: &\quad State \times Name \times Expr \rightarrow \mathbb{R} \end{aligned}$$

The first rule for solving an equation is rather simple: If the set of blocks to solve has only one element, the block is interpreted as an assignment.

SOLVE SINGLE

$$\frac{B = \{\ x := E\ \} \qquad \Gamma' = \Gamma \oplus (x \mapsto \mathcal{A}((\mathcal{M}, \mathcal{E}, \Gamma), E)}{\langle (\mathcal{M}, \mathcal{E}, \Gamma),\ \texttt{solve}\ (B)\ \rangle \rightarrow (\mathcal{M}, \mathcal{E}, \Gamma')}$$

The actual new value for the variable x depends on the evaluation function. Complete DAEs can be solved by using builtin constructs for numerical integration and time.

SOLVE MULTI

$$\frac{\begin{gathered} B = \{\ x_1 := E_1\ , \dots,\ x_n := E_n\ \} \\ F \in \mathbb{R}^n\ F_i = \mathcal{A}((\mathcal{M}, \mathcal{E}, \Gamma), E_i) - \Gamma(x_i) \\ J \in \mathbb{R}^{n \times n}\ J_{(i,j)} = \mathcal{J}((\mathcal{M}, \mathcal{E}, \Gamma), x_i, E_j), i \neq j \\ J_{(i,i)} = -1 \qquad \Delta x \in \mathbb{R}^n : J\Delta x = -F \\ \Gamma' = \Gamma \oplus (x_1 \mapsto \Gamma(x_1) + \Delta x_1) \dots \oplus (x_n \mapsto \Gamma(x_n) + \Delta x_n) \end{gathered}}{\langle (\mathcal{M}, \mathcal{E}, \Gamma),\ \texttt{solve}\ (B)\ \rangle \rightarrow (\mathcal{M}, \mathcal{E}, \Gamma')}$$

This (last) rule of the operational semantics demands the solution of a linear systems of equations for Δx. Informally, by using Newtons method (see, e.g. [10]), we compute a fixed point for the function $F(\bar{x}) = \mathcal{A}(\bar{E}) - \Gamma(\bar{x})$.

Note, that this interpretation assumes a unique order on the variables in the system. Such an order can easily be computed (we did not include it in the semantics to keep the rules as small as possible). The numerical properties of this evaluation rule completely depend on $\mathcal{A}$ and $\mathcal{J}$.

4. Compiling systems of linear equations

In the following section we will discuss, how TinyModelica can be used to solve systems of linear equations. Especially we will show, how the block expressions must evaluate to yield a valid solution.

4.1 Block triangular form

Consider a linear system of equations: $A\bar{x} = 0$ with $\bar{x} \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n \times n}$. In the following, we assume that x_i is likewise the expression yielding the value of an unknown in the compiled model as a component of the solution vector $\bar{x}$.

For every equation $\bar{a}_i\bar{x} = 0$, we can reduce the sum to the non-zero components $\bar{a}_i\bar{x} = \bar{\alpha}_i\bar{z} = 0$. Now we can create a solution vector $b(i,j)$ for every equation $\bar{a}_i$ and every variable x_j:

$$b(i,j) = (\frac{\alpha_{i,1}}{-\alpha_{i,j}} \dots \frac{\alpha_{i,j-1}}{-\alpha_{i,j}}, 0, \frac{\alpha_{i,j+1}}{-\alpha_{i,j}} \dots)$$

Naturally, we compile every equation into all such blocks. If we evaluate such a block according to SOLVE SINGLE, we have a solution for $\bar{a}_i$. Note, that such a vector directly delivers the partial derivatives. Therefore for any set of blocks $\{(b(i_1, j_1), \dots b(i_n, j_n)\}$, we can calculate F and J as follows:

$$J = F = \begin{bmatrix} b(i_1, j_1) - e_{j_1} \\ \vdots \\ b(i_n, j_n) - e_{j_n} \end{bmatrix}$$

We know that, after evaluating SOLVE MULTI, $J\Delta\bar{x} = -F\bar{x}$. It follows directly, that $F(\Delta\bar{x} + \bar{x}) = 0$. Therefore, after one evaluation, Γ contains a valid solution to $\{\bar{a}_{i_1}, \ldots, \bar{a}_{i_n}\}$.

In the case of multiple sets of blocks to be solved (which should be considered the usual case), from [6] it is known that the calculation order returned by the causalization $B_1 \prec_{calc} \ldots \prec_{calc} B_n$ can be represented as a block matrix in lower triangular form:

$$\left[\begin{array}{cccc|c} J_1 & 0 & \cdots & 0 & 0 \\ A_{21} & J_2 & \cdots & 0 & 0 \\ \vdots & & \ddots & \vdots & \\ A_{n1} & A_{n2} & \cdots & J_n & 0 \end{array}\right]$$

Notably, here $J_i \neq F_i$, since the F_i also take the A_{nm} as inputs:

$$F_i = \left[\begin{array}{cccc} A_{i1} & \cdots A_{i(i-1)} & J_i \end{array}\right]$$

Therefore after every evaluation of SOLVE MULTI, the following equation holds:

$$J\Delta\bar{x} = -F\left[\begin{array}{c} \bar{y} \\ \bar{x} \end{array}\right] = -\left[\begin{array}{cccc} A_{i1} & \cdots A_{i(i-1)} & J_i \end{array}\right]\left[\begin{array}{c} \bar{y} \\ \bar{x} \end{array}\right]$$

This (again), leads to the fact that:

$$\left[\begin{array}{ccc} A_{i1} & \cdots & A_{i(i-1)} \end{array}\right]\bar{y} + J(\Delta\bar{x} + \bar{x}) = 0$$

In other words: applying SOLVE MULTI to systems of linear equations is not much different from applying Gaussian elimination to a block matrix. This shows that this semantic rule can be applied efficiently to systems of linear as well as nonlinear equations.

5. Nonlinear equations

Although we only described systems of linear equations, the proposed method naturally works in the nonlinear case: Instead of directly calculating the block expressions one has to use a decent computer algebra system to generate inverse functions, to generate the block representation of a single equation. It may of course occur that even a computer algebra system does not find a direct solution (or multiple solutions exist). In such a case a numerical method could be used to emulate a direct solution, as long as $\mathcal{J}$ delivers the partial derivatives (which in turn could be derived e.g. by automatic differentiation).

For the operational semantics, the existence of nonlinear equations is no big problem: After a single step of a Newton iteration, in SOLVE MULTI, we need to check the validity of the solution:

SOLVE MULTI RECURSIVE

$$\frac{\begin{array}{c} B = \{\ x_1 := E_1\ , \ldots,\ x_n := E_n\ \} \\ F \in \mathbb{R}^n\ F_i = \mathcal{A}((\mathcal{M}, \mathcal{E}, \Gamma), E_i) - \Gamma(x_i) \\ J \in \mathbb{R}^{n \times n}\ J_{(i,j)} = \mathcal{J}((\mathcal{M}, \mathcal{E}, \Gamma), x_i, E_j), i \neq j \\ J_{(i,i)} = -1 \qquad \Delta x \in \mathbb{R}^n : J\Delta x = -F \qquad \mathbf{F} \approx \mathbf{0} \\ \Gamma' = \Gamma \oplus (x_1 \mapsto \Gamma(x_1) + \Delta x_1) \ldots \oplus (x_n \mapsto \Gamma(x_n) + \Delta x_n) \end{array}}{\langle (\mathcal{M}, \mathcal{E}, \Gamma),\ \texttt{solve (}B\texttt{)}\ \rangle \rightarrow (\mathcal{M}, \mathcal{E}, \Gamma)}$$

SOLVE MULTI STOP

$$\frac{\begin{array}{c} B = \{\ x_1 := E_1\ , \ldots,\ x_n := E_n\ \} \\ F \in \mathbb{R}^n\ F_i = \mathcal{A}((\mathcal{M}, \mathcal{E}, \Gamma), E_i) - \Gamma(x_i) \\ \mathbf{F} \sim \mathbf{0} \end{array}}{\langle (\mathcal{M}, \mathcal{E}, \Gamma),\ \texttt{solve (}B\texttt{)}\ \rangle \rightarrow (\mathcal{M}, \mathcal{E}, \Gamma)}$$

A more problematic case is the distance of the initial values from a solution: The method we use, might diverge, if the initial solution is too far from the next one.

Since this paper is not about numerics, we ignore those problems for now and focus on systems of linear equations, where an exact solution can be computed.

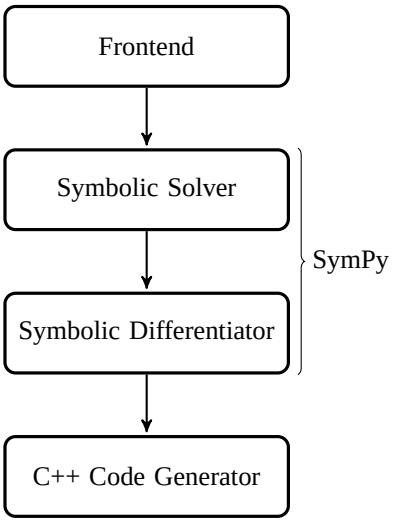


Figure 1. Prototype Compiler Architecture

6. Prototype Implementation

We implemented a TinyModelica runtime system and a Modelica to TinyModelica compiler as a prototype. The runtime is based on the C++ boost libraries [1]. Currently it features a simple explicit Euler integration, Newton's method and an implementation of TinyModelica's causalization and solve statements.

The Compiler is written in Java and translates Modelica source (actually a subset of Modelica) into C++ source files. Those output files contain classes and functions, that implement the model and equation instantiation part of the TinyModelica semantics. Models can be compiled into linkable objects and (linked against the runtime) be used in different contexts.

For the computer algebra part of the compiler we chose SymPy [2], a computer algebra system written in Python

[1] www.boost.org

[2] www.sympy.org

(Figure 1). This construction allows also nonlinear systems (although we did not yet implement the iteration semantics in the runtime by now).

6.1 Example

We tested our implementation with a very simple example for a linear system of equations:

```
model Test1
      Real a,b,c;
      equation
      a + 3*b - 2*c = time;
      2 * a + 2*b + 6*c = 4;
end Test1;

model Test2
      Test1 test1;
      equation
      test1.a + test1.b - test1.c = 0;
end Test2;

model MainTest
    Test2[1000] test2;
end MainTest;
```

We used Modelica as a concrete syntax for our TinyModelica test implementation. The translation of the above shown models into TinyModelica's abstract syntax should be pretty obvious (with the notable exception of the array definition used, our implementation was extended at that point). Below is the definition of `Test2` in TinyModelica syntax as an example:

```
model Test2
      var test1 : Test1
      equation e1
      test1.a := test1.c - test1.b
      test1.b := test1.c - test1.a
      test1.c := test1.a + test1.b
end
```

All three models were compiled separately into one C++ header and source file. By varying the size of the `test2` array, we can demonstrate the value of separate compilation. As a reference, we compiled the model also with OpenModelica 1.7 [7].

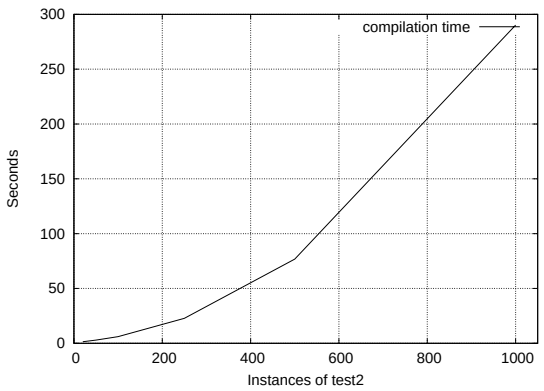


Figure 2. Compilation times in OpenModelica

The process of compiling a model with OpenModelica can be described by two phases: A Modelica front-end, which parses the source code, flattens the model and does all kind of symbolic manipulations. The output of this front-end is some C++ program that is able to solve the generated system of equations. That C++ program is then compiled and linked by a appropriate compiler. We measured the time of the back-end (the GNU C++ compiler in our case), since the front-end was faster by some orders of magnitude.

Figure 2 shows, how the compilation time for the example model grows byond unbearable limits. As mentioned this does *not* imply some performance problem with the omc, since we measured the time that is spent by the compilation of the generated C++ source code. The only problem is the sheer *size* of the generated code (the source file grew to ca. 3.2MB). The time of re-compilation with our prototype remained constantly around one second (which is mainly caused by the heavy use of C++ templates, using pure C could lead to even faster compilation times). In turn our front-end also took around a second which was caused by general slow startup of Java applications, the loading of SymPy from an embedded python interpreter and the prototype style implementation. More important than the actual numbers of our implementation is the fact that they remained *constant* regardless how many instances of `test2` models were instantiated.

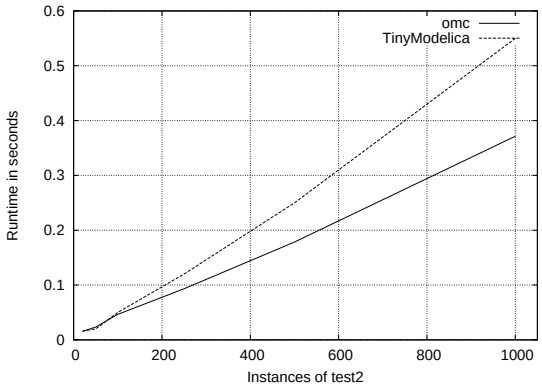


Figure 3. Performance comparison, OpenModelica - TinyModelica

To demonstrate the scalability of our approach, we measured the runtime of a one-second simulation of the test model. Note, that due to the absence of differential equations, the implemented integration method could not affect the runtime behavior. Figure 3 shows our performance comparison. Unsurprisingly, OpenModelica simulated this trivial model very fast, even for many instances. As one could expect the simulation time grows lineary. Although our prototype is not nearly as advanced as the OpenModelica implementation, it shows roughly the same scaling.

7. Conclusion

Separate Compilation of acausal equations is not only possible, but can also be implemented in an efficient equation solver. Although the worst case space consumption grows

from $\mathcal{O}(n)$ to $\mathcal{O}(n^2)$, in practice this compilation scheme saves space.

Obviously compilation time is also drastically improved: Generating all possible variations of a single equation turns out to much more efficient as soon as several instances of a model are active during simulation.

Although it might seem to be a rather technical issue, the development of efficient separate compilation techniques might have a sever impact on upcoming modeling languages: The operational semantics of languages like Modelica are currently built into the tools as interpreters. This does not only limit the size of the compiled models. It also impacts the runtime features a simulation can offer: Features like model structural dynamics [15, 12] are hard if not impossible to implement with current compiler techniques. Additionally, with our method, model libraries can be distributed in a *compiled* form, without any need for encryption methods.

8. Related Work

A different approach to separate compilation of Modelica-like languages is mentioned in [11]. Here it is proposed to extend the language itself to allow for separate compilation. As we have shown, it is not necessary to change anything at the language level as long as a different approach to model instantiation is chosen.

Defining flattening as operational semantics of a modeling language is not a new idea. It has been developed i.e. in [4] and [3]. Although to our knowledge there has been no further research into causalization or index reduction in that context.

The negative effects of a global compilation model for Modelica have been addressed in [14]. Instead of compiling every module separately, another compilation stage is proposed to detect good candidates of code re-usage. This approach clearly has the advantage of generating very good (i.e. fast) code, but seems rather complex to implement. In contrast to our proposal it is an optimization of the state-of-the-art compilation model.

Instead of relying on a classical compilation scheme [8] presents an approach to *completely* move the handling of a system of equations (i.e. symbolic processing, compilation etc.) into the runtime of a host system (Haskell in that case). This idea goes much further than our proposal. The idea of just-in-time compilation for hybrid systems is very appealing (discrete values can become constants and the code can be even more efficient than with ahead-of-time compilation). In form of specialization, a just-in-time compiler like LLVM might as well be useful in our case. Yet there is no reason to postpone the actual symbolic manipulation and code generation into the runtime, as we have shown.

9. Future Work

With TinyModelica we have shown that it is possible to implement a separate compilation scheme with efficient code generation for algebraic equations. Though, some points are still open research questions:

- Index reduction is very important for the solution of most practical models. Currently it is unclear how a compiler might generate code prior to knowing the index of the global system. Usually a index reduction demands a symbolic differentiation up to to the index of the system. Automatic differentiation (see e.g. [2]) might be a solution here.
- The performance difference observed between our prototype and OpenModelica may be caused by the runtime causalization as well as some inefficiencies (e.g. additional function calls in the code). More research here might yield a faster simulation.
- Causalization might be moved into a separate linking phase. This could as well speed up the simulation start as allow earlier error detection (e.g. singular or underdetermined systems of equations).

References

[1] The Modelica Association. Modelica - a unified object-oriented language for physical systems modeling, 2010.

[2] Willi Braun, Lennart Ochel, and Bernhard Bachmann. Symbolically derived jacobians using automatic differentiation - enhancement of the openmodelica compiler.

[3] David Broman. Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments. Licentiate thesis. Thesis No 1337. Department of Computer and Information Science, Linköping University, December 2007.

[4] David Broman and Peter Fritzson. Higher-order acausal models. In *2nd International Workshop on Equation-Based Object-Oriented Languages and Tools,2008*, pages 59–. Linköping University Electronic Press, 2008.

[5] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 266–277, New York, NY, USA, 1997. ACM.

[6] François E. Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer, 1 edition, March 2006.

[7] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The openmodelica modeling, simulation, and development environment. In *Proceedings of the 46th Conference on Simulation and Modeling*, pages 83–90, 2005.

[8] George Giorgidze and Henrik Nilsson. Mixed-level embedding and jit compilation for an iteratively staged dsl. In *Proceedings of the 19th international conference on Functional and constraint logic programming*, WFLP'10, pages 48–65, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] Christoph Höger, Florian Lorenzen, and Peter Pepper. Notes on the separate compilation of modelica. In Peter Fritzson, Edward Lee, François E. Cellier, and David Broman, editors, *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pages 43–51. Linköping University Electronic Press, 2010.

[10] C. T. Kelley. *Solving Nonlinear Equations with Newton's Method*. SIAM, Philadelphia, 2003.

[11] Ramine Nikoukhah. Extensions to modelica for efficient code generation and separate compilation. In *Proceed-*

ings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools, Linköping Electronic Conference Proceedings, page 49â€“59. Linköping University Electronic Press, Linköpings universitet, 2007.

[12] Christoph Nytsch-Geusen and Thilo Ernst. Mosilab: Development of a modelica based generic simulation tool supporting model structural dynamics. In Gerhard Schmitz, editor, *Proceedings of the 4th International Modelica Conference, Hamburg, March 7-8, 2005*, pages 527–535. TU Hamburg-Harburg, 2005.

[13] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *Siam Journal on Computing*, 1:146–160.

[14] Dirk Zimmer. Module-preserving compilation of modelica models. In *Proceedings of the 7th International Modelica Conference, Como, Italy, 20-22 September 2009*, Linköping Electronic Conference Proceedings, pages 880–889. Linköping University Electronic Press, Linköpings universitet, 2009.

[15] Dirk Zimmer. *Equation-based Modeling of Variable-structure Systems*. PhD thesis, ETH Zürich, 2010.