EOOLT 2008

Proceedings of the

2nd International Workshop on Equation-Based Object-Oriented Languages and Tools

Paphos, Cyprus, July 8, 2008,
in conjunction with ECOOP

Editors

Peter Fritzson, François Cellier, David Broman

Copyright

Linköping Electronic Conference Proceedings, No. 29
Linköping University Electronic Press
Linköping, Sweden, 2008

ISSN 1650-3740 (online, Linköping University Electronic Press)
http://www.ep.liu.se/ecp/029/
ISSN 1650-3686 (print, Linköping University Electronic Press)

Table of Contents

Preface

Computer aided modeling and simulation of complex systems, using components from multiple application domains, such as electrical, mechanical, hydraulic, control, etc., have in recent years witnessed a significant growth of interest. In the last decade, novel equation-based object-oriented (EOO) modeling languages, (e.g. Modelica, gPROMS, and VHDL-AMS) based on a-causal modeling using equations have appeared. Using such languages, it has become possible 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 areas (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.

Despite the fact that this is a new not very established workshop series, there was a good response to the call-for-papers. Thirteen papers were accepted to the workshop program out of fifteen submissions. All papers were subject to rather detailed reviews by the program committee, on the average four reviews per paper. The workshop program started with a welcome and introduction to the area of equation-based object-oriented languages, followed by the keynote talk by Hans Vangheluwe and paper presentations. Discussion sessions were held after presentations of each set of related papers.

On behalf of the program committee, the Program Chairmen would like to thank all those who submitted papers to EOOLT'2008. Special thanks go to Loucas Louca who helped with the local on-site organization of the workshop. Many thanks go also to the program committee for reviewing the papers. The venue for EOOLT'2008 was Paphos, Cyprus, in conjunction with the ECOOP'2008 conference.

Linköping, July 2008

Peter Fritzson
François Cellier
David Broman

Program Chairmen

Peter Fritzson, Chair	Linköping University, Linköping, Sweden
François Cellier, Co-Chair	ETH, Zurich, Switzerland
David Broman, Co-Chair	Linköping University, Linköping, Sweden

Program Committee

Peter Fritzson	Linköping University, Linköping, Sweden
François Cellier	ETH Zurich, Switzerland
David Broman	Linköping University, Linköping, Sweden
Bernhard Bachmann	University of Applied Sciences, Bielefeld, Germany
Bert van Beek	Eindhoven University of Technology, Netherlands
Gilad Bracha	Cadence Design Systems, San Jose, CA, USA
Felix Breitenecker	Technical University of Vienna, Vienna, Austria
Jan Broenink	University of Twente, Netherlands
Peter Bunus	Linköping University, Linköping, Sweden
Ernst Christen	Lynguent, Inc., Portland, OR, USA
Sebastián Dormido	National University for Distance Education, Madrid, Spain
Olaf Enge-Rosenblatt	Fraunhofer Institute for Integrated Circuits, Dresden, Germany
Peter Feiler	SEI, Carnegie-Mellon University, Pittsburg, USA
Stefan Jähnichen	Fraunhofer FIRST and TU Berlin, Berlin, Germany
Petter Krus	Linköping University, Linköping, Sweden
Loucas Louca	University of Cyprus, Nicosia, Cyprus
Jacob Mauss	QTronic GmbH, Berlin, Germany
Pieter Mosterman	MathWorks, Inc., Natick, MA, USA.
Ramine Nikoukhah	INRIA Rocquencourt, Paris, France
Henrik Nilsson	University of Nottingham, Nottingham, United Kingdom
Dionisio de Niz	Carnegie Mellon University, Pittsburgh, USA
Martin Otter	DLR Oberpfaffenhofen, Germany
Chris Paredis	Georgia Institute of Technology, Atlanta, Georgia, USA
César de Prada	University of Valladolid, Valladolid, Spain
Juan José Ramos	Autonomous University of Barcelona, Spain
Peter Schwarz	Fraunhofer Inst. for Integrated Circuits, Dresden, Germany
Paul Strooper	University of Queensland, Brisbane, Australia
Michael Tiller	Emmeskay, Inc., Plymouth, MI, USA
Martin Törngren	KTH, Stockholm, Sweden
Alfonso Urquía	National University for Distance Education, Madrid, Spain
Hans Vangheluwe	McGill University, Montreal, Canada

Workshop Organization

Peter Fritzson	Linköping University, Linköping, Sweden
David Broman	Linköping University, Linköping, Sweden
François Cellier	ETH, Zurich, Switzerland
Loucas Louca	University of Cyprus, Nicosia, Cyprus

Multi-Paradigm Language Engineering and Equation-Based Object-Oriented Languages

(keynote abstract)

Hans Vangheluwe

School of Computer Science, McGill University, Montréal, Canada
Hans.Vangheluwe@mcgill.ca

Abstract

Models are invariably used in Engineering (for design) and Science (for analysis) to precisely describe structure as well as behaviour of systems. Models may have components described in different formalisms, and may span different levels of abstraction. In addition, models are frequently transformed into domains/formalisms where certain questions can be easily answered. We introduce the term "multi-paradigm modelling" to denote the interplay between multi-abstraction modelling, multi-formalism modelling and the modelling of model transformations.

The foundations of multi-paradigm modelling will be presented. It will be shown how all aspects of multi-paradigm modelling can be explicitly (meta-)modeled enabling the efficient synthesis of (possibly domain-specific) multi-paradigm (visual) modelling environments. We have implemented our ideas in the tool AToM^3 (A Tool for Multi-formalism and Meta Modelling) [3].

Over the last decade, Equation-based Object-Oriented Languages (EOOLs) have proven to bring modelling closer to the problem domain, away from the details of numerical simulation of models. Thanks to Object-Oriented structuring and encapsulation constructs, meaningful exchange and re-use of models is greatly enhanced.

Different directions of future research, combining multi-paradigm modelling concepts and techniques will be explored:

1. meta-modelling and model transformation for domain-specific modelling as a layer on top of EOOLs;
2. on the one hand, the use of Triple Graph Grammars (TGGs) to declaratively specify consistency relationships between different models (views). On the other hand, the use of EOOLs to complement Triple Graph Grammars (TGGs) in an attempt to come up with a fully "declarative" description of consistency between models to support co-evolution of models;
3. the use of graph transformation languages describing structural change to modularly "weave in" variable structure into non-dynamic-structure modelling languages.

Keywords Multi-Paradigm Modelling, Meta-Modelling, Model Transformation, Equation-Based Object-Oriented Languages, Consistency, Variable Structure

1. Multi-Paradigm Modelling

In this section, the foundations of Multi-Paradigm Modelling (MPM) are presented starting from the notion of a *modelling language*. This leads quite naturally to the concept of *meta-modelling* as well as to the explicit modelling of *model transformations*.

Models are an *abstraction* of reality. The structure and behaviour of systems we wish to analyze or design can be represented by models. These models, at various *levels of abstraction*, are always described in some *formalism* or *modelling language*. To "model" modelling languages and ultimately synthesize (visual) modelling environments for those languages, we will break down a modelling language into its basic constituents [4]. The two main aspects of a model are its syntax (how it is represented) on the one hand and its semantics (what it means) on the other hand.

The syntax of modelling languages is traditionally partitioned into *concrete syntax* and *abstract syntax*. In textual languages for example, the concrete syntax is made up of sequences of *characters* taken from an *alphabet*. These characters are typically grouped into *words* or *tokens*. Certain sequences of words or *sentences* are considered valid (i.e., belong to the language). The (possibly infinite) *set* of all valid sentences is said to make up the language.

For practical reasons, models are often stripped of irrelevant concrete syntax information during syntax checking. This results in an "abstract" representation which captures the "essence" of the model. This is called the *abstract syntax*. Obviously, a single abstract syntax may be represented using multiple concrete syntaxes. In programming language compilers, abstract syntax of models (due to the nature of programs) is typically represented in *Abstract*

2nd International Workshop on Equation-Based Object-Oriented Languages and Tools. July 8, 2008, Paphos, Cyprus.

Syntax Trees (ASTs). In the context of general modelling, where models are often graph-like, this representation can be generalized to *Abstract Syntax Graphs* (ASGs).

Once the syntactic correctness of a model has been established, its meaning must be specified. This meaning must be *unique* and *precise*. Meaning can be expressed by specifying a *semantic mapping function* which maps every model in a language onto an element in a *semantic domain*. For example, the meaning of a Causal Block Diagram (e.g., a Simulink diagram) can be specified by mapping onto an Ordinary Differential Equation. For practical reasons, semantic mapping is usually applied to the abstract rather than to the concrete syntax of a model. Note that the semantic domain is a modelling language in its own right which needs to be properly modelled (and so on, recursively). In practice, the semantic mapping function maps abstract syntax onto abstract syntax.

To continue the introduction of meta-modelling and model transformation concepts, languages will explictly be represented as (possibly infinite) sets as shown in Figure 1. In the figure, insideness denotes the sub-set relationship. The dots represent model which are elements of the encompassing set(s).

As one can always, at some level of abstraction, represent a model as a graph structure, all models are shown as elements of the set of all graphs Graph. Though this restriction is not necessary, it is commonly used as it allows for the design, implementation and bootstrapping of (meta-)modelling environments. As such, any modelling language becomes a (possibly infinite) set of graphs. In the bottom centre of Figure 1 is the abstract syntax set A. It is a set of models stripped of their concrete syntax.

1.1 Meta-models

Meta-modelling is a heavily over-used term. Here, we will use it to denote the explicit description (in the form of a finite model in an appropriate meta-modelling language) of the Abstract Syntax set. Often, meta-modelling also covers a model of the concrete syntax. Semantics is however not covered. In the figure, the Abstract Syntax set is described by means of its meta-model. On the one hand, a meta-model can be used to *check* whether a general model (a graph) *belongs to* the Abstract Syntax set. On the other hand, one could, at least in principle, use a meta-model to *generate* all elements of the language.

1.2 Concrete Syntax

A model in the Abstract Syntax set (see Figure 1) needs at least one concrete syntax. This implies that a concrete syntax mapping function κ is needed. κ maps an abstract syntax graph onto a concrete syntax model. Such a model could be textual (e.g., an element of the set of all Strings), or visual (e.g., an element of the set of all the 2D vector drawings). Note that the set of concrete models can be modelled in its own right.

1.3 Meaning

Finally, a model m in the Abstract Syntax set (see Figure 1) needs a unique and precise meaning. As previously discussed, this is achieved by providing a Semantic Domain and a semantic mapping function M. Rule-based Graph Transformation formalisms are often used to specify semantic mapping functions in particular and model transformations in general. Complex behaviour can be expressed very intuitively with a few graphical rules. Furthermore, Graph Grammar models can be analyzed and executed.

1.4 Formalism Transformation

In an attempt to mimimize accidental complexity [2], modellers often transform a model in one formalism to model in another formalism, retaining salient properties.

2. Domain-specific Modelling

Domain- and formalism-specific modelling have the potential to greatly improve productivity as they [5].

- match the user's mental model of the problem domain;
- maximally constrain the user (to the problem at hand, through the checking of domain constraints) making the language easier to learn and avoiding modelling errors "by construction";
- separate the domain-expert's work from analysis and transformation expert's work.
- are able to exploit features inherent to a specific domain or formalism. This will for example enable specific analysis techniques or the synthesis of efficient (simulation) code exploiting features of the specific domain.

The time required to construct domain/formalism-specific modelling and simulation environments can however be prohibitive. Thus, rather than using such specific environments, generic environments are typically used. Those are necessarily a compromise. The above language engineering techniques allow for rapid development of domain-specific (visual) modelling environments with little effort if mapping onto a semantic domain (such as an EOOL) is done.

3. Consistency/Co-evolution of Model Views

In the development of complex systems, multiple views on the system-to-be-built are often used. These views typically consist of models in different formalisms. Different views usually pertain to various partial aspects of the overall system. In a multi-view approach, individual views are (mostly) less complex than a single model describing all aspects of the system. As such, multi-view modelling, like modular, hierarchical modelling, simplifies model development. Most importantly, it becomes possible for individual experts on different aspects of a design to work in isolation on individual views without being encumbered with other aspects. These individual experts can work mostly *concurrently*, thereby considerably speeding up the development process. This realization was the core of Concurrent Engineering. This approach does however have a cost associated with it. As individual view models evolve, inconsistencies between different views are often introduced. Ensuring consistency between different views requires periodic con-

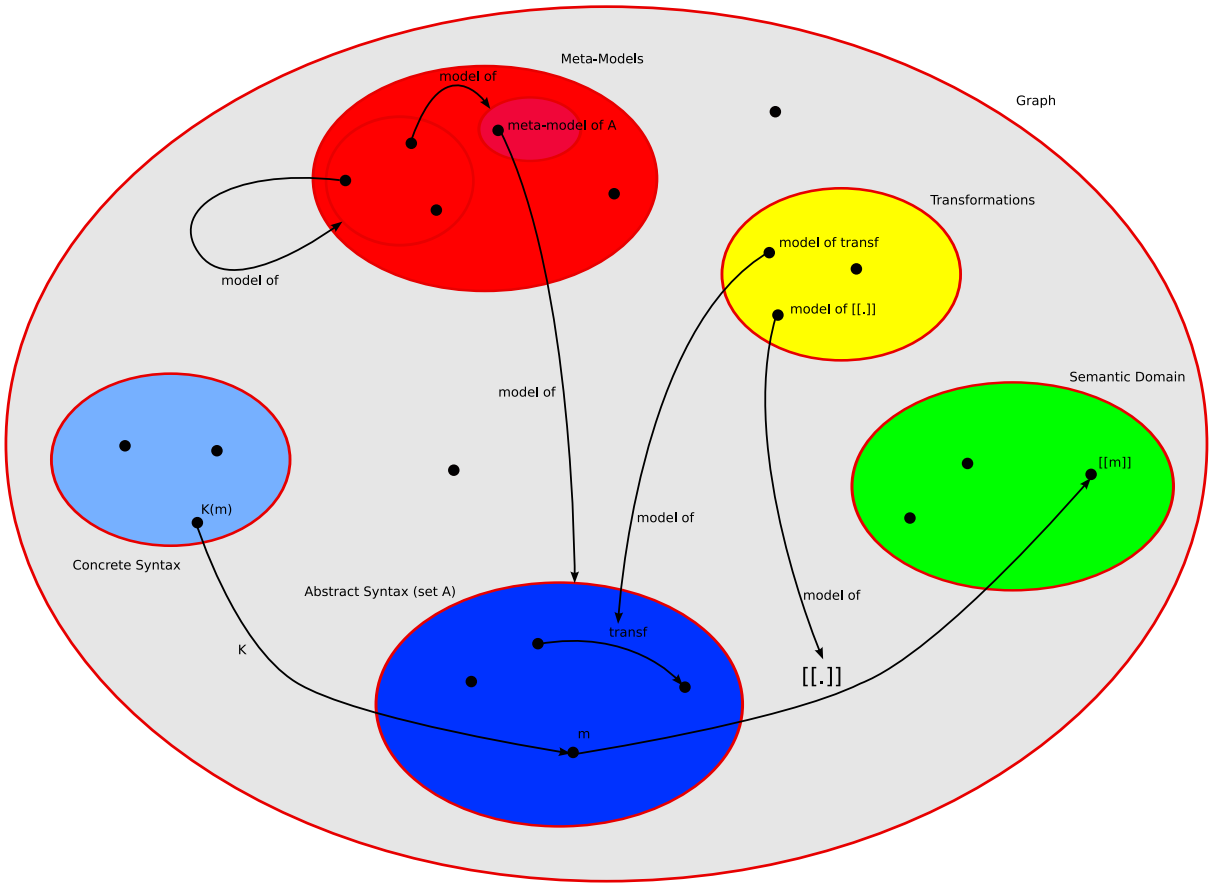


Figure 1. Modelling Languages as Sets

certed efforts from the model designers involved. In general, the detection of inconsistencies and recovering from them is a tedious, error-prone and manual process. Automated techniques can alleviate the problem. Here, we focus on a representative sub-set of the problem: consistency between geometric (Computer-Aided Design – CAD) models of a mechanical system, and the corresponding dynamics simulation models. We have selected two particular but representative modelling tools: SolidEdge for geometric modelling [8], and Modelica [1] for dynamics and control simulation.

The core geometric entities are Assemblies. SolidEdge Assemblies are composed of other Assemblies, Parts and Relationships. Relationships describe mechanical constraints between geometric features of two distinct parts, and there can be many such relationships between parts.

On the dynamics side, to represent an equivalent structure in Modelica, we have a model which can be hierarchically composed of other models, bodies, relationships and geometric features. This last type of model element is introduced to have a counterpart to represent the geometric information which is intrinsic to a SolidEdge part.

Associations (correpondences) that must exist between SolidEdge and Modelica models are shown in a meta-model triple in Figure 2. Note that this model is *declarative* as it does not specify how and what to modify to correct possible inconsistencies. Triple Graph Grammar theory [7] introduced by Schür provides a procedure for automatically deriving operational update transformations (in the form of triple graph rewrite rules) from the declarative meta-model [6]. If either the geometry or dynamics models change, the association model can be used to determine what has been added or deleted from either side.

On the other hand, the use of EOOLs to complement Triple Graph Grammars (TGGs) in an attempt to come up with a fully “declarative” description of consistency between models to support co-evolution of models;

4. Modelling of Variable Structure

Various formalisms have been devised to describe the discontinuous change of the structure of systems. The rule-based description of graph transformations is ideal to elegantly describe structural change. A rule’s left-hand-side describes the conditions under which a state-event occurs. In modelling languages for hybrid systems, crossing conditions on variable values are used to specify *when* a state-event occurs. The handling of a state-event may introduce discontinuous changes in the value of variables. The rule-based approach adds detection of particular object configurations to the low-level variable-value conditions. A rule’s right-hand-side describes the handling of the state-event. This may not only include variable value changes, but also creation/destruction of entities and their interconnections. A promising avenue for future research is the modular

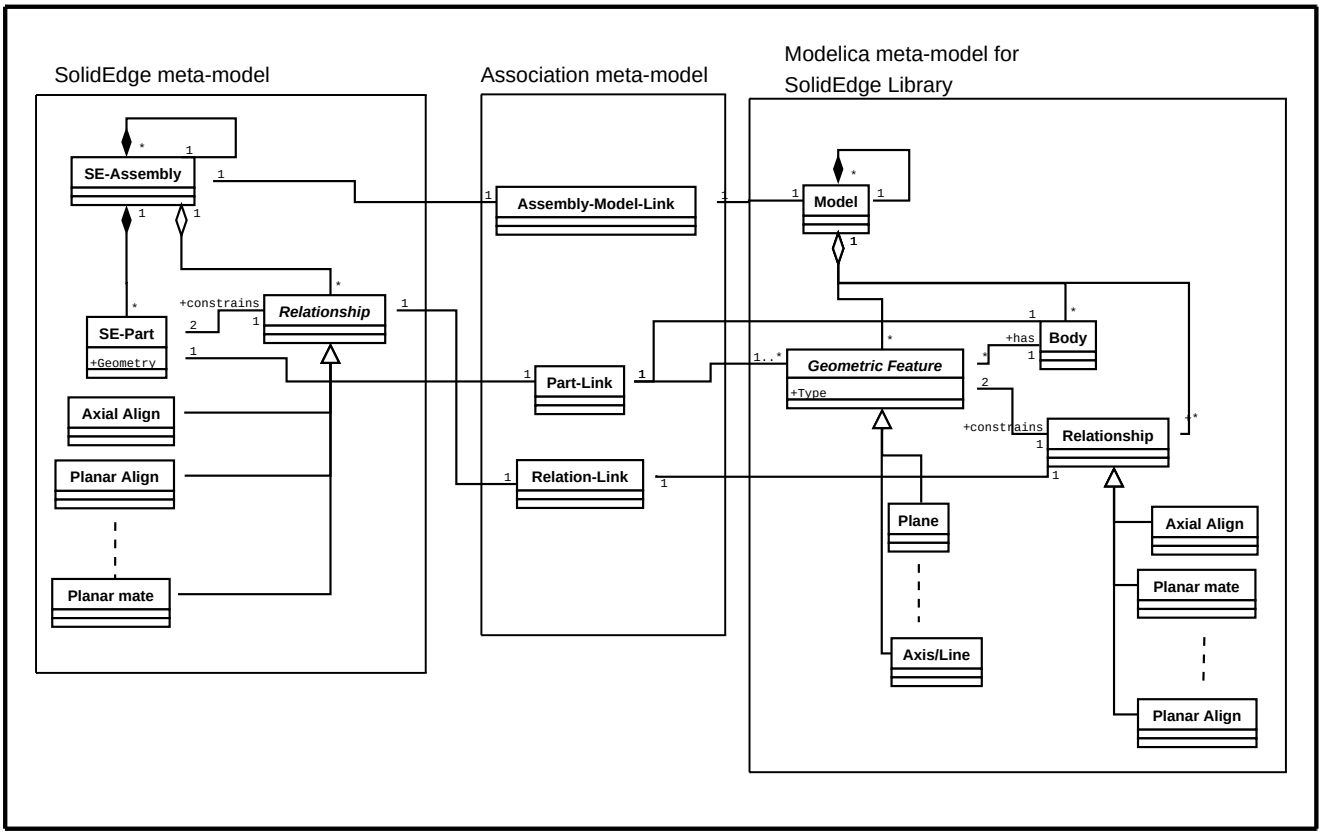


Figure 2. Relating SolidEdge and Modelica models

"weaving in" of rule-based variable structure description language constructs into non-dynamic-structure modelling languages such as EOOLs.

References

[1] Modelica™ Association. A unified object-oriented language for physical systems modeling. Modelica homepage: `www.modelica.org`, since 1997.

[2] F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[3] Juan de Lara and Hans Vangheluwe. AToM^3: A tool for multi-formalism and meta-modelling. In *European Joint Conference on Theory And Practice of Software (ETAPS), Fundamental Approaches to Software Engineering (FASE)*, Lecture Notes in Computer Science 2306, pages 174 – 188. Springer, April 2002. Grenoble, France.

[4] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Technical report, Jerusalem, Israel, 2000.

[5] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley, 2008.

[6] Alexander Königs. Model Transformation with Triple Graph Grammars. In *Model Transformations in Practice Satellite Workshop of MODELS 2005, Montego Bay, Jamaica*, 2005.

[7] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In G. Tinhofer, editor, *WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science (LNCS)*, pages 151–163, Heidelberg, 1994. Springer Verlag.

[8] SolidEdge®. `www.solidedge.com`.

Short Biography

Hans Vangheluwe is an Associate Professor in the School of Computer Science at McGill University, Montreal, Canada. He heads the Modelling, Simulation and Design (MSDL) research lab. He has been the Principal Investigator of a number of research projects focused on the development of a multi-formalism theory for Modelling and Simulation. Some of this work has led to the WEST++ tool, which was commercialised for use in the design and optimization of bioactivated sludge Waste Water Treatment Plants. He was the coordinator of the "Simulation in Europe" Basic Research Working Group. He was also one of the original members of the Modelica design team. His current interests are in domain-specific modelling and simulation and more in general, tool support for Multi-Paradigm modelling. MSDL's tool AToM^3 (A Tool for Multi-formalism and Meta-Modelling) developed in collaboration with Prof. Juan de Lara uses meta-modelling and graph grammars to specify and generate domain-specific environments. He has applied model-driven techniques in a variety of areas such as modern computer games, dependable and privacy-preserving systems (the Belgian electronic ID card), embedded systems, and to the design and synthesis of advanced user interfaces.

Seamlessly Integrating Software & Hardware Modelling for Large-Scale Systems

Toby Myers[1] Peter Fritzson[2] R. Geoff Dromey[1]

[1]School of Information and Computing Technology, Griffith University, Australia,
toby.myers@student.griffith.edu.au, g.dromey@griffith.edu.au
[2]Department of Computer and Information Science, Linköping University, Sweden, petfr@ida.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 modelled separately and with different approaches. This can cause failures later in development during the integration of software and hardware designs, due to incompatible assumptions of software/hardware interactions. This paper proposes a method of integrating the software engineering approach, Behavior Engineering, with the mathematical modelling approach, Modelica, to address the software/hardware integration problem. The environment and hardware components are modelled in Modelica and integrated with an executable software model designed using Behavior Engineering. This allows the complete system to be simulated and interactions between software and hardware to be investigated early in development.

Keywords software-hardware codesign, large-scale systems, Behavior Engineering, Modelica.

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.

One approach of evaluating software/hardware integration is to build prototypes of the software and hardware. This approach allows software/hardware interactions to be investigated, but also diverts attention away from the individual modelling of the respective software and hardware models. Investigating integration using software/hardware prototypes also has the disadvantage of occurring later in development, requiring decisions to have already been made as to how integration will occur.

Addressing the issues involved with integrating software and hardware models of systems earlier in development can reduce the risk of incompatibilities between the software and hardware specifications. Earlier investigation of software/hardware interactions minimises changes that must be made later in development when they are harder and more expensive to fix. If the method of investigating integration uses simulation of specifications, it allows many different integration configurations to be evaluated to assist in finding the best solution. The simulation of software/hardware co-specifications uses abstract models of the software and hardware to focus on timing of the interactions between the hardware and software. Co-specification simulation is used by many system design tools such as STATEMATE and MATLAB [6].

The principle of separation of concerns advocates that due to the differing nature of software and hardware, different modelling techniques should be used. Software modelling consists of capturing the required functionality, and how the functionality can best be organised to facilitate future reuse, extensibility, etc. Hardware modelling focuses on interactions with the physical environment through sensors and actuators which is best described mathematically. Currently, UML is the dominant graphical modelling notation for software, whereas Modelica is the major equation-based object-oriented (EOO) mathematical modelling language for modelling complex physical systems.

Previous work in this area resulted in the ModelicaML UML profile [15, 14] partly based on the SysML profile [13]. ModelicaML combined the major UML diagrams with Modelica graphic connection diagrams. However, there are problems with this approach. The imprecise

2nd International Workshop on Equation-Based Object-Oriented Languages and Tools. July 8, 2008, Paphos, Cyprus.

semantics and portability problems of UML create difficulties for executable specifications. Moreover, there is no well-defined process of precisely capturing and converting informal software requirements into more formal representations that can be analysed and further transformed into executable models.

Fortunately, the Behavior Engineering (BE) approach (see Section 3) addresses several of these problems. BE is a systems & software engineering approach of modelling software-intensive systems that has precise requirements capture. The behavioral view of BE has a formal semantic described in process algebra. BE also supports model-checking, simulation, and the code-generation of executable models.

Thus, we propose an integrated approach, where BE is used to model and capture requirements of the software aspects of a product, whereas Modelica is used for high-level modelling of the system's environment and hardware components. We consider the integration method to be seamless, as the software and hardware models are combined in an inconspicous way which allows both formalisms to focus independently on their respective domains. We also propose this method is suited to be applied to large-scale systems, as both BE and Modelica have been used independently to model large-scale systems [16, 7]. This distinguishes this approach from co-design approaches such as COSMOS[9] and Polis[2] which are focused towards the more fine-grained software/hardware interactions of embedded systems.

Adoption of an integrated approach to product/system design should allow for a much more effective product development process since a system can be analysed and tested in all stages of development. The integration of BE and Modelica models supports this through allowing different hardware/software configurations to be investigated, such as:

- The periodic/aperiodic sampling of sensors and the action of actuators on the physical environment can be simulated to determine the effect on the software of the system. This may also involve simulating the failure of a sensor/actuator or errors in communication.
- The capabilities of the various combinations of hardware and software platforms on which the software could be deployed can be simulated by choosing periodic/aperiodic frequencies at which to allow interactions between the Modelica and BE models.
- The hardware and software can be tested in different simulated environments and scenarios.

In this paper we combine these two formalisms for the first time, in a study of the integrated software/hardware modelling of an Automated Train Protection (ATP) system. BE is used to model the control software of the ATP system, and Modelica is used to model physical components like the train, the driver, actuators, sensors, etc. The modelled ATP system is used to illustrate the benefits of investigating the integration of software/hardware specifications early in development.

In Section 2 & 3 we first give some background on Modelica and BE, before presenting the details of our integration method in Section 4. The integration method is then applied to a case study of the system modelling and simulation of an ATP system in Section 5.

2. Modelica Background

Modelica [12, 17, 8] is an open standard for system architecture and mathematical modelling. It is envisioned as the major next generation language for modelling 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 modelling 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.

A component may internally consist of other connected components, i.e., as in Figure 1 showing hierarchical modelling.

The multidomain 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 modelling.* 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 modelling 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 modelling, the structure of a Modelica model naturally corresponds to the structure of the physical system.

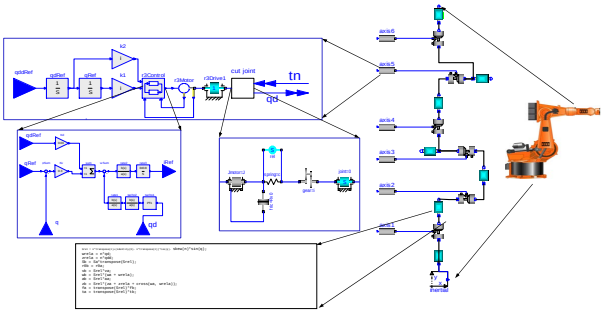


Figure 1. Hierarchical Modelica model of an industrial robot

- *Acausal modelling.* Modelling 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 support the Modelica specification, ranging from open-source products such as OpenModelica [12], to commercial products like Dymola [5] and MathModelica [11].

3. Behavior Engineering Background

BE [3] is an integrated approach that supports the engineering of large-scale dependable software intensive systems at both the systems engineering and software engineering level. BE has been proven as a useful technique in requirements analysis of large-scale industry projects, detecting defects at a rate approximately two to three times higher than conventional techniques [16]. The BE approach uses the Behavior Modelling Language (BML) and the Behavior Modelling Process (BMP) to transform a system described in natural language requirements to a design composed of a set of integrated components.

3.1 The Behavior Modelling Language

The BML is a graphical, formal language consisting of three tree-based views: Behavior Trees, Composition Trees and Structure Trees[1].

A Behavior Tree (BT) is a "formal, tree-like graphical form that represents behavior of individual or networks of entities which realize or change states, make decisions, respond-to/cause events, and interact by exchanging information and/or passing control." [4]. The formal semantics of BTs are described in the Behavior Tree Process Algebra (BTPA) language [1]. BTPA supports simulation, formal verification by model-checking and is a foundation for BT execution. BTs can describe multiple threads of behavior. Coordination is achieved using either message-passing (events), shared variable blocking or synchronisation. A summary of the BT notation is shown in Figure 2.

Composition Trees (CTs) contain the complete system vocabulary, which is consistent with the vocabulary used in BTs as they both originate from the same natural language requirements. CTs are a tree of components arranged into a compositional hierarchy using structural and functional aggregation or specialisation relations. Each component in the BT contains the complete set of states, attributes, events and relations in which the component is responsible for. CTs are an important tool in resolving defects not visible in individual Requirement Behavior Trees, such as aliases.

3.2 The Behavior Modelling Process

The BMP is closely tied with the BML. The BMP consists of a number of distinct stages: Translation, Integration, Refinement and Design. Each of these stages utilises the BML to address the problems of scale, complexity and imperfect knowledge that arise when dealing with systems described by a large number of natural language requirements.

Translation proceeds one requirement at a time, resulting in a Requirement Behavior Tree (RBT) that is created from the original natural language description. As each RBT is translated, the Requirement Composition Tree (RCT) should be updated to include any new information such as additional components, states, etc. Also, in order to ensure the translation process is as rigorous as possible, it is important not to add or remove information but to capture the intention that is expressed in the natural language description.

Being able to deal with one requirement at a time, localises the information that the modeller must absorb and helps to control the complexity of modelling the system. It also makes it possible for a team of translators to work on modelling the system in parallel, using the RCT to coordinate their work.

Two example RBTs are shown in Figure 4. Discussion of the translation of an example RBT from the original requirements is discussed in section 5.1.

Once all the requirements have been translated they are integrated to form an Integrated Behavior Tree (IBT) which can then be used to gain a holistic understanding of the problem space. The process of integration itself also helps to discover imprecise, conflicting and missing requirements in the description of the system. This is because forming the IBT is a fitness test for the requirements, if requirements cannot integrate it indicates there are problems with the description of the system.

When the IBT has been completed, the integrated view of the system's behavior helps to detect further defects in the original natural language requirements. Resolution of these defects produces a specification of the system, known as a Model Behavior Tree (MBT).

As the specification is still in the problem space, design decisions must be made to move to the solution space. The result is a Design Behavior Tree (DBT). Important design decisions include determining the boundaries between the system and the environment and the system and the components. The system-environment boundary determines how the system described by the DBT interacts with the environment, essentially determining the interface of the system. The system-component boundary involves a tradeoff between shifting complexity to either the DBT or to the components.

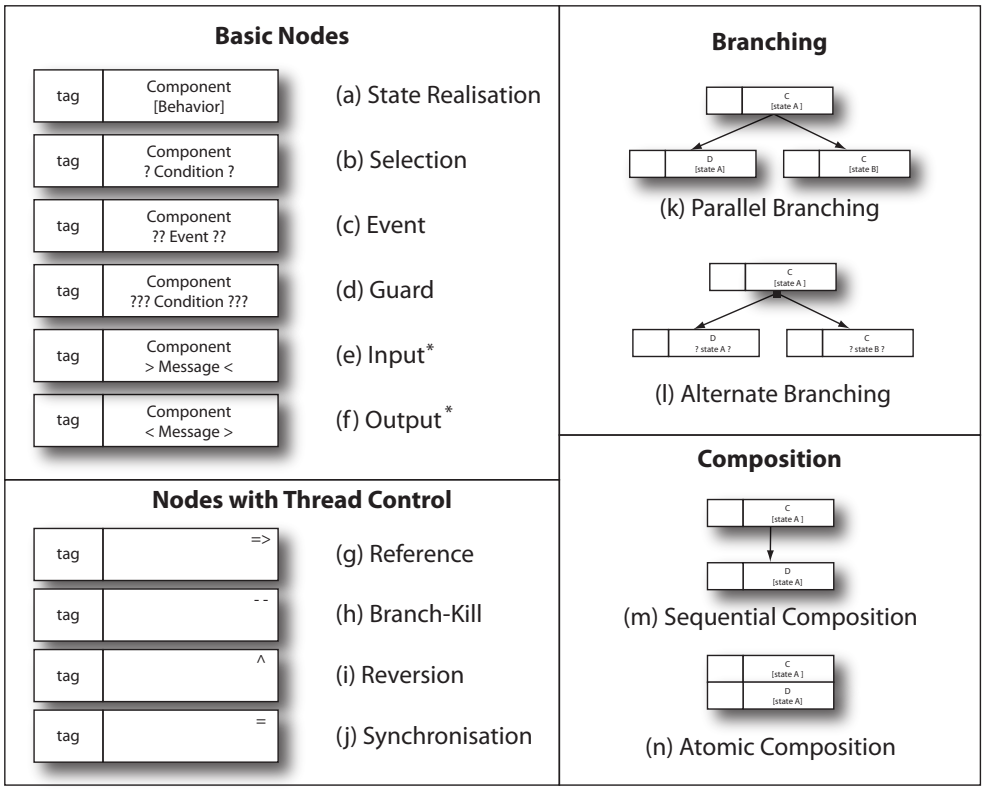
An example DBT is shown in Figure 5. The design decisions used to make this DBT are described in Section 5.1.

3.3 Executing a BE Model

BTs contain a description of the functionality of the system which makes them the primary interest when discussing executable models.

One approach to execute a BE model is to consider a BT as a set of interconnected interleaved state machines. Each component can be implemented by decomposing its individual state machine and implementing it. The BT is

[1] Due to space restrictions Structure Trees will not be discussed



(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

also implemented as a state machine which coordinates the component state machines.

Another approach to execute a BE model is to consider BTs as a model to describe multiple-threaded behavior, making each BT node a process. This allows traditional process control schedulers consisting of New, Ready, Blocked, Running, and Exit states to be applied to BTs.

For example, a state realisation node would take the following path through the scheduler: New, Ready, Running, Exit. Moving from the Ready to Running State is determined by a scheduling algorithm, ranging from simple examples such as First In, First Out (FIFO) to more complex priority-based schedulers. When a state realisation node is in the running state any encapsulated computation associated with the component's state is executed. Upon reaching Exit its child nodes are added to the scheduler in the New state to continue execution.

Alternatively, a guard node would take the following path through the scheduler: New, Blocked, Ready, Running, Exit. The guard node stays in the Blocked state until a change in another thread of behavior causes its condition to become true, upon which it changes to the Ready state and progresses similarly to the state realisation node. The scheduler also consists of more complex rules for BT execution such as alternative branching and atomic composition.

The benefit of the process control approach is that code generation from a BT is easily automatable. All that is required in addition to the automatable code generation is a version of the scheduler for the platform on which the executable BE model is deployed.

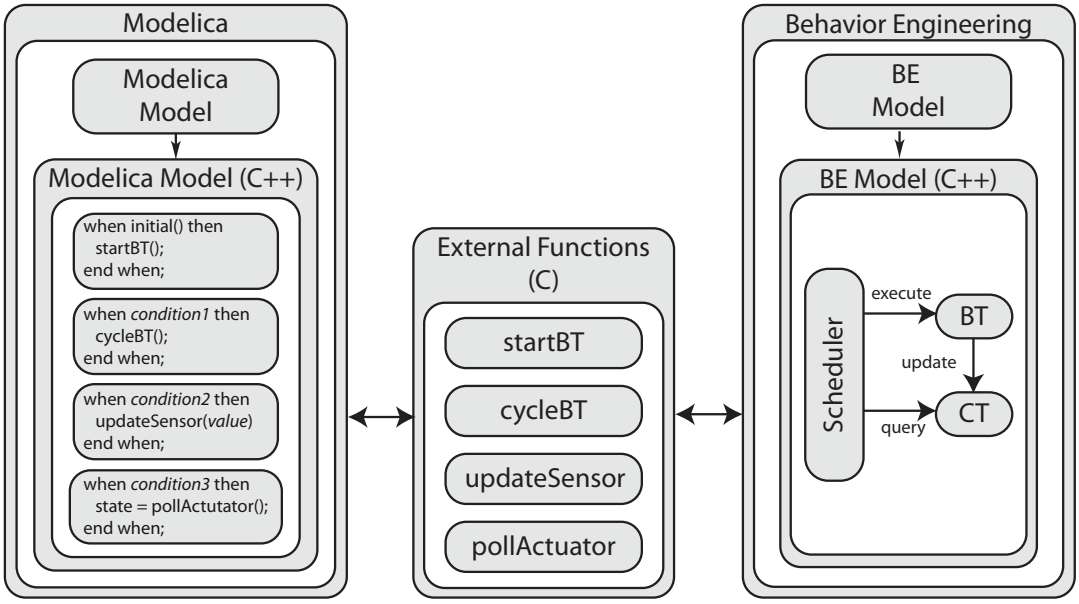


Figure 3. Interactions between Modelica and BE Models

4. Integrating Modelica & BE Models

Integration of Modelica and BE models occurs after the models are compiled/code generated into C++ source files. Integration between the Modelica model and BE model is performed using Modelica external functions mapped to C source code. The 'C' external functions are then linked to the 'C++' implementation of the BE model. This method of integration makes the Modelica model responsible for managing all interactions with the BE model.

Figure 3 shows the integration of a Modelica model and a BE model. There are three possible types of interaction: starting/cycling the BT scheduler; adding an event to the scheduler containing sensor information; or, polling the scheduler for an actuator command. The *initial()* function is used to start the execution of the BT. Boolean conditions are then used to determine when to cycle the BT scheduler, pass on sensor information or receive actuator commands.

If interactions are periodic, a boolean clock setup with a sample function can be used to set the frequency with which the interaction will occur. If the interaction should occur based upon a physical event simulated in Modelica, the event can change the boolean condition which will initiate the interaction with the BE model. More complex aperiodic, randomised, or interactions with losses in communication or failures of components can also be simulated using Modelica constructs. Failures of sensors, actuators or the communication between them and the software can be simulated by mearly not performing the interaction that would normally occur.

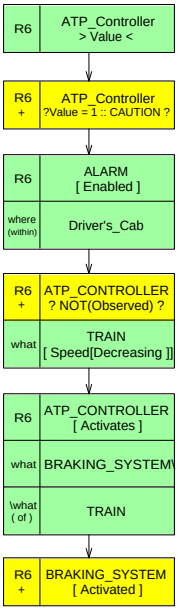
This method of interaction ensures that the details of the interactions that are simulated are documented as part of Modelica model. It also allows many possible designs to be simulated by considering how they will effect the timing of the interactions between the physical and software systems. For example, if the software is to be run on a multi-threaded operating system, the boolean condition could consist of a timing profile which emulates at what times the BT scheduler will be executed. This timing profile could be randomised to determine how the system operates under different loads, or may just address one specific or worst-case scenario. If more than one operating system is being considered, a timing profile could be setup for each operating system and multiple simulations peformed to determine the differences, if any, on the system as a whole.

5. Case Study: An Automated Train Protection System

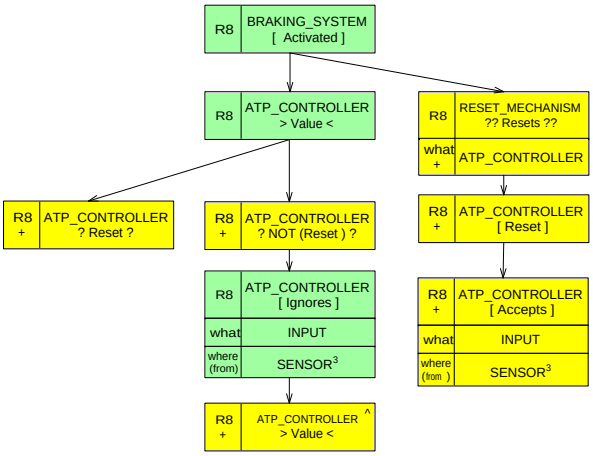
Most rail systems have some form of train protection system that use track-side signals to indicate potentially dangerous situations to the driver. The simplest train protection systems consist of signals with two states: green to continue along the track and red to apply the brake to stop the train. More sophisticated systems include detailed information such as speed profiles for each section of the track.

Accidents still occur using a train protection system when a driver fails to notice or respond correctly to a signal. To reduce the risk of these accidents, Automated Train Protection (ATP) systems are used that automate the train's response to the 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.

The ATP system used for this paper has three track-side signals: proceed, caution and danger. When the ATP system receives a caution signal, it monitors the driver's behavior to ensure the train's speed is being reduced. If the driver fails to decrease the train's speed after a caution signal or the ATP system receives a danger signal then the train's



(a) RBT for Requirement 6



(b) RBT for Requirement 8

Figure 4. Example Requirement Behavior Trees of the ATP System

brakes are applied. The complete requirements of the ATP system can be found in Table 1. The requirements of the ATP system have been used previously in related work to demonstrate composition of components using exogenous connnectors [10].

Section 5.1 discusses the construction of the BE model of the ATP system from the requirements and Section 5.2 discusses the Modelica model of the ATP systems physical components and environment.

5.1 ATP - Behavior Engineering Model

Figure 4 shows two example RBTs of the ATP system. 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 condition 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. Relations should be read as questions that can be asked of the primary behavior, which the associated relational behavior answers. For example, "Where is the Alarm enabled? Within the Driver's Cab". Capturing the information about the Driver's Cab ensures that the original intent of the requirements is not removed. The next BT node assumes that it is implied that the ATP Controller observes 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.

During integration of the RBTs of the ATP system the following problems were found:

- *Conflicting Behavior (R7-R8).* After the Braking System is activated, R7 states that a proceed signal disables the Alarm whereas R8 states all sensor input is ignored until the ATP Controller is reset.
- *Conflicting Behavior (R7-R9).* After the Braking System is activated, R7 states that a proceed signal disables the alarm whereas R8 states that the Reset Mechanism deactivates the Train's Brakes and disables the Alarm.
- *Missing Behavior (R6).* What should the ATP Controller do if the Train's speed is observed to be decreasing?
- *Missing Behavior.* What should the ATP Controller do if an undefined signal is returned to the ATP Controller?

Each of these problems would need to be resolved with the client to ensure that the system behaves as is desired. However for the purposes of this case study the following assumptions were made:

- R8 and R9 were given priority over R7. That is, a proceed signal can only disable the Alarm after the Alarm has been enabled but prior to the Brakes being activated. After the Brakes have been activated all sensor input is ignored until the ATP Controller is reset. Also, resetting the ATP Controller after the Brakes have been activated causes the Train's Brakes to be deactivated and disables the Alarm.
- If the ATP Controller observes the train's speed to be decreasing then: if a danger signal is received the Brakes are immediately activated; or, if a proceed signal is received the Alarm is disabled. However, if the train's speed increases before either of these signals are received then the ATP Controller should activate the Train's Braking System.

Figure 5 shows the DBT of the ATP system resulting from design decisions made to the MBT. A (M) in the tag shows the nodes of the DBT where interaction occurs with the Modelica model. The following design decisions were made to the MBT:

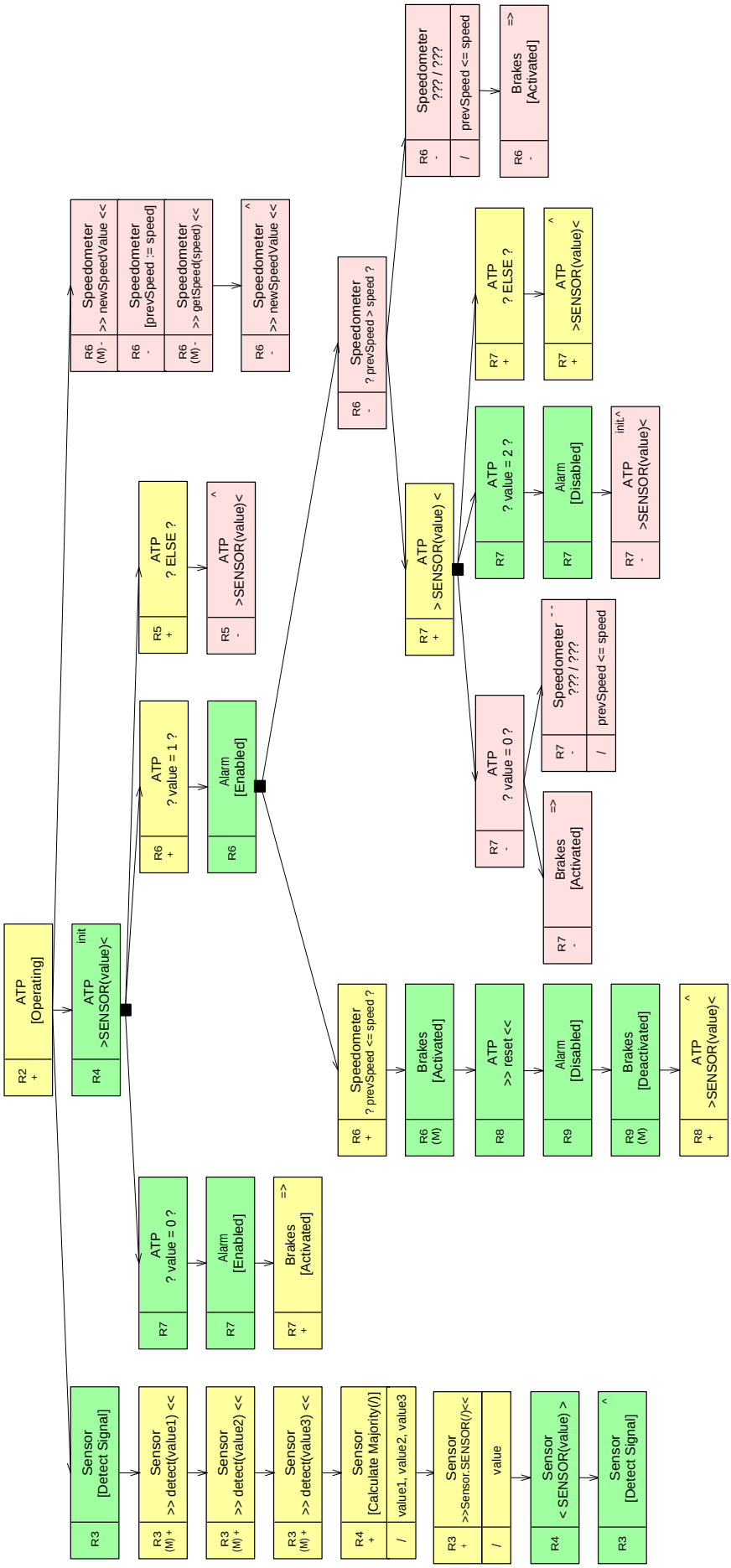


Figure 5. Design Behavior Tree of ATP system

- Train, Signal, the individual Sensors, Driver's Cab, Reset Mechanism, and Noise components are outside the boundaries of the DBT.
- A Speedometer component is required to receive the train's speed and store the previous speed value so that changes in the speed of the train may be determined.
- Alternative branching and atomic composition was added to ensure appropriate threaded behavior. Atomic composition is required for when the speedometer component's speed value is updated. This is because for a small period of time the current speed equals the previous speed causing the prevSpeed<=speed guard to evaluate to true, regardless of the new speed value. Alternative branching ensures that once one of the mutually exclusive branches has been taken (e.g. value=0), none of the other branches can be executed (e.g. value=1, ELSE).

5.2 ATP - Modelica Model

The Modelica model describes the physical components that make up the environment in which the ATP system will operate. It consists of components such as the Train, the Driver, the Train Track, and the Sensors of the track-side signals. Figure 6 shows the component diagram of the Modelica model. The Driver component is responsible for controlling the Train's speed and resetting the ATP system. The Train component simulates its velocity and position on the Track based upon its mass, maximum acceleration power and maximum brake force. The Train Track provides the signal sensors with the signal value at the signal position. The signal sensors then simulate the presence of noise, occasionally misreading a signal value. The sensor values, Train speed and driver reset are all provided to the ATP controller which in turn provides whether to apply the Train's Brakes. A simplified version of the Modelica textual model of the ATP environment is shown in Figure 7.

5.3 Integration of the Modelica and BE Models

Simulating the integrated Modelica/BE models provides plots which graphically show the interactions between software and hardware in reference to time. This allows the investigation and documentation of scenarios in a clear way. The types of scenarios that can be investigated are:

- The frequency of the execution of the BE model relative to the Modelica model simulates the performance capabilities of the hardware platform on which the BE model will be deployed.
- The sampling frequency/response time of sensors and actuators can be simulated by the frequency of interaction between the Modelica model and the BE model.
- The system can be tested with different Trains, Drivers, Train Tracks, etc.

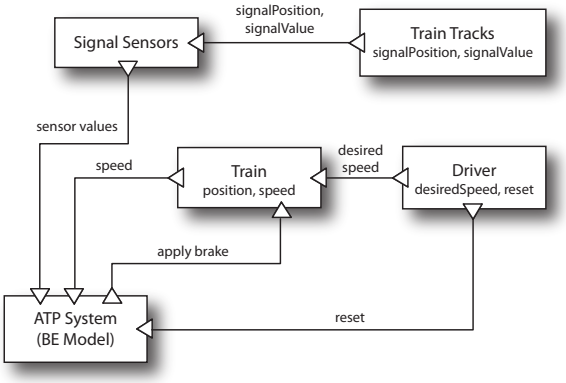


Figure 6. Component diagram of the Modelica ATP Environment model

Figure 8 shows four example simulations of the integrated model of the ATP System. All the simulations are performed with a train model based on a British Rail Class 57 diesel locomotive, which has a mass of 120 tonnes, a maximum speed of 120.7 km/h, a maximum brake force of 80 tonnes and a power at rail of 1860 kW with an assumed 80% efficiency due to losses in pressure and friction.

The train's braking time of two seconds is due to its low velocity (approximately 45km/h) and small weight due to the absence of carriages. The same train operating at 100 km/h would take approximately eight seconds to brake, and at 200 km/h would take 32 seconds. The addition of carriages would further increase the time the train would take to brake to a complete stop. These braking times highlight the need to test software-hardware integration under numerous circumstances.

The simulations performed on this case study show the ATP system operating with the same configuration of sensors, actuators and hardware platform. The change that is tested is the driver's response to the signals on the track, the results of which now ensures that the ATP system is functioning as specified by the requirements. Further simulations could now be performed to investigate the ATP system operating both in different scenarios and also the suitability of different sensors, actuators and hardware platforms.

6. Conclusion

This paper investigates the software/hardware integration problem caused by the increasing codependancy of software and hardware in large-scale systems. An integrated approach is described, which integrates separate software and hardware models to aid the investigation of software/hardware interaction through simulation. An ATP system is used as a case study to describe both separate software/hardware modelling with BE and Modelica and software/hardware integration and investigation. This integrated approach allows various software/hardware interactions to be investigated such as software execution speed, sensor sampling frequencies, and actuator response times. It also provides a graphical and documentable output of the investigation the behavior of the software and hardware in different scenarios.

Acknowledgments

This work was produced with the assistance of funding from the Australian Research Council (ARC) under the ARC Centres of Excellence program within the ARC Centre of Complex Systems (ACCS), the Swedish Vinnova under the Safe and Secure Modeling and Simulation project and the Swedish Research Council (VR).

References

[1] Robert Colvin and I. J. Hayes. A Semantics for Behavior Trees. ACCS Technical Report ACCS-TR-07-01, ARC Centre for Complex Systems, April 2007.

[2] Tullio Cuatto, Claudio Passeronge, Luciano Lavagno, Attila Jurecska, Antonino Damiano, Claudio Sansoè, A. Sangiovanni-Vincentelli, and Alberto Sangiovanni-Vincentelli. A case study in embedded system design: an engine control unit. In *Proceedings of the 35th annual conference on Design automation (DAC '98)*, pages 804–807, New York, NY, USA, 1998. ACM.

[3] R. G. Dromey. Formalizing the Transition from Requirements to Design. In Jifeng He and Zhiming Liu, editors, *Mathematical Frameworks for Component Software - Models for Analysis and Synthesis*, pages 156–187. World Scientific Series on Component-Based Development, 2006. Invited Chapter.

[4] R.G. Dromey. From Requirements to Design: Formalizing the Key Steps. In *IEEE International Conference on Software Engineering and Formal Methods*, pages 2–11, Brisbane, Sept 2003. SEFM-2003. Invited Keynote Address.

[5] Dynasim. Dymola. http://dynasim.com.

[6] Rolf Ernst. Codesign of embedded systems: Status and trends. *IEEE Design and Test*, 15(2):45–54, 1998.

[7] Peter Fritzon, Vadim Engelson, Andreas Idebrant, Peter Aronsson, Håkan Lundvall, Peter Bunus, and Kaj Nyström. Modelica âĂŞ A Strongly Typed System Specification Language for Safe Engineering Practices. In *Proceedings of the SimSAFE Conference*, Kralskoga, Sweden, June 2004.

[8] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, 2004.

[9] Tarek Ben Ismail, Mohamed Abid, and Ahmed Jerraya. Cosmos: a codesign approach for communicating systems. In *Proceedings of the 3rd international workshop on Hardware/software co-design (CODES '94)*, pages 17–24, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[10] Kung-Kiu Lau, Ling Ling, and Zheng Wang. Composing Components in Design Phase using Exogenous Connectors. In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO '06)*, pages 12–19, 2006.

[11] MathCore. Mathmodelica. http://www.mathcore.com.

[12] Modelica Association. Modelica: A Unified Object-Oriented Language for Physical Systems Modeling: Language Specification Version 3.0, Sept 2007. http://www.modelica.org.

[13] OMG. System Modeling Language (SysML). http://www.omgsysml.org.

[14] Adrian Pop, David Akhvlediani, and Peter Fritzon. Integrated UML and Modelica System Modeling with ModelicaML in Eclipse. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications (SEA'07)*, 2007.

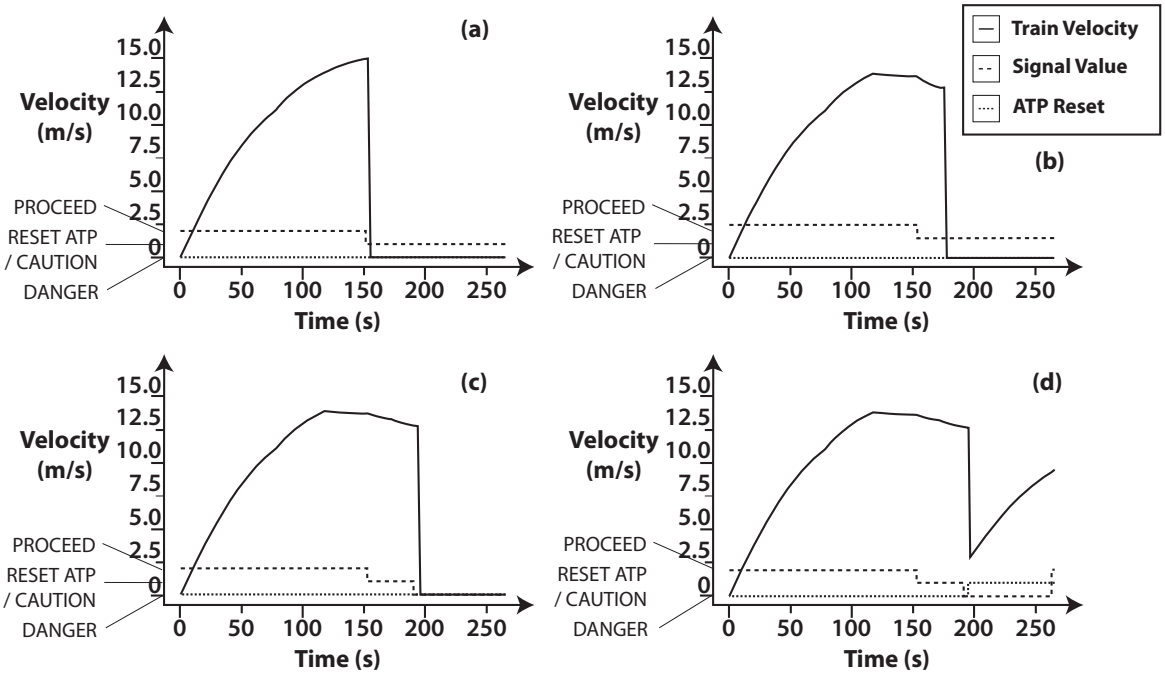
[15] Adrian Pop, David Akhvlediani, and Peter Fritzon. Towards Unified System Modeling with the ModelicaML UML Profile. In *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT'07)*, pages 13–24, 2007.

[16] Danny Powell. Requirements evaluation using behavior trees - findings from industry. In *Australian Software Engineering Conference (ASWEC'07)*, April 2007.

[17] Michael Tiller. *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers, 2001.

```
// External Functions included here

model Track
  discrete Integer currentSignalValue "Value of Last Signal displayed to Driver/ATP System";
  parameter Real[:] signalPosition "Positions of Signals on the Track";
  parameter Integer[:] signalValue "Values of Signals on the Track";
equation
  // Determine current signal value
end Track;

model Train
  Real s, v, m, maxSpeed, maxBrakeForce, maxAccelerationPower, maxAccelerationForce;
  parameter Real accPowerEff = 0.80 "Engine Efficiency in %";
equation
  maxAccelerationPower/accPowerEff = maxAccelerationForce*v;
end Train;

record Driver
  Real desiredAcceleration;
  parameter Real[:] desiredSpeed;
  parameter Real[:] position;
end Driver;

model Main
  // Define track, train, driver parameters

  parameter Real[10] sensor1 = {0,0,1,2,0,0,2,2,0,0} "Sensor1 value at signalPosition";
  Real sensor1Reading "Current Sensor1 reading";
  // Similar for Sensor 2 & 3

  Real fa, fd, doBrake(start=0), minAccelerationForce, desiredAccelerationForce;
  discrete Boolean clock1, clock2, ...;
  // Define clock frequencies

equation
  when initial() then startBT(0); end when;
  when clock1 then cycleBT(0); end when;
  when clock2 then doBrake = if (train1.v >= 0) then getBrake(0) else 0;
  // if driver reset's ATP send message
  // if signal changes send new sensor values

  fa = if doBrake>0 then 0
  elseif // ensure not over maximum Acceleration force
  else desiredAccelerationForce;
  fd = if doBrake>0 then train1.maxBrakeForce else 0;

  a = (fa-fd)/train1.m;
  der(v) = a;
  der(track1.s) = train1.v;
  // if train passing signal then update sensors
  // determine driver's desired acceleration (a = (desiredSpeed - train1.v)/ (2*distance))
end Main;
```

Figure 7. Simplified Textual Modelica model of the ATP Environment



[a] Driver ignores caution signal and increases speed, brakes are activated
[b] Driver sees caution signal and reduces speed but then increases speed, brakes are activated
[c] Danger signal, brakes are activated regardless of driver already decreasing speed
[d] Danger signal, brakes are activated, ATP is reset and brakes are deactivated

Figure 8. Simulation of the ATP System

Requirement	Description
R1	The ATP system is located on board the train. It involves a central controller and five boundary subsystems that manage the sensors, speedometer, brakes, alarm and a reset mechanism.
R2	The sensors are attached to the side of the train and detect information on the approach to track-side signals, i.e. they detect what the signal is displaying to the train driver.
R3	In order to reduce the effects of component failure three sensors are used. Each sensor generates a value in the range 0 to 3, where 0, 1 and 2 denote the danger, caution, and proceed signals respectively. The fourth sensor value, i.e. 3, is generated if an undefined signal is detected, e.g. may correspond to noise between the signal and the sensor.
R4	The sensor value returned to the ATP controller is calculated as the majority of the three sensor readings. If there does not exist a majority then an undefined value is returned to the ATP controller.
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.
R7	In the case of a danger signal being returned to the ATP controller, the braking system is immediately activated and the alarm is enabled. Once enabled, the alarm is disabled if a proceed signal is subsequently returned to the ATP controller.
R8	Note that if the braking system is activated then the ATP controller ignores all sensor input until the system has been reset.
R9	If enabled, the reset mechanism deactivates the train's brakes and disables the alarm.

Table 1. Requirements of the ATP system

The Impreciseness of UML and Implications for ModelicaML

Jörn Guy Süß[1] Peter Fritzson[2] Adrian Pop[2]

[1]ITEE, The University of Queensland, Australia, jgsuess@itee.uq.edu.au
[1]Linköpings Universitet, Sweden, {petfr,adrpo}@ida.liu.se

Abstract

The Modelica community has long pursued the vision of Integrated Whole Product Modelling. This implies the ability to integrate best practice modelling languages and techniques. With ModelicaML a first step towards an open integration within the sphere of the Eclipse Modelling Framework exists. This paper argues for a development direction of ModelicaML that creates a small core with well-defined semantics, instead of the current version that is based on an extension of SysML. To this end, modelling standards and their practicabilities are discussed and exemplified through a usage scenario.

Keywords EMF, UML, ModelicaML, design

1. Introduction

Modeling in general and object-oriented modeling in particular have shown themselves to be useful tools on a software engineer's workbench. With ever more software being produced for increasingly complicated application areas, adequate semantics and languages gain in importance. The term Model is heavily overloaded, even in software engineering. In general, a model is a purpose-built abstraction of something. It exhibits properties that are essential to the abstraction and can hence be treated like the modelled object with regards to those properties.

Very often, a model is used as for simulation. Here, we take 'simulation' to mean an experiment carried out on a model, as opposed to the modelled subject itself [11]. For example, the sentence "a heart is an open book" defines a model for human emotional behaviour that may imply a certain mode of access, changeability, etc. As the heart is not a book, the model can only be used to make predictions for the intended purpose, i.e. to explain human emotional behaviour. This model will fail as a simulation of biological behaviour. The example also shows that the frame of reference or school of thought defines the allowable shapes of a model and the predictions it will make. In the example, whatever *our* mind expects of an open book, we will ascribe to the emotional behaviour of the heart. People will differ in their expectation of the structure and behaviour of books. Hence their mental simulations of the behaviour of the human heart will differ.

While the contemplation of human emotions based on such private mental models is useful and enjoyable, systems engineering is a collaborative task, involving a lot of interaction. The communication about the subject has to be adequate for this use. If we communicate about systems in order to build them effciently, safely and correctly, our school of thought or language to express our models needs to be unambiguously standardised. It also needs to practically usable, which implies terseness and focus on the task at hand.

In this spirit this paper proposes a refactoring of the ModelicaML modelling language that is used to fashion engineering models based on the Modelica language in order to simulate them. We propose that in order to expose Modelica in models, a specific object-oriented approach known as Meta-modeling should be used to create the interface. This involves both creation of visual editors, and storage facilities. We will introduce Meta-Modelling and the associated Meta-Object Facilities as we go along.

The rest of the paper is structured as follows: Section 2 provides an overview of the current version of ModelicaML. Section 3 explains MOF and metamodelling in broad terms. Section 4 describes metamodelling in more detail and argues for a specific product as the basis of ModelicaML. Section 5 gives reasons why certain parts of the current version of ModelicaML should be removed. Section 6 describes how ModelicaML can be used to model an engineering problem and predict relevant properties. Section 7 summarizes the recommendations.

2. Overview of ModelicaML

The current version of ModelicaML is a customisation or profile of SysML, which in turn is a customisation or profile of UML.

In industrial practice, the term profile is taken to mean an alteration of an existing modelling language, that is transitively related to UML through through profiling; ModelicaML is a profile of SysML, which is a profile of UML.

2nd International Workshop on Equation-Based Object-Oriented Languages and Tools. July 8, 2008, Paphos, Cyprus.

Every alteration step can affect the semantics or the visual representation of concepts. They can narrow, refine, extend and change concepts of the profiled source language. Effectively, the term profile only means that the new language is somehow related to UML, in order to increase its popularity. As a result, there are no technically standardised means to capture this wider notion of profiles. The stricter notion of profiles is discussed in Subsection 5.5.

With respect to SysML, ModelicaML reuses, extends and provides several new diagrams. The ModelicaML diagram overview is shown in Figure 1. Diagrams are grouped into four categories: Structure, Behavior, Simulation and Requirement. The ModelicaML profile is presented in [1], [29], and [30]. The most important properties of the ModelicaML profile are the following:

- The ModelicaML profile supports modeling with all Modelica constructs and properties i.e. restricted classes, equations, generics, variables, etc.
- Using ModelicaML diagrams it is possible to describe most of the aspects of a system being designed and thus support system development process phases such as requirements analysis, design, implementation, verification, validation and integration.
- The profile supports mathematical modeling with equations since equations specify behavior of a system. Algorithm sections are also supported.
- Simulation diagrams are introduced to model and document simulation parameters and simulation results in a consistent and usable way.
- The ModelicaML meta-model is consistent with SysML in order to provide SysML-to-ModelicaML conversion.

The current version of ModelicaML is based on SysML, which in turn is promoted as a variant of UML. This paper proposes that the next revision of ModelicaML should be reduced and made independent of SysML and UML. To support this argument, we give an introduction to the relevant technologies and then turn to a usage example.

3. Why expose Modelica through MOF?

This section of the paper proposes that Modelica should be represented through MOF (Meta Object Facility) for all its technical external representation, because MOF is easy to understand, and hence to integrate and is backed by solid technology.

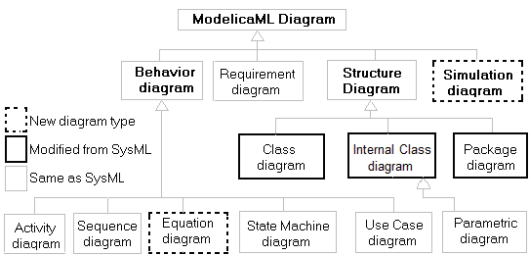


Figure 1. ModelicaML diagram overview.

The MOF is a technical framework built on the assumption that engineering languages can be described well in the ontological terms that underlie object orientation: The world consists of complex objects with primitive attributes. These belong to homogenous classes. The class may be related through an is-a relationship (sub-classing), an is-part-of relationship (aggregation), or via simple association. The latter two may be constrained by cardinalities.

With this tenant, every engineering language can be described as a class diagram. Figure 2 shows such a diagram for a database. An instance of this meta-model, would be a model for a database. Some details of the diagram read like this: ForeignKey, Column, Table and Key are all ModelElements and hence have the attributes 'name' and 'kind'. A Table is made up of ForeignKeys, Columns and Keys. A ForeignKey is associated with a Key. The API generated by a MOF service would allow to create the four mentioned elements and to set their relationships.

Essentially, MOF can be thought of as a standard for defining the cores of domain-specific software engineering tools, much like SQL is a standard for defining cores of relational databases. With MOF, schemas are compiled and the resulting machinery provides a low-volume storage facility and a rich API-based interface. We will describe MOF in more detail in the next section.

MOF combines the natural 'ontological' way of describing a technical area with a powerful generator framework which reduces the cost of building and maintaining a domain specific modeling tool. The ontological description can be represented in diagrams, which is easier to understand than the code of a hand-written modelling tool. In MOF the model-handling code of the tool is generated directly from the diagrams, which avoids errors and tedious manual translation work. For this reason we argue, that Modelica should primarily be exposed through a MOF metamodel for integration with other software engineering tools; This metamodel should be used in the future as its external representation for tool integration purposes.

4. Why should EMF be Modelica's MOF?

This section of the paper proposes that ModelicaML should be based on the MOF variant provided by the Eclipse Modelling Framework, EMF. Before we unfold the argument for this proposition, we will need to inspect MOF in greater detail, as the previous section only presented MOF in fairly abstract terms. This section will provide a bit more detail about the history and practicabilities of MOF. Like SQL, all MOF variants are basically similar, but differ in detail. Hence tools connected to and data defined based on different versions of a MOF variant, say MOF 1.2 and MOF 1.4, are incompatible if one assumes the use of all expressive features. Traditionally the dependencies of MOF are shown as a pyramid, as seen in Figure 3. Normally, this pyramid has three levels. For this paper, we have added a

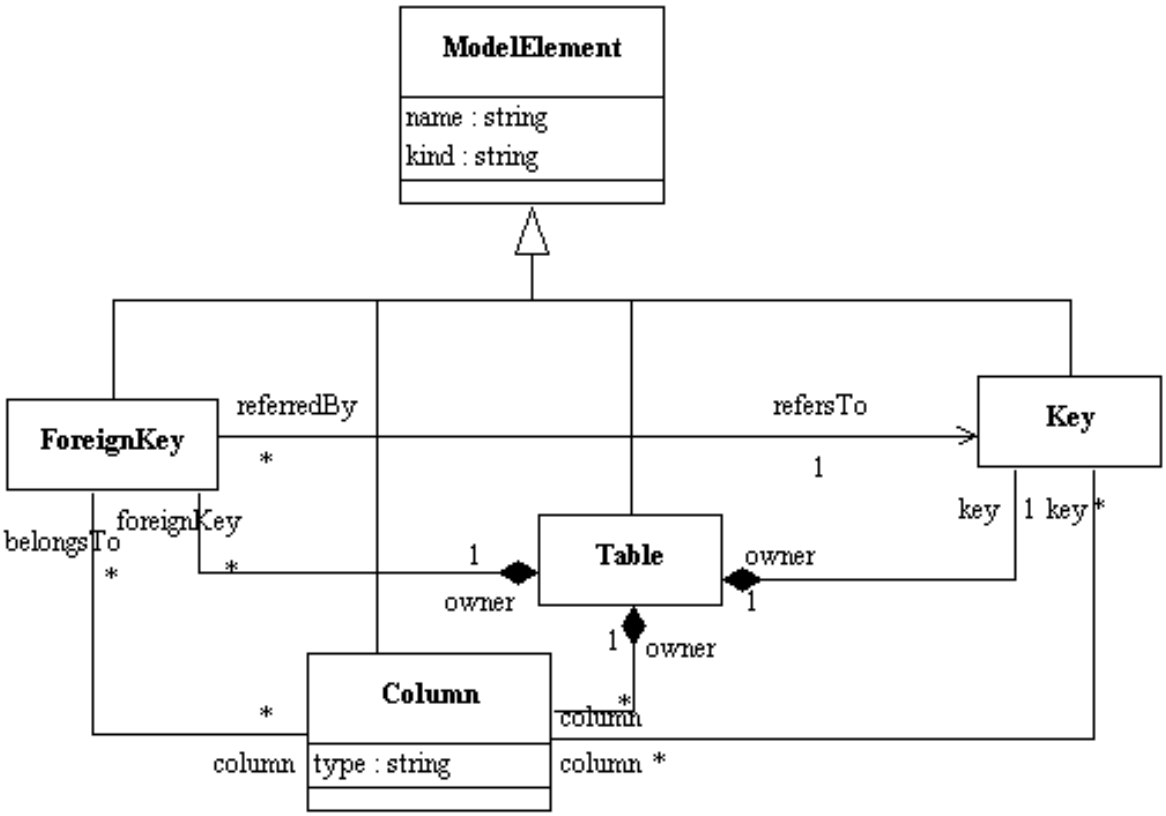


Figure 2. Example of a simple database metamodel. (INRIA)

fourth level as a means to explain the versioning problems that MOF suffers from.

4.1 Philosophy, Standards and Products

At the top of the stack at meta-level four (M4) we find the idea that was introduced in the previous section: object orientation is good for presenting domain-specific languages. This level does not have an equivalent in software. It is purely philosophical and we will call it the philosophy level.

Inspired by this philosophy, developers create implementations of MOF services. Some of the implementation guidelines they adopt are shared in the technical standards of the OMG. However, these standards have never substantially guaranteed exchangeability of services or interchange of artefacts and are thus omitted from the diagram [32, 12]. The OMG is a vendor-based organization. Modelling tools, including MOF implementations, were and are high-price margin software and hence vendors are not interested in interoperability in order to achieve customer lock-in. This is reflected in the nature of the OMG standards, which usually exclude concrete technical detail. Hence, standards in the OMG sense are not standards in a product compliance sense, rather guidelines and inspiration for implementation. In order to submit a standard, a vendor has to show an implementation, but that implementation is not required to work together with reference implementations of related standards. We will call this meta-level three the product level, rather than the standards level. The differences between the implementations usually result in incompatible storage formats, incompatible primitive data types, and incompatible event models of APIs. Thus, A MOF implementation from IBM, from SUN and from SAP would likely be incompatible in these respects.

With the help of a MOF product, software architects can express the domain-specific languages (DSLs) in metamodels. This level is known as the meta-level two, and we will call it the language level. The meta-models of the DSLs are expressed using the features and semantics of a particular MOF implementation that is in turn defined at the product level. Consequently, DSLs cannot generally be exchanged among MOF services.

Once an API has been generated from a metamodel with the help of a MOF product, it can be dressed up with a user interface and serve as a modelling tool for an engineer who is familiar with the corresponding DSL. Following the example above, we could now write the graphic interface to a database design tool. The engineer can now model an engineering problem – the design of a particular database, e.g. for a library - and store and retrieve that model from disk. The well-formedness conditions that have been defined for the engineering language in form of the constraints on the languages class schema help the engineer to design a correct model by vetoing models that cannot have an equivalent in the practice of that engineering domain. Following the example, the engineer could not store a model with a

column that was not part of a table, because the composition relationship enforces this behavior. This level is known as meta-level one and we will call it the model level.

If the engineering language actually describes physical artefacts, then a model can have a correspondence in the real world. For example, for the model of a car, one could point to an actual car in the parking lot. Or alternatively, for the modelling of a non-physical, but still existing, software library, one could identify and point to a software library installed inside an actual computer. This meta-level zero is also known as the real-world level. It is often represented in literature, but is actually an artifact of the underlying philosophy of class-based thinking, which implies instantiation.

Figure 3 aims to summarize the four levels by giving examples of popular models[1] at different levels. The original MOF was developed more then ten years ago for server-side use, but recent developments in the last five years have made its desktop use ubiquitously feasible. The Eclipse Modeling Framework (EMF) is the most recent and arguably most popular variant of MOFand is well-integrated with the Eclipse IDE.

4.2 EMF Offers Generated User Interfaces

In addition to the manipulation and storage API generated by all MOF products, EMF provides a library for interactive manipulation that plugs directly into the Eclipse IDE. This substantially reduces on the time it takes to create an interactive graphical editor for a specific DSL. For this reason the current implementation of ModelicaML already uses EMF as its basis.

In addition, EMF offers a related facilities knowns as the Graphic Modelling Framework (GMF). GMF allows the definition of diagrammatic editors in a model-based fashion. In other words, the tool-designer creates a model of how the diagram editor should look for a specific DSL, and the generator builds the diagrammatic interface. For our example, Tables could be defined as being shown as boxes, Attributes as text lines within the boxes and Keys as arrows pointing among the Attributes.

4.3 EMF has the largest base of reusable editors

As a result the number of (graphical) editors built on EMF is far greater than that of any previous MOF product. This also affects the availability of developers that can create Modelica integrations, if ModelicaML is based on the same platform.

4.4 EMF has the most Models

Because there are more EMF-based tools in actual use, the number of models transitively defined based on EMF via different meta-models also outnumbers those in any other MOF product. The atlantic EMF model zoo is a great example of this http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/ We hence have a good base of engineering models that could be connected to Modelica models.

4.5 EMF can Glue Large (Meta-) Models easily

In addition to this, the architecture of EMF contains a design feature that resolves an issue that plagued previous MOF products. How do you integrate elements of two pre-existing meta-models? The OMG MOF standards do not address this issue, as it is seen as a technicality. Earlier MOF products required the meta-models to be loaded and all references resolved before models could be created. This practically limited the use of large combined models; EMF is lazy and knows the concept of a proxy, which only resolves if necessary. EMF also introduces a mapping between external resources and the model contents based on the concept of a Uniform Resource Identifier (URI). Exploiting this EMF can glue heterogenous models: A model stored in several files appears as one structure to the user of the model interface API. In this way, existing model files can be linked to new model files that reference them. This simplifies version control, as it avoids duplicate storage of data.

4.6 Other MOF Versions Do not Offer as much MDA

The critical mass of EMF use has fostered the growth of general facilities for the manipulation of models. For example, given a model of a building's floor plan and a model of a wiring plan for buildings, it would be feasible to derive a partial wiring plan from the floor plan. This model transformation could be written as a special-purpose Java program using nested iteration loops, but maintenance effort of this solution would be high. In databases, the maintenance problem has led to the development of SQL-DML, where the desired manipulation to the table entries are declaratively expressed by referencing the defined table structure. Within the MOF philosophy a generalizing approach can be adopted to create a general purpose model transformation language that allows to express the desired manipulation of instance models with reference to their asscoiated meta-models.

With such general facilities at hand the development of specialised engineering tools turns from a programming task into the art of defining a set interrelated models, their relationships and connecting model transformations and their visualizations. This idea of small interrelated models is at the heart of the Model-Driven Architecture (MDA) [15, 35]. However, general facilities are expensive tools to build, so the practice of MDA is strongly linked to EMF and its critical mass.

For the reasons presented, we see EMF as the best basis for the representation and integration of Modelica models.

[1] The diagram also clarifies why different versions of the popular modeling language UML cannot actually be exchanged, leading to a breakup in the space of artefacts. The next section will treat UML in more detail.

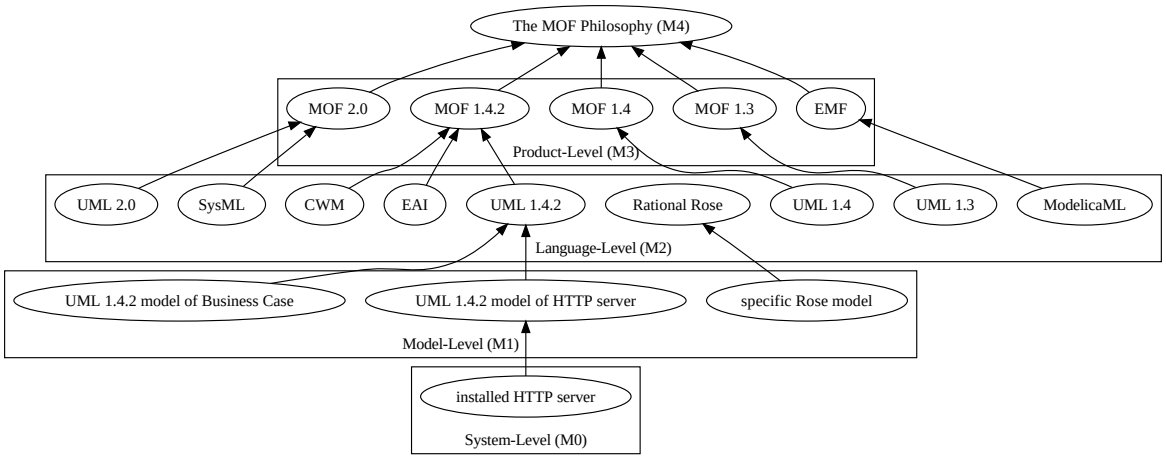


Figure 3. Four layer metamodel pyramid.

5. Why should ModelicaML not be UML?

When people talk about modeling in the context of software engineering they very quickly turn to talk about the Unified Modeling Language (UML). What they usually mean, is the language of class diagrams which only accounts for a very small part of the of the UML. UML comprises a number of modeling languages; It was conceived as a vehicle for the reuse of object-oriented models and a commercially-motivated merging approach to the three competing OO-modeling philosophies of Booch, Rumbaugh and Jacobsen. UML has been very successful in unifying the different graphical notations of these languages; Today's software engineers will always represent class is a rectangle with three partitions: name, attributes and methods.

In the following subsections we will present a number of arguments why ModelicaML should not reuse UML, but rather be a small well-defined core.

5.1 UML Standard Models are not Available, Exchangeable or Reusable

UML was conceived independently of MOF. Consequently UML tools were not built on any MOF core. They were implemented directly ad hoc. As a result, the semantics, the file formats and the APIs of the model manipulation facilities all became different. Since the tools themselves were expensive and specialized, there was little incentive for third parties to produce extensions that would allow the use of models for any purpose in the further development process. Hence the models were cut off from the rest of the development process and remained artifacts of documentation, typically created in the initial phases of projects. Consequently, errors found in later phases of projects were not fed back into the models and the quality of the models remained low. Finally, due to the high cost of their creation, models were treated as expensive intellectual property and hence not made available outside companies using UML. Although UML's inventors used class diagrams to describe UML's semantics, and this approach allowed the use of MOF products for the implementation of UML tools, most UML tools are still custom-built; Low availability, exchangeability, and reuse is hence typical for UML tools these days. The tool list available at `http://galaxy.andromda.org/docs/case-tools.html` gives a good impression of this situation. While an exchange of UML models is defined in a standard called XMI, this standard only exchanges the model data, but not the corresponding diagrams, and the document format varies by UML version and MOF product. Consequently, the exchange of data among tools is almost impossible. We want ModelicaML models to be easily exchangeable and reusable; So these properties of UML are undesirable and a reason not to base ModelicaML on UML.

5.2 UML is too Big

UML is the result of a merger of three modelling approaches, combining their diagram techniques and modelling concepts. Even at the start UML was already a sizable specification. From version 1.3, the first OMG-authorized release, to the current version 2.1 the size of the specification has expanded almost fourfold to over 2000 pages. Figure 4 shows this development. As a result of its substantial size, the UML standards are very hard to implement. Developers simply get lost in the text that mixes models, semi-formal specification in Object Constraint Language (OCL) and informal text descriptions. Very often, developers implement the bits they think they understand, and ignore the rest.

5.3 UML is not defined for EMF

None of the UML standards released through the OMG reference EMF as the underlying MOF version. For UML2.0 an implementation for MOF exists, but it is not officially endorsed as compliant with the standard. Instead, the current version of UML is defined against a sub-standard of the MOF 2.0 standard known as Complete MOF (CMOF). CMOF has a number of complex features [2]. To this date, the author knows of no working implementations of CMOF

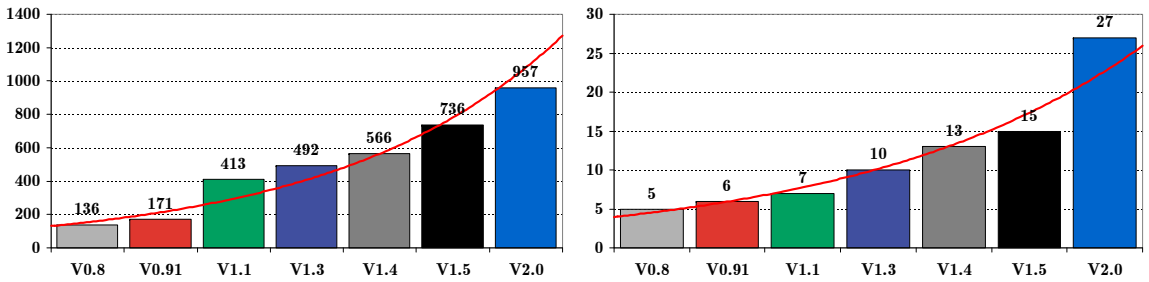


Figure 4. UML Version Development: size of the standard in pages and chapters.

or implementations of UML based on such a MOF variant. If ModelicaML is based on EMF, it cannot be based on the current specification version of UML at the same time.

5.4 UML is Semantically Unsound

Further, the UML standards shy away from defining several important technological matters, which would be necessary to make UML models reusable and exchangeable. For example, the standards do not successfully define what a valid UML model is, and at what point a UML tool would need to assert such validity. There is no test suite and no clearing house. The precise UML group, a think-tank of computer scientists, has worked for ten years on defining the precise meaning and consistency of UML and has yet to deliver a joint and clear statement. As Modelica requires precise semantics, it would suffer from problems of inconsistency and ambiguity, whenever a UML model was to be reused.

5.5 UML's Profiles are a Problem

The UML contains a mechanism known as "Profile". Profiles were originally defined as a mechanism to constrain the relatively weak semantics of UML through the application of a constraint language at the level of the UML meta-model. Unfortunately the Profile concept is often misused, in that additional semantics are added that are not necessarily compatible with UML.

According to the original definition of a profile as a constrained UML subset, the predicates of that constraint language would be evaluated against the content of a model. If a predicate was violated, the modeller would be provided with appropriate feedback. It is important to note, that this approach does *not change* the UML meta-models at all. For this reason, profiles are called light weight meta-model extensions. Hence the technical application of a profile implies that an OCL interpreter is present in every UML tool that can use a profile. The authors to this date know of no tool that satisfies this requirement. Consequently, anything called a profile is either relying on a proprietary tool extension, or it is a mere paper artifact, that does not actually support portable modeling.

The next problem with profiles is the desire of certain UML users to change, rather than to constrain, the meta-models of the UML. In other words, these users, often academics, want something that *looks* like UML, but is semantically incompatible with the UML metamodel . The profile becomes as a means of advertising one's own modelling language under the sales label of the UML. The UML standard version 2.0 vaguely describes mechanisms to alter the meta-model via a profile. There has been little interest in investigation or treatment of the resulting problems, as the mere size of UML 2.0 defeats a complete implementation in a tool anyway. During the closing panel of the last UML conference, the experts agreed that no tool was ever going to implement the UML 2.0 completely.

Today, profiles are mostly perceived as a means to describe visual alterations to UML diagram types. This use of profiles is even further removed from practical portable modeling, as there is no standardized algorithm that describes the allowable graphical renderings of a UML model in diagrams. Hence, there is also no portable amendment interface to this rendering algorithm. If profile-based alterations to rendering would be portable, they would need to be standardized parameters to the well-defined rendering algorithm. Hence even the use of amended diagram features is either proprietary or a suggestion on the manual use of drawing tools.

If ModelicaML was based on UML, it would need to implement a correct UML infrastructure including the ill-defined profile mechanism. This effort seems to be unjustified for the ModelicaML project, which aims at effective technical integration.

5.6 UML's Sublanguages are a Problem

In addition to the meta-model, the UML standard also includes two additional languages. The previously mentioned Object Constraint Language is used to define predicates against class diagrams and for pre- and post-conditions of methods. The Action Language is a slightly abstracted imperative language intended for a more precise description of behavior of systems. In order to be a sub-language of UML, ModelicaML would need to implement these other two languages as well. Both languages have little to do with Modelica's philosophy of equation-oriented modelling.

5.7 UML's children are not UML

Because standard-based UML is practically unmanageable in a tool for the reasons outlined above, vendors have begun to offer tools that offer reduced domain-specific languages

with well-defined syntaxes that are shown in diagram types resembling those of the UML. For the embedded systems domain, these are primarily class diagrams, sequence diagrams and state charts. In order to describe constraints and state transitions, the tools are often augmented with a textual constraint and action language.

Executable UML (xtUML) is such a sub-language. It revives and implements the OO-methodology of Shlaer-Mellor. This methodology was the fourth important contender at the time of UML's inception. However, it did not find its way into the commercial UML venture. xtUML is seeing increasing use in industry as a specification language. For example it is used for the specification of embedded controller software at Saab Bofors Dynamics AB (SB). Other UML sublanguages for system design are SysUML (Boeing, Saab) and RealTimeUML (RTUml) (Ericsson).

It is important to understand that none of these UML-named languages are actually UML since they are not based on CMOF or MOF. Their metamodels differ from that of the UML, and their models would not be interchangeable with standard-compliant UML tools or products, if these existed. They are proprietary engineering languages, branded "UML" for sales purposes. Apart from this misnomer, they can be used well for systems engineering and in collaboration with Modelica.

The current version of ModelicaML is based on SysML. As with any UML derivative, the metamodel of SysML is very large and contains a number of concepts that do not have a correspondence in Modelica. These elements do not form the focus of Modelica. They are superfluous and hence should not be part of ModelicaML. If they were kept, they would need to be fully expressed in diagrams, and their associated well-formedness rules from the SysML standard would need to be enforced. Further, the semantics of Modelica and SysML differ in some core areas. There, the SysML metamodel was altered to support Modelica semantics. As a result, standard SysML cannot be imported or exported from ModelicaML. Also, names of meta classes are all taken from SysML, even though the concepts may carry different names in Modelica. Consequently, there is a semantic mismatch and a long-time Modelica user will not easily find familiar concepts in the ModelicaML API, because they carry SysML names.

It seems advisable that the next version of ModelicaML should be defined via a meta-model that is as small as possible and independent of that of SysML or UML. Such consolidation will also help to improve quality, as the same amount of maintenance time will be applied to fewer artefacts and less code.

6. Usage Scenario and Proposal

The previous sections have outlined the next generation of ModelicaML as the integration interface of Modelica regarding EMF-based tools and IDEs. The revised ModelicaML will be based on EMF, but it will be smaller, and independent of SysML, UML and its children. What can we do with such an interface? How can it help Modelica to collaborate with other engineering languages?

The following sections examine a hypothetical scenario of collaboration between modelling tools using xtUML and Modelica. xtUML and Modelica have different strengths , and we will highlight these differences first. The rest of the section sketches a scenario around a real application of xtUML present at Saab Bofors.

6.1 xtUML and Modelica

xtUML describes state transitions of a model in the way most software engineers find natural: The state of the system is changed by explicitly defined actions and kept consistent by declarative constraints that should never be violated during its existence. Modelica on the other hand, due to its origin as a simulation framework, describes the behavior of a system's parts through equations. Apart from this, Modelica and xtUML know the same concepts of local attributes, generalization and aggregation, as they exist in all other object-oriented languages.

Summarily, xtUML is a language used to *explicitly* describe and drive the behavior of a system, Modelica is a language to *implicitly* describe and observe the behavior of a system. xtUML's strength is construction, Modelica's strength is analysis. Of course, both languages can be used in the respective other domain, but they will be less natural. For example, Modelica can also be used for expressing algorithmic or block-based controller code.

6.2 Missile Control at Saab-Bofors

Concretely, Saab-Bofors uses xtUML to describe the programs that drive the control of anti-aircraft missiles. The process is a typical application of MDA. It begins with a set of models related through model transformations and kept sound by validation procedures. Saab-Bofors uses an xtUML tool called Bridgepoint to create its models of the anti-aircraft missile software, validate it and to generate ADA program code that can be compiled into object-code that can be linked into an executable. The approach is special because software is flexibly apportioned to programmable hardware or controller software. The artifact flow is shown in an informal diagram in Figure 5.

6.3 Testing using Modelica

As stated above, Modelica is very useful for simulation. Testing of engineerng systems regularly involves building simulations of complex reactive environments of systems to explore system behaviour in different scenarios [34]. Among other things Modelica has been used in the past to simulate aerodynamic behavior of military aircraft. Modelica is special in comparison with other simulation systems, because Modelica can blend physical, electrical and electronic characteristics of a system seamlessly. So, Modelica lends itself well to designing and simulating the functions

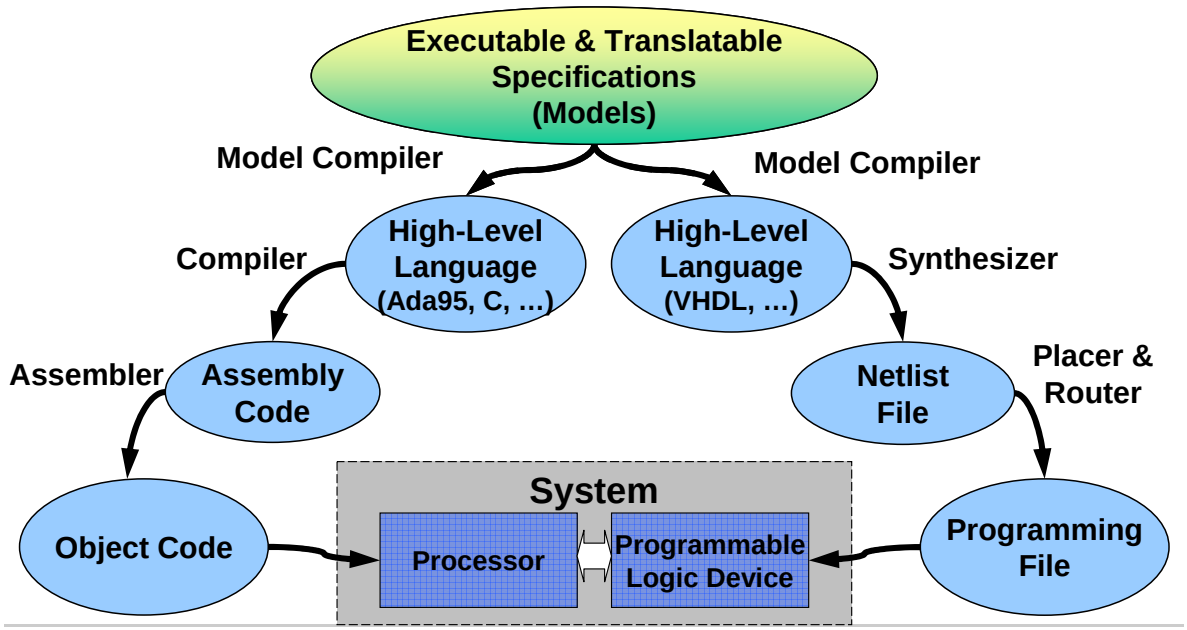


Figure 5. Model-Driven Process at Saab-Bofors. [36]

of a robotic arm, including torque, step motors, sensors, and control.

The following subsection describe three strategies for using Modelica with an engineering language like xtUML: Simulation of the environment, co-simulation of the subject-under-test, generation of simulation parameters, and implementation.

6.3.1 Linking Modelica to xtUML as an Environment

To integrate Modelica with a different modelling technique, the integration has to be modelled and rendered to executable code. Figure 6 shows one example of how xtUML and Modelica would collaborate: Bridgepoint is an Eclipse-based product, which is also internally built on EMF. As a result, its metamodels can be exported and referenced as part of other metamodels, implicitly making the data in the xtUML model available by navigation from other models. On the other side, a Modelica model of the environment of the missiles processor is prepared. This model only uses standard Modelica features. Its description uses the efficient diagrammatic features that Modelica users are familiar with. As is customary with Modelica, the model is translated into C code, which can also be compiled into object code for linking. Now, the features of the two software components have to interface. This connection information is encoded in a link-model that references features in the xtUML model and relates them to the corresponding features in the Modelica model. Now, a source code for the overall simulation can be generated, using this referential information. The source code is translated and the object codes of the environment simulator and missile control logic tied in. The resulting binary can then be run and will produce a simulation with good performance characteristics, due to the compilation of the code, as opposed to model interpretation. This approach can be extended for the special case of hardware-software splitting by providing a Modelica model of the programmable logic in addition to the processor model.

6.3.2 Modelica as Co-Simulator

In the previous scenario Modelica is used exclusively to describe the environment. However, Modelica could also be used to produce a specification of the expected test response: A sort of test oracle. This is also known as a peer model. A peer model is a model, that describes the SUT by different modelling means, in order to validate the behaviour. In the case of a peer model of Modelica for an xtUML component, the Modelica component describes the *expected* behaviour of the xtUML component in mathematical terms. The peer model can be used to check margins of error on the behaviour of the component, while going through the scenarios.

6.3.3 Modelica as Scenario Generator

Scenario coverage and test driving can be another target of Modelica modelling. This involves fashioning a model to describe what scenario initialisation data should be created and in which order the simulations should be run. In this context, Modelica is used to create mathematical models of the variant parameters.

6.3.4 Modelica as Implementation

Finally, Modelica could be used to provide implementations for components of interest. Modelica's semantics provide good clarity for all mathematical interactions, and its flow concepts allow natural modelling of component connections. This makes Modelica useful for the definition of components that resemble filters, pipes and streams. The Modelica standard is open and technically well-defined. As

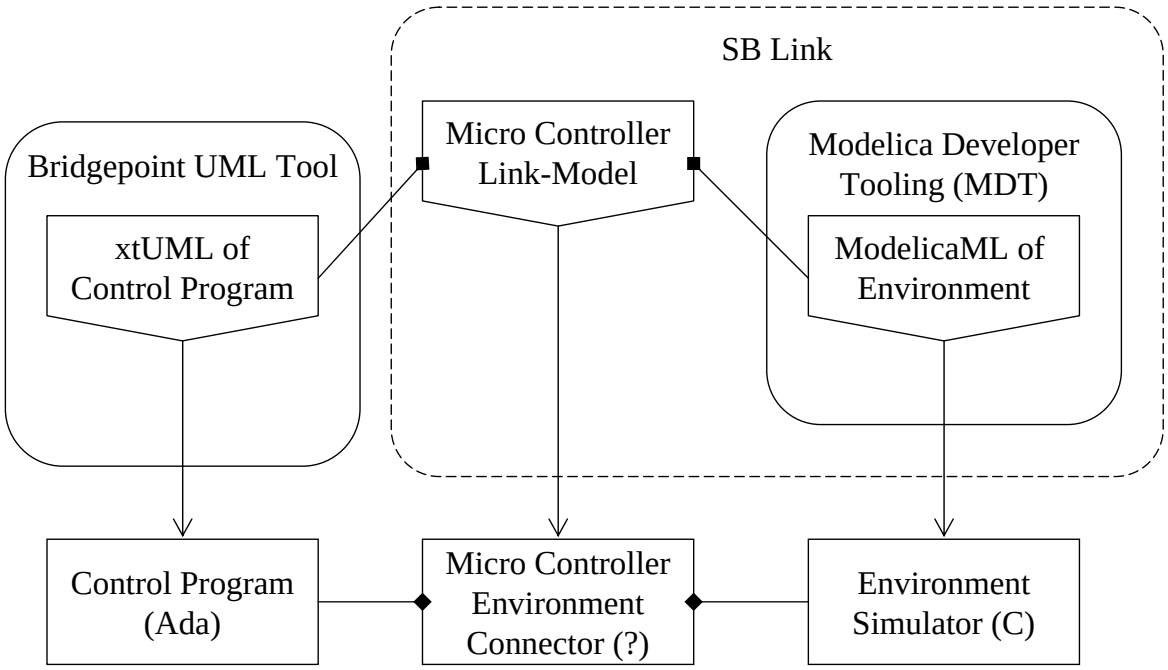


Figure 6. Link between ModelicaML and xtUML in the Saab-Bofors Scenario.

a consequence, model compilers can be written with confidence. For the Saab-Bofors scenario, part of the code for the driver application could be modelled in an imperative xtUML style, another part could be modelled in a reactive fashion in Modelica. The resulting object-codes could subsequently be linked and executed.

7. Summary and Outlook

In this paper we have reasoned about the further development strategy for ModelicaML and its core metamodel. We have argued that EMF is the the most effective choice as the implementation framework, but have discarded the use of full UML and its descendants and profiles for practical reasons. Instead, we have proposed an architecture based on a direct reflection of Modelica with a small footprint. Finally we have discussed four integration strategies in the context of the motivating example of the Saab-Bofors model-based workbench.

We will bring these considerations into the Modelica community as a basis for further discussion towards the standardization of the higher-level Modelica tooling and integration interface.

References

[1] David Akhvlediani. Design and implementation of a UML profile for Modelica/SysML. Technical Report LITH-IDA-EX–06/061-SE, Linköpings Universitet, April 2007. Final Thesis.

[2] Marcus Alanen and Ivan Porres. Difference and union of models. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML*, volume 2863 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2003.

[3] Marcus Alanen and Ivan Porres. Differences and Union of Models. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 2–17. Springer, 2003.

[4] Alex E. Bell. Death by UML Fever. *ACM Queue*, 2(1):72–80, March 2004.

[5] Fadi Chabarek. Development of an OCL Parser for UML Extensions. Diplomarbeit, Technical University Berlin, Computation and Information Structures, TU Berlin Fak.IV Franklinstraße 28/29 · D-10587 Berlin, March 2003.

[6] Dan Chiorean and Dragos Cojocari. Implementation of OCL Support in UML CASE Tools - the ROCASE Experience. Information Systems Modelling ISM '01, May 9 - 11, 2001 Hradec nad Moravicí, Czech Republic, November 2001.

[7] Andy Evans. Making UML Precise. In Luis Andrade, Ana Moreira, Akash Deshpande, and Stuart Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.

[8] Martin Fowler. What Is the Point of the UML? In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, page 325. Springer, 2003.

[9] Robert B. France, Sudipto Ghosh, Trung Dinh-Trong, and Arnor Solberg. Model-Driven Development Using UML 2.0: Promises and Pitfalls. *IEEE Computer*, 39(2):59–66, 2006.

[10] 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.

[11] Peter Fritzson and Peter Bunus. Modelica-A General Object-Oriented Language for Continuous and Discrete-Event System Modeling and Simulation. In *Annual Simulation Symposium*, pages 365–380. IEEE Computer Society, 2002.

[12] Anna Gerber and Kerry Raymond. MOF to EMF: there and back again. In Michael G. Burke, editor, *OOPSLA Workshop on Eclipse Technology eXchange*, pages 60–64. ACM, 2003.

[13] Martin Gogolla, Jean-Marie Favre, and Fabian Büttner. On Squeezing M0, M1, M2, and M3 into a Single Object Diagram. In Thomas Baar, Dan Chiorean, Alexandre Correa, Martin Gogolla, Heinrich Hußmann, Octavian Patrascoiu, Peter H. Schmitt, and Jos Warmer, editors, *Proc. MoDELS'2005 Workshop Tool Support for OCL and Related Formalisms*. In: Satellite Events at MoDELS'2005 Conference. Jean-Michel Bruel (Ed.). Springer, LNCS 3844. Long Version: EPFL (Switzerland), Technical Report LGL-REPORT-2005-001, 2005.

[14] Martin Gogolla and Brian Henderson-Sellers. Analysis of UML Stereotypes within the UML Metamodel. *Lecture Notes in Computer Science*, 2460:84–99, 2002.

[15] Object Management Group. *OMG Unified Modeling Language 2.0*. OMG, http://www.omg.com/uml/, 2005.

[16] The Precise UML Group. The Precise UML Group Homepage.

[17] Mario Jeckle. UML Profiles und sonstige UML-bezogene Aktivitäten. http://www.jeckle.de/uml_spec.htm, 2004.

[18] Mario Jeckle. Unified Modeling Language (UML) Tools. http://www.jeckle.de/umltools.html, 2004.

[19] Mario Jeckle, Chris Rupp, Barbara Zengler, Stefan Queins, and Jürgen Hahn. UML 2.0 - Neue Möglichkeiten und alte Probleme. *Informatik Spektrum*, 27(4):323–331, 2004.

[20] Cris Kobryn. UML 2001: A Standardization Odyssey. *Communications of the ACM*, 42(10):29–37, October 1999.

[21] Cris Kobryn. Will UML 2.0 be agile or awkward? *Commun. ACM*, 45(1):107–110, 2002.

[22] Haohai Ma, Weizhong Shao, Lu Zhang, Zhiyi Ma, and Yanbing Jiang. Applying OO Metrics to Assess UML Meta-models. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *UML 2004 - The Unified Modeling Language. Modeling Languages and Applications. 7th International Conference, Lisbon, Portugal, October 2004, Proceedings*, volume 3271 of *LNCS*, pages 12–26. Springer, 2004.

[23] OMG. *Requirements for UML Profiles*, 1.0 edition, June 1999.

[24] OMG. *Model Driven Architecture (MDA)*, July 2001.

[25] OMG. *Meta Object Facility(MOF) Specification*, April 2002. Version 1.4.

[26] OMG. *Unified Modeling Language Specification, Version 1.3*, March 2003.

[27] OMG. *Unified Modeling Language Specification, Version 1.4*, July 2004.

[28] OMG. *SysML*, May 2006.

[29] Adrian Pop, David Akhlevidiani, and Peter Fritzson. Towards Unified System Modeling with the ModelicaML UML Profile. In *EOOLT'2007*, Berlin, July 2007.

[30] Adrian Pop, David Akhvlediani, and Peter Fritzson. Integrated UML and Modelica System Modeling with ModelicaML in Eclipse. In *The 11th IASTED Int. Conf on Software Eng. and Appl. (SEA 2007)*, Cambridge, MA, USA, Nov 19-21 2007.

[31] Arnor Solberg, Robert France, and Raghu Reddy. Navigating the MetaMuddle. In *Proceedings of the 4th Workshop in Software Model Engineering (WiSME 2005)*, Montego Bay, Jamaica, 2005.

[32] Prawee Sriplakich, Xavier Blanc, and Marie-Pierre Gervais. Collaborative software engineering on large-scale models: requirements and experience in modelbus. In Roger L. Wainwright and Hisham Haddad, editors, *SAC*, pages 674–681. ACM, 2008.

[33] Jim Steele. UML2 gripes, May 2004. Blog on MOF2 inconsistencies. Snapshot on 11/19/04.

[34] Jörn Guy Süß, Adrian Pop, Peter Fritzson, and Luke Wildman. Towards integrated model-driven testing of scada systems using the eclipse modeling framework and modelica. In *Australian Software Engineering Conference*, pages 149–159. IEEE Computer Society, 2008.

[35] Jos Warmer. MDA Explained. Addison-Wesley, to appear, 2003.

[36] Erik Wedin. Model-Based Development of Embedded Systems with MDA and xtUML. Presentation at the MOD-PROD Workshop on Model-based Product Development at the University of Linköping, Sweden, Feb 2007.

Multi-Aspect Modeling in Equation-Based Languages

Dirk Zimmer

Institute of Computational Science, ETH Zürich, Switzerland, dzimmer@inf.ethz.ch

Abstract

Current equation-based modeling languages are often confronted with tasks that partly diverge from the original intended application area. This results out of an increasing diversity of modeling aspects. This paper briefly describes the needs and the current handling of multi-aspect modeling in different modeling languages with a strong emphasis on Modelica. Furthermore a small number of language constructs is suggested that enable a better integration of multiple aspects into the main-language. An exemplary implementation of these improvements is provided within the framework of Sol, a derivative language of Modelica.

Keywords language-design, object-oriented modeling

1. Motivation

Contemporary equation-based modeling languages are mostly embedded in graphical modeling environments and simulators that feature various types of data-representation. Let that be for instance a 3D-visualization or a sound module. Consequently the corresponding models are accompanied by a lot of information that describes abundantly more than the actual physical model. This information belongs to other aspects, such as the modeling of the iconographic representation in the schematic editor or the preference of certain numerical simulation techniques. Hence, a contemporary modeler has to cope with many multiple aspects.

In many modeling languages such kind of information is stored outside the actual modeling files, often in proprietary form that is not part of any standard. But in Modelica [6], one of the most important and powerful EOO-languages, the situation has developed in a different way. Although the language has been designed primarily on the basis of equations, the model-files may also contain information that is not directly related to the algebraic part. Within the framework of Modelica, the most important aspects could be categorized as follows:

- **Physical modeling:** The modeling of the physical processes that are based on differential-algebraic equations (DAEs). This modeling-aspect is also denoted as the primary aspect.
- **System hints:** The supply of hints or information for the simulation-system. This concerns for example hints for the selection of state-variables or start values for the initialization problem.
- **3D Visualization:** Description of corresponding 3D-entities that enable a visualization of the models.
- **GUI-Representation:** Description of an iconographic representation for the graphical user-interface (GUI) of the modeling environment.
- **Documentation:** Additional documentation that addresses to potential users or developers.

We will use this classification for further analysis, since it covers most of the typical applications fairly well. Nevertheless, this classification of modeling aspects is of course arbitrary, like any other would be.

Let us analyze the distribution of these aspects with respect to the amount of code that is needed for them. Figure 1 presents the corresponding pie-charts of three exemplary models of the Modelica standard library. These are the “FixedTranslation” component for the MultiBody-library, the PMOS model of the electrical package and the “Pump and Valve” model in the Thermal library. The first two of them represent single components; the latter one is a closed example system.

In the first step of data-retrieval, all unnecessary formatting has been removed from the textual model-files. For each of these models, the remaining content has then been manually categorized according to the classification presented above. The ratio of each aspect is determined by counting the number of characters that have been used to model the corresponding aspect.

The results reveal that the weight of the primary aspect cannot be stated to be generally predominant. The distribution varies drastically from model to model. It varies from only 14% to 53% for these examples.

Yet one shall be careful by doing an interpretation of the pie-charts in figure 1. The weight of an aspect just expresses the amount of modeling code with respect to the complete model. This does not necessarily correlate with the invested effort of the modeler and even less it does correlate with the overall importance of an aspect. It needs to be considered that code for the GUI-representation is mostly computer-generated code that naturally tends to be lengthy. On the other hand side, the code that belongs to the primary aspect of equation-based modeling is often surprisingly short. This is due to the fact that this represents the primary strength of Modelica. The language is optimized to those concerns and enables

2nd International Workshop on Equation-Based Object-Oriented Languages and Tools. July 8, 2008, Paphos, Cyprus.

EOOLT 2008 website: http://www.eoolt.org/2008/

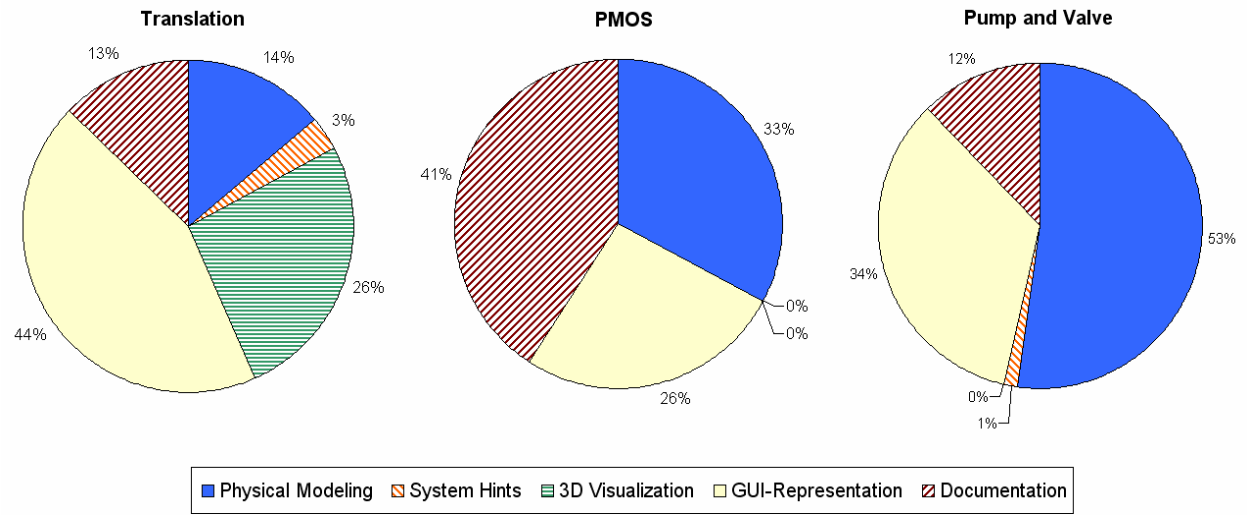


Figure 1. Code-Distribution of aspects in Modelica-models

convenient and precise formulations. Unfortunately, this can hardly be said about the other aspects in our classification.

The discussion about the Modelica and other EOO-language is often constrained to its primary aspect of physical modeling. But in typical models of the Modelica standard-library this primary aspect often covers less than 25% of the complete modeling code. Any meaningful interpretation of figure 1 reveals that the disregard on other modeling aspects is most likely inappropriate - especially when we are concerned with language design. For any modeling language that owns the ambition to offer a comprehensive modeling-tool, the ability to cope with multiple aspects has become a definite prerequisite.

It is the aim of this paper to improve modeling languages with respect to these concerns. To this end, we will suggest certain language constructs that we have implemented in our own modeling language: Sol. The application of these constructs will be demonstrated by a small set of examples. But first of all, let us take a look at the current language constructs in Modelica and other modeling languages.

2. Current handling of multiple aspects

2.1 Situation in VHDL-AMS, Spice, gPROMS, Chi

The need for multiple aspects originates primarily from industrial applications. Hence this topic is often not concerned for languages that have a strong academic appeal. One example for such a language is Chi [3]. For the sake of simplicity and clarity, this language is very formal and maintains its focus on the primary modeling aspect.

In contrast, languages like SPICE3 [9] or VHDL-AMS [1,10] and Verilog-AMS[12] are widely used in industry. Unlike Modelica, these languages do typically not integrate graphical information into their models. The associated information that describes the schematic diagram and the model icons is often separately stored, often in a proprietary format. For instance, the commercial product Simplorer [11] generates its own proprietary files for the model-icons. The corresponding VHDL-code does not relate to these files.

However, different solutions are possible: both AMS-languages contain a syntax-definition for attributes. These can be used to store arbitrary information that relate to certain model-items. Since there is only a small-number of predefined attributes (as unit descriptors, for instance), most of the attributes will have to be specified by the corresponding processing tools.

Furthermore these two languages and SPICE3 own an extensive set of predefined keywords. This way it is possible to define output variables or to configure simulation parameters. The situation is similar in ABACUSS II [5], which is the predecessor to gPROMS [2]. This language offers a set of predefined sections that address certain aspects of typical simulation run like initialization or output.

2.2 Multiple aspects in Modelica

The Modelica language definition contains also a number of keywords that enable the modeler to describe certain aspects of his model. For instance, the attributes **`stateSelect`** or **`fixed`** represent system-hints for the simulator. In contrast to other modeling languages, Modelica introduced the concept of annotations. These items are placed within the definitions of models or the declarations of members and contain content that directly relates on them. Annotations are widely used within the framework of Modelica. The example below presents an annotation that describes the position, size and orientation of the capacitor icon in a graphic diagram window.

```
Capacitor C1(C=c1) "Main Capacitor"
       annotation (extent=[50,-30; 70,-10],
       rotation=270);
```

Example 1. Use of an annotation in Modelica.

Since annotations are placed alongside the main modeling code, they inflate the textual description and tend to spoil the overall clarity and beauty. A lot of annotations contain also computer-generated code that hardly will be interesting for a human reader. Thus, typical Modelica-

editors mostly hide annotations and make them only visible at specific demand of the user. However, this selection of code-visibility comes with a price. First it reduces the convenience of textual editing, since cut, copy and paste operations may involve hidden annotations. Second, the selection of visibility happens on a syntactical level not on a semantic level.

Storing data for GUI-representation or other specific hints and information has been initially a minor topic in the design process of Modelica. Still, there was a compelling need for it. To meet these urgent requirements, the Modelica community decided to introduce the concept of annotations into the modeling language. Already the first language definition of Modelica contained the concept of annotations and also presented some applications for GUI-representation and documentation. The corresponding annotations have been used as a quasi-standard despite the fact that they only have been weakly documented. Annotations served also as an official back-door entrance to non-official, proprietary functionalities. Since it happens frequently in software engineering that certain things just grow unexpectedly, many further annotations have been introduced meanwhile. Nowadays, annotations contain a lot of crucial content that revealed to be almost indispensable for the generation of effective portable code. Therefore it is no surprise that just recently a large set of annotations had to be officially included in version 3 of the Modelica language definition [8]. This way, what started out as a small, local and semi-proprietary solution, became now a large part in the official Modelica standard.

To store the information that belongs to certain aspects, different approaches are used in Modelica and often more than one language-tool is involved. The following list provides a brief overview on the current mixture of data-representation:

- The physics of a model is described by DAEs and is naturally placed in the main Modelica model.
- Hints or information for the simulation-system are mostly also part of the main Modelica language but some of them have to be included in special annotations.
- Information that is used by the GUI is mostly included in annotations. But the GUI uses also uses information from textual descriptions that are part of the main-language.
- The description of 3D-visualization is done by dummy-models within main-Modelica code.
- Documentation may be extracted from the textual descriptions that accompany declarations and definitions, but further documentation shall be provided by integrating HTML-code as a text-string into a special annotation. Other annotations store information about the author and the library version.

2.3 Downfalls of the current situation

Obviously, this fuzzy mixture of writings and language constructs reveals the lack of a clear, conceptual approach. As nice as the idea of annotations appears in the first moment, it also incorporates a number of problematic insufficiencies.

The major drawback is that only pre-thought functionalities are applicable. The modeler has no means to define annotation by its own or to adapt given constructs to his personal demands. Furthermore, syntax and semantics of each annotation needs to be defined in the language definition. Since there is always a demand for new functionalities, the number of annotations will continue to increase. This leads to a foreseeable inflation of the Modelica language definition.

2.4 Lack of expressiveness

These downfalls originate from a lack of expressiveness in the original Modelica language. Whenever one is concerned with language design [7], it is important to repetitively ask some fundamental questions. How can it be that a language so powerful to state highly complicated DAE-systems is unable to describe a rectangle belonging to an iconographic representation? Why do we need annotations at all?

These questions are clearly justified and point to the fact that the development scope of the Modelica language might have been too narrowly focused on the equation based part. Therefore, extension that would have been of great help in other domains, have been left out:

1. There is no suitable language construct that enables the declaration of an interface to an environment that corresponds to a certain aspect.
2. Instances of objects cannot be declared anonymously within a model.
3. The language provides no tool for the user that enables him or her to group statements into semantic entities.
4. The language offers no means to refer on other (named) objects. Neither statically nor dynamically.

By removing these four lacks, we will demonstrate that the use of annotations can be completely avoided and that the declarative modeling of multiple aspects can be handled in a conceptually clear and concise manner. The following section will discuss this in more detail and provide corresponding examples.

3. Multi-aspect modeling in Sol

Sol is a language primarily conceived for research purposes. It owns a relatively simple grammar (see appendix) that is similar to Modelica. Its major aim is to enable the future handling of variable-structure systems. To this end, a number of fundamental concepts had to be revised and new tools had to be introduced into the language. The methods that finally have become available

suit also a better modeling of multiple aspects. These methods and their application shall now be presented.

3.1 Starting from an example

In prior publications on Sol [13,14] the "Machine" model has been introduced as standard example. It contains a simple structural change and consists of an engine that drives a flywheel. In the middle there is a simple gear box. Two versions of an engine are available: The first model `Engine1` applies a constant torque. In the second model `Engine2`, the torque is dependent on the positional state, roughly emulating a piston-engine. Our intention is to use the latter, more detailed model at the machine's start and to switch to the simpler, former model as soon as the wheel's inertia starts to flatten out the fluctuation of the torque. This exchange of the engine-model represents a simple structural change on run-time.

```
model Machine
implementation:
  static Mechanics.FlyWheel F{inertia<<1};
  static Mechanics.Gear G{ratio<<1.8};
  dynamic Mechanics.Engine2 E {meanT<<10};

  connection c1(a << G.f2, b << F.f);
  connection c2(a << E.f, b << G.f1);
  when F.w > 40 then
    E <- Mechanics.Engine1{meanT << 10};
  end;
end Machine;
```

Example 2. Simple machine model in Sol.

The first three lines of the implementation declare the three components of the machine: fly-wheel, gear-box and the engine. The code for the corresponding connections immediately follows. The third component that represents the engine is declared dynamically. This means that the binding of the corresponding identifier to its instance is not fixed and a new instance can be assigned at an event. This is exactly what happens in the following declaration of the when-clause. A new engine of compatible type is declared and transmitted to the identifier E. The old engine-model is thereby implicitly removed and the corresponding equations are automatically updated.

This model contains the physics part only. We now want to add other aspects to the model. We would like to add a small documentation and to specify the simulation parameters. Furthermore we want to add information about model's graphical representation in a potential, graphical user-interface. The following sub-sections will present the necessary means and their step by step application.

3.2 Environment packages and models

Many modeling aspects refer to an external environment that is supposed to process the exposed information. This environment may be the GUI of the modeling environment or a simulator program. Therefore it needs to be specified how a model can address a potential environment. To this end, Sol features environment packages and models that enable to define an appropriate interface. Let us take a look at an example:

```
environment package Documentation

        model Author
        interface:
              parameter string name;
        end Author;

        model Version
        interface:
              parameter string v;
        end Version;

        model ExternalDoc
        interface:
              parameter string fname;
        end ExternalDoc;

end Documentation
```

Example 3. Environment package.

This example consists in a package that contains models which can be used to store relevant information for the documentation of arbitrary models. The keyword **`environment`** does specify that the models of the corresponding package address the environment and are therefore not self-contained. They merely offer an interface instead. The actual implementation and semantics of the package remains to be specified by the environment itself.

It is important to see that stipulating the semantics would be a misleading and even futile approach. Different environments will inevitable have to feature different interpretations of the data. For instance, a pure simulator will complete ignore the "Documention" models whereas a modeling editor may choose to generate an HTML-code out of it. Nevertheless it is very meaningful to specify a uniform interface within the language. This provides the modeler with an overview of the available functionalities. Furthermore the modeler may choose to customize the interface for its personal demands using the available object-oriented means of the Sol-language.

3.3 Anonymous Declaration

The language Sol enables the modeler to anonymously declare models anywhere in the implementation. The parameters can be accessed by curly brackets whereas certain variable members of the model's interface are accessible by round brackets. This way, the modeler can address its environment in a convenient way just by declaring anonymous models of the corresponding package. An application of this methodology is presented below in example 4 for the Machine model.

Anonymous declarations are an important element of Sol, since they enable the modeler to create new instances on the fly, for example at the execution of an event. This is very helpful for variable-structure systems. However, within the context of multi-aspect modeling, anonymous declarations serve primarily convenience. It is of course possible to assign names to each of the documentation items. They can be declared with an identifier like any other model, but this is mostly superfluous and would lead to bulky formulations.

```
model Machine
implementation:
  […]
  when F.w > 40 then
    E <- Mechanics.Engine1{meanT << 10 };
  end;

  Documentation.Author{name<<"DirkZimmer"};
  Documentation.Version{v << "1.0");
  Documentation.ExternalDoc
  {fname<<"MachineDoc.html"};

end Machine;
```

Example 4. Use of anonymous declarations.

3.4 Model sections

Sol has been extended by the option for the modeler to define sections using an arbitrary package name. Sections incorporate three advantages: One, code can be structured into semantic entities. Two, sections add convenience, since the sub-models of the corresponding package can now be directly accessed. Three, section enable an intuitive control of visibility. A modern text editor may now hide uninteresting sections. The user may then be enabled to toggle the visibility according to its current interests. This way, the visibility is controlled by semantic criteria and not by syntactical or technical terms.

```
model Machine
implementation:
  […]
  when F.w > 40 then
    E <- Mechanics.Engine1{meanT << 10 };
  end;

  section Documentation:
    Author{name << "Dirk Zimmer"};
    Version{v << "1.0"};
    ExternalDoc{fname<<"MachineDoc.html"};
  end;

  section Simulator:
    IntegrationTime{t << 10.0};
    IntegrationMethod{method<<"euler",
    step << "fixed", value << 0.01};
  end;

end Machine;
```

Example 5. Sections.

The documentation part of the machine model has now been wrapped within a section. A second section addresses another environment called "Simulator" and shows an exemplary specification of some simulation parameters. Both sections could be hidden by an editor if the user has no interest in their content.

3.5 Referencing of model-instances

The provided methods so far, are fully sufficient for simple application cases. The proper implementation of a GUI-representation is yet a more complex task that demands a more elaborate solution.

In the classic GUI-framework for object-oriented modeling, each model owns an icon and has a diagram window that depicts its composition from sub-models. Figure 2 displays the aspired diagram of the exemplary machine-model that contains the icons of its three sub-models. The connections are represented by single lines. The following paragraphs outline one possible solution in Sol.

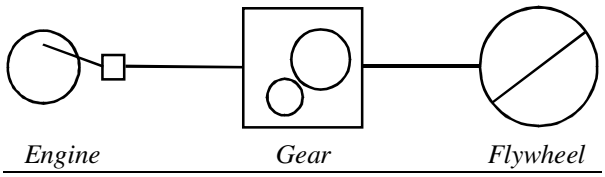


Figure 2. The diagram representation.

The problem is that many models will own GUI-information but only the information of certain model-instances shall be acquired. This originates in the need for language constructs that enable hierarchical or even mutual referencing between model-instances. Sol meets these requirements by giving model-instances a first-class status [4]. This means that model-instances cannot only be declared anonymously but also these instances can be transmitted to other members or even to parameters.

This capability had already been applied in example 2 to model the structural change of the engine. The statement

```
E <- Mechanics.Engine1{meanT << 10};
```

declares anonymously an instance of the model "Engine1" and then transmits this instance to the dynamic member E. Hence the binding of the identifier to its instance gets re-determined which causes a structural change.

A similar pattern will occur in our solution for the GUI-design. Let us take a look at the corresponding environment-package.

- **environment package** Graphics
  - **model** Line
  - **model** Rectangle
  - **model** Ellipse
    - **model** Canvas
      - **model** Line
      - **model** Rectangle
      - **model** Ellipse
  - **model** GraphicModel

Figure 3. Structure of the "Graphics" package.

Figure 3 gives a structural overview of the environment-package **Graphics**. This package provides rudimentary tools for the design of model-icons and diagrams. These are represented by models for rectangles, ellipses and lines. The package contains also a **Canvas** model that enables drawings on a local canvas. Furthermore the package contains a partial model **GraphicModel** that serves as template for all models that support a graphical GUI-representation. It defines two sub-models: one for

the icon-representation and one for the diagram representation. Models that own a graphical representation are then supposed to inherit this template model. Please note that the icon has a canvas model as parameter.

```
model GraphicModel
interface:
       model Icon
       interface:
              parameter Canvas c;
       end Icon;

       model Diagram
       end Diagram;

end GraphicModel;
```

Example 6. A template for graphical models.

A graphical modeling environment may now elect to instantiate one of these sub-models. This will cause further instantiations of models belonging to the "Graphics"-package that provide the graphical environment with the necessary information. Below we present an exemplary icon model for our engine that corresponds to the icon in Figure 2.

```
model Engine2 extends Interfaces.OneFlange;
              //that extends GraphicalModel
interface:

  parameter Real meanT;

  redefine model Icon
  implementation:
       c.Ellipse(sx<<0.0,sy<<0.2,
                 dx<<0.6,dy<<0.8);
       c.Rectangle(sx<<0.9,sy<<0.45,
                   dx<<1.0,dy<<0.55);
       c.Line(sx<<0.3,sy<<0.3,
              dx<<0.9,dy<<0.5);
  end Icon;

implementation:

  […]

end Engine2;
```

Example 7. An implementation of an icon

The icon of example 7 "paints" on a local canvas that is specified by the corresponding parameter c. The transmission of this parameter is demonstrated in Example 8 that represents the whole diagram of figure 2. This model declares the icons of its sub-models and creates a local canvas for each of them by an anonymous declaration. The two connections c1 and c2 also own a Line-model for their graphical representation.

```
model Machine
  extends Graphics.GraphicalModel;
interface:

  redefine model Diagram
  implementation:
  section Graphics:
    F.Icon{c<<Canvas{x<<10,y<<10,
                     w<<10,h<<10}};
    G.Icon{c<<Canvas{x<<30,y<<10,
                     w<<10,h<<10}};
    E.Icon{c<<Canvas{x<<50,y<<10,
                     w<<10,h<<10}};
    c1.Line(sx<<20,sy<<15,
            dx<<30,dy<<15);
    c2.Line(sx<<40,sy<<15,
            dx<<50,dy<<15);
    c.Rectangle(0,0,70,30);
  end;
  end Diagram;

implementation:

  […]

  section Documentation:
  […]

  section Simulator:
  […]

end Machine;
```

Example 8. An implementation of a diagram

The "GraphicalModel" involves another key-concept of Sol. The language enables the modeler to define models also as member-models in the interface section. When instantiated, these models belong to their corresponding instance and are therefore not independent. This means that the Diagram or Icon model always refer to their corresponding super-instance. Consequently, they also have access to all the relevant parameters and can adapt.

Please note, that the resulting GUI-models are potentially much more powerful than their annotation-based counterparts in Modelica. All the modeling power of Sol is now also available for the graphical models. For instance, only a minimal effort is needed to make the look of an icon adapt to the values of a model-parameter. No further language construct would be required. A model could even feature "active" icons that display the current system-state and hence enable a partial animation of the system within the diagram-window. Even the structural change of the machine-model could be made visible in the diagram during the simulation. Such extensions (if desired or not) become now feasible and demonstrate the flexibility of this approach.

However, the provided examples are merely a suggestion and represent just one possible and convenient solution within the framework of Sol. There are also many other language constructs that would lead to feasible or even more general solutions. Many of them could easily be integrated into equation-based languages. Some of them are featured in Sol. With respect to Modelica, this is unfortunately not the case yet.

4. Conclusion

Handling complexity in a convenient manner and organizing modeling knowledge in a proper form have always been primary motivations for the design of modeling languages. The introduction of object-oriented mechanism has yield to a remarkable success and drastically simplified the modeling of complex systems. Object-orientation essentially enabled the modeler to break models into different levels of abstraction. Hence, the knowledge could be organized with respect to depth.

However, certain models combine many different aspects that have to be linked together at a top level. Here the knowledge needs to be organized with respect to breadth. For those tasks, current mechanisms in EOO-languages are underdeveloped.

This paper focuses on **four conceptual language constructs** for EOO-languages that in combination drastically increase the ability to deal with multiple aspects. These are:

1. **Environment-packages** that enable the aspect-specific declaration of interfaces.
2. **Anonymous declarations** of model instances.
3. **Sections** can be used to form semantic entities and control visibility.
4. **Referencing mechanisms** between model-instances. (In Sol, these mechanisms are provided by giving model-instances a first class status and enabling so-called member-models.)

The proposed constructs have been implemented in our experimental language Sol and their application is demonstrated by a set of corresponding examples. The resulting advantages of this approach are manifold:

- The methods how to address a potential environment are made available within the language. The modeler may browse through the provided functionalities like she or he is used to do it for standard libraries.
- The existing object-oriented mechanisms can be applied on these environment-models. Hence the modeler can customize the interface for its personal demands and is not constrained to a predefined solution.
- Anonymous declarations enable a convenient usage of these models, anywhere in the implementation. The resulting statements are naturally readable and integrate nicely into the primary, equation-based part.
- User-defined sections help to organize the model and offer an excellent way to filter for certain modeling aspects. Uninteresting information may consequently be hidden without hindering the editing of the code. The filtering criteria are not based on syntax anymore, there are based on semantic entities that have been formed by the modelers themselves. Furthermore sections enable a clear separation of computer generated modeling code.
- The embedment into an existing object-oriented framework enables a uniform approach for a wider range of modeling aspects. Furthermore, it increases the interoperability between these aspects.

However, the most important conclusion is that the ability of the language to help and to extend itself by its own means has been drastically improved with respect to other languages like Modelica. Further development is now possible within the language does not require a constant update and growth of the language definition.

Appendix

The following listing of rules in extended Backus-Naur form (EBNF) presents an updated version of the core grammar for the Sol modeling language. The rules are ordered in a top-down manner listing the high-level constructs first and breaking them down into simpler ones. Non-terminal symbols start with a capital letter and are written in bold. Terminal symbols are written in small letters. Special terminal operator signs are marked by quotation-marks. Rules may wrap over several lines.

The inserted modifications concern solely the modeling of multiple aspects. With respect to a prior version of the grammar [13], the changes are minor and concern only 3 rules: **ModelSpec**, **Statement** and **Section**.

Listing 1: EBNF-Grammar of Sol

```
Model       =  ModelSpec Id Header
               [Interface] [Implemen] end Id ";"
ModelSpec   =  [redefine | partial | environment]
               (model | package | connector | record)

Header      =  {Extension} {Define} {Model}
Extension   =  extends Designator ";"
Define      =  define (Const | Designator) as Id ";"

Interface   =  interface ":" {(IDecl | ParDecl) ";"} {Model}
ParDecl     =  parameter Decl
IDecl       =  [redelcare]  LinkSpec [IOSpec] [CSpec] Decl
ConSpec     =  potential | flow
IOSpec      =  in | out

Implemen    =  implementation ":" StmtList
StmtList    =  [Statement {";" Statement }]
Statement   =  [ Section | Condition | Event |
                 Declaration | Relation ]

Section     =  section Designator ":" StmtList end [section]
Condition   =  if Expression then StmtList ElseCond
ElseCond    =  (else Condition) | ([else then StmtList] end [if])
Event       =  when Expression then StmtList  ElseEvent
ElseEvent   =  (else Event)|([else then StmtList] end [when]
Declaration =  [redeclare] LinkSpec Decl
LinkSpec    =  static | dynamic
Decl        =  Designator Id [ParList]

Relation    =  Expression Rhs
Rhs         =  ("=" | "<<" | "<-")  Expression

ParList     =  "{" [Designator Rhs {"," Designator Rhs }]  "}"
InList      =  "(" [Designator Rhs {"," Designator Rhs }]  ")"
```

```
Expression  =  Comparis {(and|or) Comparis }
Comparis    =  Term [("<"|"<="|"=="|"<>"|">="|">")Term]
Term        =  Product {( "+" | "-" ) Product }
Product     =  Power { ("*" | "/")  Power }
Power       =  SElement {"^" SElement }
SElement    =  [ "+" | "-" | not ] Element
Element     =  Const | Designator [InList] [ParList]
               | "(" Expression ")"

Designator  =  Id {"." Id }
Id          =  Letter {Digit | Letter}
Const       =  Number | Text | true | false
Number      =  ["+"|"-"] Digit { Digit }
               ["." {Digit }] [e ["+"|"-"] Digit { Digit }]
Text        =  "\"" {any character} "\""
Letter      =  "a" | ... | "z" | "A" | ... | "Z" | "_"
Digit       =  "0" | ... | "9"
```


Acknowledgements

I would like to thank Prof. Dr. François E. Cellier for his helpful advice and support. This research project is sponsored by the Swiss National Science Foundation (SNF Project No. 200021-117619/1).

References

[1] P.J. Ashenden, G.D. Peterson, and D.A. Teegarden. *The System Designer's Guide to VHDL-AMS* Morgan Kaufmann Publishers. 2002.

[2] P.I. Barton and C.C. Pantelides. Modeling of Combined Discrete/Continuous Processes. *American Institute of Chemical Engineers Journal*. 40, pp.966-979, 1994.

[3] D. A. van Beek and J.E. Rooda. Languages and Applications in Hybrid Modelling and Simulation: Positioning of Chi. *Control Engineering Practice*, 8(1), pp.81-91, 2000.

[4] Rod Burstall. Christopher Strachey – Understanding Programming Languages. *Higher-Order and Symbolic Computation* **13**:52, 2000.

[5] J.A. Clabaugh, ABACUSS II Syntax Manual, Technical Report. *Massachusetts Institute of Technology. http://yoric.mit.edu/abacuss2/syntax.html.* 2001.

[6] Peter Fritzson. *Principles of Object-oriented Modeling and Simulation with Modelica 2.1*, John Wiley & Sons, 897p. 2004.

[7] C.A.R. Hoare. Hints on Programming Language Design and Implementation. *Stanford Artificial Intelligence Memo*, Stanford, California, AIM-224, 1973.

[8] Modelica® - A Unified Object-Oriented Language for Physical Systems Modeling Language Specification Version 3.0. *Available at www.modelica.org* .

[9] Thomas L. Quarles. Analysis of Performance and Convergence Issues for Circuit Simulation. *PhD-Dissertation*. EECS Department University of California, Berkeley Technical Report No. UCB/ERL M89/42, 1989.

[10] Peter Schwarz, C. Clauß, J. Haase and A. Schneider. VHDL-AMS und Modelica - ein Vergleich zweier Modellierungssprachen. *Symposium Simulationstechnik ASIM2001*, Paderborn 85-94, 2001.

[11] Ansoft Corporation: Simplorer
Avaiable at: http://www.simplorer.com .

[12] Verilog-AMS Language Reference Manual Version 2.2 *Available at http://www.designers-guide.org/VerilogAMS/* .

[13] Dirk Zimmer. Introducing Sol: A General Methodology for Equation-Based Modeling of Variable-Structure Systems. *Proceedings of the 6th International Modelica Conference*, Bielefeld, Germany, Vol.1 47-56, 2008.

[14] Dirk Zimmer. Enhancing Modelica towards variable structure systems. *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools*, Berlin, Germany, 61-70, 2007.

Biography

Dirk Zimmer received his MS degree in computer science from the Swiss Federal Institute of Technology (ETH) Zurich in 2006. He gained additional experience in Modelica and in the field of modeling mechanical systems during an internship at the German Aerospace Center DLR 2005. Dirk Zimmer is currently pursuing a PhD degree with a dissertation related to computer simulation and modeling under the guidance of Profs. François E. Cellier and Walter Gander. His current research interests focus on the simulation and modeling of physical systems with a dynamically changing structure. He is also interested in topic of language-design and processing.

Beyond Simulation: Computer Aided Control System Design Using Equation-Based Object Oriented Modelling for the Next Decade

Francesco Casella[1] Filippo Donida[1] Marco Lovera[1]

[1]Politecnico di Milano, Dipartimento di Elettronica e Informazione, Italy
{casella,donida,lovera}@elet.polimi.it

Abstract

After 20 years since their birth, equation-oriented and object-oriented modelling techniques and tools are now mature, as far as solving simulation problems is concerned. Conversely, there is still much to be done in order to provide more direct support for the design of advanced, model-based control systems, starting from object-oriented plant models. Following a brief review of the current state of the art in this field, the paper presents some proposals for future developments: open model exchange formats, automatic model-order reduction techniques, automatic derivation of simplified transfer functions, automatic derivation of LFT models, automatic generation of inverse models for robotic systems, and support for nonlinear model predictive control.

Keywords Control system design, symbolic manipulation, model order reduction, CACSD.

1. Introduction

Control system engineering requires to master the dynamics of plants which are in general complex, interacting, multi-physics and multi-disciplinary. This explains why object-oriented modelling (OOM) and a-causal, equation-based, object-oriented languages (EOOL) always had a very strong connection with control system design. It is by no means accidental that much pioneering work in the OOM field was carried out within systems and control departments and research groups: consider, for example, the Omola language and the associated OmSim simulation environment, developed at the Department of Automatic Control of Lund Technical University [29, 30, 4], or the MOSES environment developed at the Dipartimento di Elettronica of Politecnico di Milano [26, 9]. During the '90, OOM was considered a very promising tool for Computer Aided Control System Design (CACSD), and there was a lot of activity in this field, which eventually culminated in the development of the Modelica Language [32].

At the beginning of that decade, papers appeared on the subject in the IEEE's Control Systems Magazine [31, 10], which discussed the potential of OOM for control system design. Reading those papers in retrospect shows that some of the promises where actually met or even exceeded: OOM is now a mature field, both from a theoretical side and from the point of view of available simulation tools. On the other hand, much work still has to be done on two fronts. The first one, which has a more "political" nature, is spreading the OOM culture among in the control engineering community, which is still largely dominated by block-oriented modelling, and by the (mis)use of Matlab/Simulink for physical systems modelling; this challenge is of paramount importance, but it out of the scope of this paper. The second one, instead, is to develop tools which allow to use EOOL models and tools not only for simulation, but also for the design of advanced control systems. The availability of such tools is crucial in order to narrow the gap between the large body of highly sophisticated control theory developed during the last 20 years, and the application of this theory to real-life cases, beyond textbook-sized examples. This is the topic of the present paper.

Given the background and the past experience of the authors, the discussion might be biased towards the Modelica language and related tools. However, strictly object-oriented features such as inheritance, encapsulation and hierarchical composition do not play any significant role in the analysis and proposals made within this paper, which essentially focuses on transformations of flattened models. On the contrary, the discussion is relevant for any equation-based modelling language, provided that it is a-causal and it allows symbolic manipulation of the equations by the compiler.

The paper is structured as follows: Section 2 gives a high-level view of the modelling activities required for control system design, while the following Section 3 discusses how currently available tools can help the control engineer in his/her task, with particular reference to Modelica tools. Sections 4 and 5, which are the core of the paper, propose several research and development directions to substantially increase the level of support to the control en-

2nd International Workshop on Equation-Based Object-Oriented Languages and Tools. July 8, 2008, Paphos, Cyprus.

gineer, willing to apply advanced control theory to real-life problems. Section 6 concludes the paper with final remarks.

2. The role of mathematical models in control system design

The design of control systems always requires some knowledge about the dynamic behaviour of the plant under control. When the plant design is mature and well-known, and the control system design is based on Proportional-Integral-Derivative (PID) controllers, the latter is often based on past experience and possibly on some empirical measurements. In this case, which covers the vast majority of installed industrial controllers, no (explicit) dynamical modelling is needed.

On the other hand, in an increasing number of cases, the performance of the control system is becoming a key competitive factor for the success of innovative, high-tech systems. To name a few examples, consider high-performance mechatronic systems (such as robots), vehicles enhanced by active integrated stability, suspension, and braking control, aerospace system, advanced energy conversion systems. All these cases possess at least one of the following features, which call for some kind of mathematical modelling for the design of the control system:

- closed-loop performance critically depends on the dynamic behaviour, which is not well-known in advance;
- the system is complex, made of many closely interacting subsystems, so that the behaviour of the whole system is more than just the sum of its parts;
- advanced control systems are required to obtain competitive performance, and these in turn depend on explicit mathematical models for their design;
- the system is very expensive and/or safety critical, requiring extensive validation of good control performance by simulation.

In most of these cases, two different classes of mathematical models are derived: compact models for control design and detailed models for system simulation.

2.1 Compact models for control design

Models belonging to this class are directly used for controller design, and are usually formulated in state-space form:

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t), p, t) \\ y(t) &= g(x(t), u(t), p, t) \end{aligned} \tag{1}$$

where x is the vector of state variables, u is the vector of system inputs (control variables and disturbances), y is the vector of system outputs, p is the vector of parameters, and t is the continuous time. A special case is that of linear, time-invariant models (LTI), which can be described as:

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned} \tag{2}$$

or, equivalently, as a transfer function:

$$G(s) = C(sI - A)^{-1}B + D. \tag{3}$$

In many cases, the dynamics of systems in the form (1) is approximated by (2) via linearization around some equilibrium point. There is also a vast body of advanced control techniques which are based on discrete-time models:

$$\begin{aligned} x(k+1) &= f(x(k), u(k), p, k) \\ y(k) &= g(x(k), u(k), p, k) \end{aligned} \tag{4}$$

where the integer time step k usually corresponds to the sampling time T_s of a digital control system. Many techniques are available to transform (1) into (4).

These models must capture the fundamental dynamics which is relevant for control system performance, while remaining as simple as possible: most advanced control design techniques start to become intractable for systems of order greater than about ten. If the models are simple enough, it is also sometimes possible to express the dependence of key dynamic features (such as, e.g., the natural frequency and damping coefficient of an oscillating dynamics) from plant design data. This can be very important to assess the impact of physical system design decisions on controller performance. For example, if the natural frequency of the first mode of oscillation limits the controller bandwidth, and it is found that this frequency mainly depends on the stiffness of a certain mechanical component, then it might be reasonable to change the mechanical design of that component in order to improve the overall performance.

In order to derive such simple models, it is usually necessary to introduce many, sometimes drastic, simplifying assumptions: all those phenomena that only marginally affect the equilibrium values and/or the control-relevant dynamics of the system are neglected. This activity requires highly skilled and experienced modellers, with a good knowledge of control design techniques, as well as of domain-specific strategies for model simplification.

2.2 Detailed models for system simulation

At the other end of the modelling spectrum, detailed simulation models can be found. Although it is always necessary to make reasonable modelling assumptions (a model is always a focused and limited description of the physical world), simulation models can include a lot more detail and second-order effects, since modern CPUs and simulation environments can easily handle complex systems with (tens of) thousands of variables. It is well-known that OOM methodologies and EOOLs provide very good support for the development of such models, thanks to equation-based modelling, a-causal physical ports, aggregation and inheritance. If the OOM model does not contain discrete variables and events, then it is basically equivalent to the set of DAEs:

$$F(x(t), \dot{x}(t), u(t), y(t), p, t) = 0 \tag{5}$$

Many EOOLs and tools also allow to describe hybrid systems, with discrete variables, conditional equations or expressions, and events. For example, see [7, 8] and references therein for hybrid system descriptions based on hybrid automata, or the Modelica language specification [41], in particular Appendix C. Although hybrid system control

is an interesting and emerging field, for the sake of conciseness this paper will focus on purely continuous-time physical models, with application to the design of continuous-time or sampled-time control systems.

These larger, more detailed models play a double role, with respect to those described in the previous sub-section. On one hand they allow to check how good (or crude) the compact models is, compared to a more detailed description, thus helping to develop good compact models. On the other hand, they allow to check the closed-loop performance of the controlled system, once a controller design is available. It is in fact well-known that validating the closed-loop performance using the same simplified model that was used for control system design is not a sound practice; conversely, validation performed with a more detailed model is usually deemed a good indicator of the control system performance, whenever experimental validation is not possible for some reason.

3. Overview of current CACSD practice with EOOLs

As of today, the practising control engineer already gets much support from EOOL-based tools for his/her control system design activities.

3.1 Support to control system synthesis

A typical starting point for the design of the control system is the analysis of the linearized dynamics of the plant, around one (or more) steady-state operating conditions. If the EOOL tool only supports simulation, then one can run open-loop simulations of the plant model, subject to step or to, e.g., pseudo-random binary sequence inputs, and then reconstruct the dynamics by system identification procedures.

A more direct approach, supported by many tools, is to directly compute the A, B, C, D matrices of the linearized system around specified equilibrium points, using symbolic and/or numerical techniques. The result is usually a high-order linear system, which can then be reduced to a low-order system by standard techniques for linear model order reduction, such as, e.g., balanced truncation.

A non-trivial issue with both approaches is the computation of the equilibrium point (what is sometimes called DC analysis in the field of electrical circuit simulation). In a typical setting, the desired steady-state values of the outputs $\bar{y}$ are known, and the tool must solve the steady-state initialization problem for the system (5):

$$F(\bar{x}, 0, \bar{u}, \bar{y}, p, 0) = 0 \tag{6}$$

in order to find out the corresponding equilibrium values of the inputs $\bar{u}$ and of the states $\bar{x}$. This problem can be numerically challenging, because it often requires solving large systems of coupled nonlinear equations by iterative methods, which might fail if the iteration variables are not properly initialized. Currently available OOM tools (and, in particular, Modelica tools) are still far from providing general robust solutions to this problem. A sub-optimal approach to find equilibrium points is to initialize system (5) by giving tentative initial values to the state variables (which makes the initialization problem easier to solve) and then to simulate it until it reaches a steady state. If the system is asymptotically stable and the inputs $\bar{u}$ are known, this is relatively straightforward; otherwise, it is necessary to add suitable feedback controllers to drive the outputs to the desired values $\bar{y}$ and/or to stabilize the system. In both cases, the simulation of this initialization transient might fail for numerical reasons before reaching the steady state, due to a bad choice of the initial states.

3.2 Closed-loop performance assessment by simulation

Regardless of the actual design methodology, once the controller has been set up, an OOM tool can be used to run closed-loop simulations, including both the plant and the controller model. Many OOM tools provide model export facilities, which allow to connect an OO plant model with only causal external connectors (actuator inputs and sensor outputs) to a causal controller model in a causal simulation environment. From a mathematical point of view, this corresponds to reformulating (5) in state space form (1), by means of analytical and/or numerical transformations.

3.3 Development of simplified models

The object-oriented approach, and in particular replaceable components, allows to define and manage families of models of the same plant with different levels of complexity, by providing more or less detailed implementations of the same abstract interfaces. For example, consider a heat exchanger model: the abstract interface has four fluid connectors, two for the hot fluid inlet and outlet, and two for the cold fluid inlet and outlet. The corresponding implementation might range from a very simple static model based on log-mean temperatures, with a few algebraic equations, up to a very detailed finite volume model using nonlinear fluid properties and empirical correlations for heat transfer, and with dozens of state variables and a few hundred algebraic equations.

This feature of OOM allows to develop simulation models with different degrees of detail (and CPU load) throughout the entire life of an engineering project, from preliminary design down to commissioning and personnel training, within a coherent framework. However, this activity is based on manual work by the modeller, who needs to develop the different implementations explicitly. Moreover, it is often not easy to obtain compact models such as (1), because this requires to apply simplifications that may not fit well the abstract component boundaries.

3.4 Generation of real-time simulation code

An important step in the development of embedded control systems is Hardware-In-the-Loop simulation (HIL), where the real control hardware is tested by connecting it to a real-time simulator, instead of the real plant. Many currently available EOOL-based tools support automatic generation of efficient real time code starting from fairly large simulation models in the form (5). A common strategy for this purpose is to apply inline integration [12, 11] to (5), i.e. to

substitute the derivatives with their approximation formulae (e.g. Euler's formula), and then solve the system using all available numerical and symbolic techniques.

In order to provide real-time code which is fast enough, it is usually important to reduce the model complexity with respect to off-line simulation models - this can be done by following the approach sketched in Section 3.3.

3.5 Optimization

Some EOOL and tools support some kind of optimization, which might be useful for control system design. For example, the gPROMS language [6] has allowed to declare mixed-integer nonlinear optimization problems for a long time. More recently, extensions to the Modelica language were proposed to formulate optimization problems [2].

3.6 Future perspectives

It is the authors' view that EOOL-based tools should support advanced control system design problems in a much more direct way, by making extensive use of control-oriented symbolic manipulation techniques. Ideally, it would be good if the control engineer could develop a detailed simulation model by using object oriented tools and re-usable model libraries, then automatically obtain simplified, compact models which are already formulated as required by the specific control technique. The availability of such tools might promote the application of advanced, model-based techniques that are currently limited by the model development process.

Being aware that this is a very long-term goal, which might even require some kind of artificial intelligence, some first steps in this direction are discussed in the following sections, with particular reference to the Modelica language and Modelica compliant tools.

4. Basic enabling technologies

The advanced, control-oriented features of future EOOL tools need some basic enabling technologies and methodologies to build upon. These are briefly discussed in this section.

4.1 Open standards for model and data exchange

Advanced applications of OOM to control system design will most likely require to use different specialized tools in a coordinated fashion, rather than relying on one-fits-all comprehensive software tools. In fact, during the last decades, the number and the quality of simulation, design and analysis tools has increased enormously: there is plenty of open and closed source software for the simulation of physical systems, control synthesis, data analysis, test, validation, personnel training via a graphical user interface, etc. Some of these tools are developed for specific purposes, while other are more general in scope (e.g., symbolic manipulation tools, differential equation solvers, data analysis packages). Unfortunately, all this software development activity did not follow any standardisation process, leading to a great diversity in the representation of the information. The definition of standard interfaces will be useful for the information exchange between different applications; as a consequence, by providing a representation for all the stages of the model manipulation (starting from the translation, going to the flattening, to the model order reduction and so forth) it will be possible to make all the applications interact at different levels, thus combining positive effects from different applications and obtaining better results.

Exchange formats for model equations and for simulation data should probably be based on the XML language, for several reasons:

- the tree structure of XML documents easily allows to represent complex data structures, including symbolic representations of equations;
- XML documents can be read with standard text-editors and browsers, thus avoiding all the problem usually raised by obscure, ad-hoc binary formats;
- there exists a large base of software (open source and commercial) for the handling of XML files;
- by re-using this existing software, it is quite straightforward to translate an XML document representing a mathematical model into any other equation-based language, and vice-versa;
- binary XML formats can be used to reduce the verbosity of XML documents and the cost of parsing them;
- there exist some languages (e.g. DTD and XSD) to formally specify the structure of the information the XML file must contain.

Such standard interfaces for flattened Modelica models and their corresponding simulation data are currently being investigated at Politecnico di Milano using the OpenModelica compiler [16, 1] as a host EOOL environment, and symbolic manipulation tools such as Mathematica, Maple or Maxima as target environments. If the model is purely continuous-time, i. e., it is equivalent to the DAE (5), then MathML [42] on one side, and ModelicaXML [35] on the other side might constitute good starting points. If hybrid models are considered, one may consider all the languages developed for the description of hybrid automata in recent years [8], even though the class of hybrid systems which can be described in Modelica with *when* statements is larger than just hybrid automata.

4.2 Model Order Reduction

Another key enabling technology is represented by mixed numerical-symbolic Model Order Reduction (MOR) techniques. These have already been successfully applied to the analysis of electrical circuit models, which are based on DAE models such as (5), see [40, 17], and are currently available in commercial tools such as Analog Insydes [13]. The MOR strategies are based on the clever application of three fundamental steps:

- specify an allowed error bound, e.g. in terms of percentage error of certain steady-state output values corresponding to given constant inputs, or in terms of maximum deviation of some outputs from a reference trajectory obtained with given input signals, or in terms

of maximum error of small-signal frequency responses around a certain operating point and within a given frequency interval;

- derive a ranking of all terms in all equations, expressing how much each term has a significant effect on the required modelling accuracy;
- remove all terms in ascending order, until the specified error bound is reached.

Other MOR techniques exist to reduce large linear systems, based on concepts such as modal analysis and projection methods; see [38] for a comprehensive overview.

The application of such MOR tools and techniques, possibly with extended functionality and algorithms, looks very promising not only for the simplification of electrical circuit models, but also for the order reduction of generic, nonlinear DAE models, obtained from the flattening of generic EOOL models. This kind of techniques should allow to automatically obtain approximated compact models such as (1), starting from much more detailed simulation models, by formulating specific approximation bounds in control-relevant terms (e.g., percentage errors of steady-state output values, norm-bounded additive or multiplicative errors of weighted transfer functions, or ℓ_∞-norm errors of output transients in response to specified input signals). Given the ever-increasing computation power that can be expected by Moore's law, the future of these techniques for CACSD applications definitely appears bright.

4.3 Reliable steady-state initialization and static model inversion

A reliable support to the control engineer's activity requires to improve the techniques to solve the steady-state equations (6), which are usually the starting point for any kind of analysis, including MOR. As pointed out earlier, solving (6) requires iterative methods which might fail if not properly initialized. Troubleshooting can be very frustrating and time-consuming, and calls for experts of both simulation methods and domain-specific models. This is not acceptable in the envisioned framework, which is based on *automatic* manipulation by EOOL tools.

One option, which is currently being investigated at Politecnico, is to introduce extensions to the Modelica language to support homotopy methods, in a way similar to the approach followed by the SPICE circuit simulation program. The basic idea is that each model has two versions: the "easy" one, for which it is easier to find a steady-state solutions, and the "true" version, which is the model to be actually used for simulation. The two models share the same variables, but use different equations. The system model obtained by the aggregation of the "easy" models is represented by

$$F_e(x, \dot{x}, y, u, p, t) = 0, \tag{7}$$

while the aggregation of the "true" models leads to

$$F_t(x, \dot{x}, y, u, p, t) = 0, \tag{8}$$

The idea is now to first solve the initialization problem for (7), which should not give rise to significant numerical problems. The solution to this simplified problem constitutes the first guess for a new problem:

$$(1-\alpha)F_e(\bar{x}, 0, \bar{u}, \bar{y}, p, 0) + \alpha F_t(\bar{x}, 0, \bar{u}, \bar{y}, p, 0) = 0, \tag{9}$$

which will be solved by varying α from 0 to 1 in small steps, eventually finding the steady-state solution of system (8). In general, this approach should help to reduce (and hopefully eliminate) the need to manually set initial guess values for iteration variables of initialization problems.

5. New functionalities for control system design

5.1 Simplified symbolic transfer functions

In many interesting cases, the performance of the control system is limited by the dynamic behaviour of the controlled plant. For example, poorly damped oscillations can limit the bandwidth of motion control systems, as well as non-minimum phase behaviour. The control engineer can gain a lot of useful insight from approximated transfer functions, where the dependence of the critical dynamic features from a few physical parameters is clearly visible. For instance, the natural frequency of a pair of complex poles in a mechanical system might depend mainly on the stiffness and on the mass of a certain physical component, or, the time constant of a right-half-plan zero in a fluid system might depend on the fluid velocity in a certain point.

This is a first case where automatic MOR techniques could prove extremely useful. Ideally, the user should specify the steady-state operating point, the relevant inputs and outputs, and some frequency-weighted error bounds, and get low-order approximated transfer functions of the linearized system, with approximated but explicit dependence of the transfer function features (gains, poles and zeros) from the physical model parameters. A suitable combination of EOOL tools (equipped with model import/export interfaces) with existing MOR tools like Analog Insydes [13] could provide very interesting results in this direction without too much effort.

5.2 Automatic derivation of LFT models

Once a model has been reduced to a low-order state-space form by the combined application of symbolic MOR techniques and clever model simplifications as explained in Section 3.3, it might be useful to automatically bring them in the form required for advanced control system design, using symbolic manipulation tools. Modern control theory provides methods and tools in order to deal with design problems in which stability and performance have to be guaranteed also in the presence of model uncertainty, both for regulation around a specified operating point and for gain scheduled control system design.

Most of the existing control design literature assumes that the plant model is given in the form of a Linear Fractional Transformation (LFT) (see, e.g., [46, 27]), a modelling paradigm which is currently an active research topic in the control engineering and system identification communities. In the robust control framework LFT models consist of a feedback interconnection between a *nominal* LTI

plant and a (usually norm-bounded) operator which represents model uncertainties, e. g., poorly known or time-varying parameters, nonlinearities, etc. A generic such LFT interconnection is depicted in Figure 1, where the nominal plant is denoted with P and the uncertainty block is denoted with Δ. Note that this representation is extremely general, and by no means limited to uncertain LTI systems; in fact, it is possible to describe any nonlinear DAE system by putting all the nonlinear functions in the Δ block and by providing an LTI model with direct feedthrough terms to describe the algebraic equations.

LFT models can be used for the design of robust and gain scheduling controllers, but they can also serve as a basis for structured model identification techniques, where the uncertain parameters that appear in the feedback blocks are estimated based on input/output data sequences.

The process of extracting uncertain parameters from the design model of the system to be controlled is a highly complex one. Symbolic techniques play a very important role in this process: the main use for such techniques is to find, via suitable pre-processing steps, equivalent representations of rationally dependent parametric matrices, which automatically lead to lower-order LFT representations. Tools already exist to perform this task [27].

The LFT modelling problem in its simplest form is associated with the problem of designing a controller for operation near a nominal operating point for the system. The problem is then formulated on a local linearised representation of the plant to be controlled and is familiarly termed "pulling out the Δs", i.e., it consists of manually or symbolically manipulating the linearised equations in order to separate the nominal part of the plant from the uncertain one, arranging them in a suitable feedback interconnection. This reformulation of the plant model lies at the vary basis of modern robust control theory and is currently supported by a number of different symbolic manipulation tools. A recent overview of the state-of-the-art in this research area can be found in [18]. As an example, consider a time-invariant, nonlinear state-space system in the form

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t), p) \\ y(t) &= g(x(t), u(t), p), \end{aligned} \tag{10}$$

where p denotes a vector of uncertain parameters, and assume that the equilibrium condition $\bar{x}$, $\bar{u}$, $\bar{y}$, which solves the steady-state equations

$$\begin{aligned} 0 &= f(\bar{x}, \bar{u}, p) \\ \bar{y} &= g(\bar{x}, \bar{u}, p) \end{aligned} \tag{11}$$

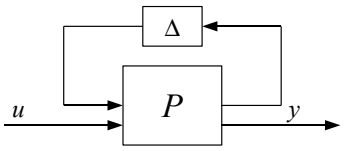


Figure 1. Block diagram of the typical LFT interconnection adopted in the robust control framework.

is available. Defining now the *deviation variables*

$$\delta x(t) = x(t) - \bar{x} \tag{12}$$

$$\delta u(t) = u(t) - \bar{u} \tag{13}$$

$$\delta y(t) = y(t) - \bar{y}, \tag{14}$$

it is possible to approximate the dynamics of (10) with a the following linear, parameter-dependent system

$$\begin{aligned} \dot{\delta x}(t) &= A(p)\delta x + B(p)\delta u \\ \delta y(t) &= C(p)\delta x + D(p)\delta u \end{aligned}, \tag{15}$$

where the four matrices A, B, C, D are the Jacobians of the two functions f and g:

$$A(p) = \frac{\partial f}{\partial x}, \quad B(p) = \frac{\partial f}{\partial u}$$
$$C(p) = \frac{\partial g}{\partial x}, \quad D(p) = \frac{\partial g}{\partial u}.$$

Under suitable assumptions (such as that the state space matrices are polynomial or rational functions of the elements of p, see, e.g., [46]) it is possible to transform the system description (15) into an LFT representation (see, again, Figure 1). As mentioned previously, converting (15) into an LFT with a Δ block of minimum dimension is a non-trivial symbolic manipulation problem.

An even more challenging formulation of the LFT modelling problem is the one of *simultaneously* representing in LFT form all the linearisations of interest for control purposes of the given nonlinear plant. Indeed, in many control engineering applications a single control system must be designed to guarantee the satisfactory closed-loop operation of a given plant in many different operating conditions in the presence of parametric and possibly non parametric uncertainty. The gain scheduling approach to the problem, which has been part of the engineering practice for decades, can be roughly summarised as follows: find one or more *scheduling variables* α which can completely parametrise the operating space of interest (e.g., the flight envelope in the case of aircraft control) for the system to be controlled; define a parametric *family* of linearised models for the plant associated with the set of operating points of interest; finally, design a *parametric* controller which can both ensure the desired control objectives in each operating point and an acceptable behaviour during (slow) transients between one operating condition and the other. Many design techniques are now available for this problem (see, e.g., [5, 22, 37]), which can be reliably solved, provided that a suitable model in parameter-dependent form has been derived for the system to be controlled. The goal here would be to arrive at a representation of the dynamics of the nonlinear system in the form depicted in Figure 2, which is usually denoted as an LPV-LFT system, where the LPV acronym stands for Linear Parameter-Varying. The model structure now includes two feedback interconnections: the block $\Delta(p)$ takes into account the presence of the uncertain parameter vector p, while the block $\Theta(\alpha)$ models the effect of the varying operating point, parametrised by the vector of time-varying parameters α.

The state-of-the-art of modelling for gain scheduling can be briefly summarised by defining two classes of modelling approaches: *analytical* methods based on the availability of (relatively) reliable nonlinear equations for the dynamics of the plant, from which suitable control-oriented representations can be derived (see, e.g., [28] and the references therein); *experimental* methods based entirely on identification, i.e., aiming at deriving LPV models for the plant directly from input/output data (see, among many others, [21, 45, 23]). The methods belonging to the first class aim at developing LPV models for the plant to be controlled by resorting to, broadly speaking, suitable extensions of the familiar notion of linearisation, developed in order to take into account off-equilibrium operation of the system. As far as experimental methods are concerned, most LPV identification techniques are based on the assumption that the identification procedure can rely on one *global* identification experiment in which both the control input and the scheduling variables are (persistently) excited in a simultaneous way. This assumption may not be a reasonable one in many applications, in which it would be desirable to try and derive a parameter-dependent model on the basis of *local* experiments only, i.e., experiments in which the scheduling variable is held constant and only the control input is excited. Such a viewpoint has been considered in [43, 34, 23], where numerical procedures for the construction of parametric models for gain scheduling on the basis of local experiments and for the interpolation of local controllers have been proposed.

To our best knowledge the only documented attempt at deriving control-oriented LFT models automatically from a nonlinear simulator is presented in [44], where the focus was on the automatic generation of LFT models for aerospace applications. Much remains to be done. An EOOL-based CACSD tool dealing with the generation of control-oriented LFT models should allow to specify some error bounds for the system approximation (with respect to steady-state, transient, and frequency response), the choice of input, output and scheduling variables, and the choice of parameters to include in the LFT representation. Based on that, it should be able to automatically compute the structure of the interconnections defined in Figures 1 and 2 for the robust and gain-scheduling control design problems, respectively, the state-space matrices of the nominal part P of the model (either as analytical expressions, if possible, or at least as algorithms for their computation) and analytical or algorithmic representations of the feedback blocks $\Theta(\alpha)$ and $\Delta(p)$. Finally, it is apparent from the short literature review presented above that currently only physical and black-box modelling methods are available, while no general purpose CACSD tools capable of combining first principles models and experimental data in a single control-oriented model seem to exist. The convergence of the two modelling approaches both in terms of methods and tools would be a very desirable outcome of the research in this field.

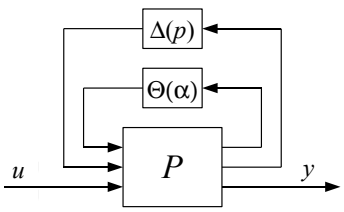


Figure 2. Block diagram of the typical LFT interconnection adopted in the robust LPV control framework.

5.3 Automatic computation of inverse models for robotic systems

The design of controllers for non-redundant robotic manipulators with N degrees of freedom usually starts from the equations of motion obtained from the Euler-Lagrange equations [39]:

$$B(q)\ddot{q} + H(q,\dot{q})\dot{q} + g(q) = \tau \tag{16}$$

$$y_p = K(q) \tag{17}$$

$$y_v = \frac{\partial K}{\partial q}\dot{q}, \tag{18}$$

where q is the N-element vector of Lagrangian coordinates, which usually correspond to the rotation angles of the actuator motors, $\dot{q}$ is the vector of the corresponding generalized velocities, y_p describes the position and orientation vector of the end effector, y_v contains the corresponding generalized velocities, τ is the vector of generalized applied forces corresponding to each degree of freedom (usually the torques applied by rotating actuators), $B(q)$ is the inertia matrix, $H(q,\dot{q})$ is the matrix corresponding to the centripetal, Coriolis, and viscous friction forces, while the vector $g(q)$ accounts for the effects of the gravitational field; all vectors have dimension N.

The classical approach to write (16) requires to compute the so-called direct kinematics (DK), i.e. how the values of q and $\dot{q}$ translate into the position and motion of the robot's end effector, then to compute the Lagrange function, i.e. the difference between kinetic and potential energy, and apply the Euler-Lagrange equations. This can be done manually, or using one of the specialized tools available for this task. Equations (16)-(18) can then be used as a basis for both controller design and system simulation.

Within an OOM approach, it is possible to save much time by developing an object-oriented model using an EOOL, e.g. using the Modelica MultiBody library [33]. Due to the kinematic constraints imposed by the joints, the original flattened model corresponds to an index-3 DAE,

$$F(x,\dot{x},y,u) = 0, \tag{19}$$

which is mathematically equivalent to the Lagrange model (16)-(18).

Currently available Modelica tools tackle the problem by applying specialized algorithms, which exploit the knowledge of the topology of the kinematic chain, as well as standard techniques such as BLT partitioning, tearing, dummy derivatives and symbolic solution of equations

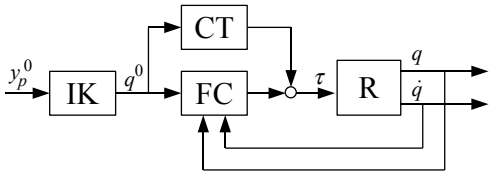


Figure 3. Block diagram of computed torque control

[33]. From a conceptual point of view, a change of state variables x allows to transform (19) into an index-1 system

$$F_1(x, \dot{x}, y, u) = 0, \tag{20}$$

where

$$x = \begin{bmatrix} x_p \\ x_v \end{bmatrix} = \begin{bmatrix} q \\ \dot{q} \end{bmatrix}, \quad y = \begin{bmatrix} y_p \\ y_v \end{bmatrix}, \quad u = \tau. \tag{21}$$

Eventually, efficient procedures are produced to solve (20) for $\dot{x}$ and y given x and u, thus actually bringing the system into state-space form:

$$\begin{aligned} \dot{x} &= f(x, u) \\ y &= g(x, u). \end{aligned} \tag{22}$$

This formulation can be used to solve simulation problems, by linking it to any ODE/DAE solver. However, there are several other interesting things that could be done with (20), from a control engineer's perspective.

Robot trajectories are originally defined in terms of end effector coordinates as functions of time $y_p^0(t)$. In order to obtain the corresponding reference trajectories in Lagrangian coordinates for the low-level robot joint controllers, (17)-(18) must be solved for q, $\dot{q}$, thus computing the so-called inverse kinematics (IK):

$$q^0 = K^{-1}(y_p^0) \tag{23}$$

$$\dot{q}^0 = \left(\frac{\partial K}{\partial q}\right)^{-1} y_v^0; \tag{24}$$

note that the Jacobian of $K(q)$ is also needed to solve (23), since analytical inverses cannot usually be obtained. Furthermore, two interesting approaches to model-based robot control are based on suitable manipulations of eq. (16): the *pre-computed torque* approach and the *inverse dynamics* approach [39].

The pre-computed torque approach is a feed-forward compensation scheme, where the theoretical torque required to follow the reference trajectory is directly fed to the torque actuators (see Fig. 3) in order to obtain a good dynamic response to the set point y_p^0. The CT block performs this task by solving (16) for τ, given the reference trajectory and its derivatives:

$$\tau = B(q^0)\ddot{q}^0 + H(q^0, \dot{q}^0)\dot{q}^0 + g(q^0). \tag{25}$$

A feedback controller (FC) is also included to deal with uncertainties and disturbances.

The inverse dynamics approach is a feedback compensation scheme, that uses the model in order to transform

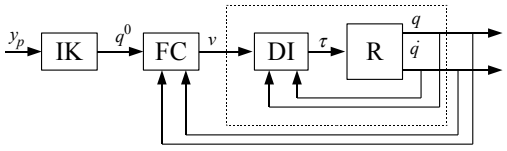


Figure 4. Block diagram of inverse dynamics control

the non-linear control problem into a linear, time-invariant one. Define a virtual input variable v, which satisfies the following equation

$$\tau = B(q)v + H(q, \dot{q})\dot{q} + g(q). \tag{26}$$

Since the inertia matrix B is structurally non-singular, it is always possible to solve (26) for v:

$$v = B^{-1}(q)\left(\tau - H(q, \dot{q})\dot{q} - g(q)\right). \tag{27}$$

Plugging v in the robot dynamics equation (16), one obtains:

$$\ddot{q} = v. \tag{28}$$

The block diagram interpretation of these equations is shown in Fig 4: thanks to the dynamic inversion (DI) block, the dynamic relationship between the virtual input v and the Lagrangian positions and velocities q and $\dot{q}$ (represented by the dotted block) is now described by a simple integrator and a double integrator, respectively. It is then easy to tune a fixed-parameter, linear feedback controller (FC) in order to obtain the desired closed-loop dynamics.

Starting from the index-1 DAE robot model (20), it is straightforward to derive the equations and then the explicit algorithms to compute the DK, IK, CT, and DI, by using the same techniques employed to bring (20) into state-space form. The DK (17)-(18) is obtained by solving (20) for y_p (and possibly y_v) given q (and possibly $\dot{q}$), while the IK is obtained by solving (20) for q (and possibly $\dot{q}$) given y_p (and possibly y_v); the subset of required equations is found by suitable analysis of the incidence matrix. The CT (25) is obtained by solving (20) for τ given q, $\dot{q}$, and $\ddot{q}$. Finally, the DI (26) is obtained by solving (20) augmented with (26) for τ given v, q, and $\dot{q}$. EOOL tools should then be able to automatically generate the code corresponding to the DK, IK, CT, and DI blocks in two forms: as algorithms to compute the outputs given the inputs (e.g., C code for direct inclusion in the robot controller), as well as equation-based Modelica blocks, which could be used for closed-loop simulation within a Modelica environment.

As a final remark, note that the method of inverse dynamics is a special case of the much more general theory of feedback linearization [20], whose goal is to obtain a LTI dynamics made by pure integrators from generic nonlinear systems, by applying suitable feedback actions as shown in Figure 4. It could also be interesting to investigate the coupling between EOOL tools and symbolic manipulation tools for the design of such controllers.

5.4 Fast and compact models for Model Predictive Control

The Model Predictive Control (MPC) approach [25, 36] is based on a few key ideas, that turn the control problem into an optimization problem. The control variable is a discrete-time variable, that changes periodically every T_s seconds:

$$u(t) = u(k), \quad kT_s \leq t < (k+1)T_s. \tag{29}$$

At each time step k, an optimization problem is solved, whose unknowns are the next values of the control variable $u(k+i)$ over a finite horizon $1 \leq i \leq N$. The first sample $u(k+1)$ is then applied to the actuators at the next time step, the rest of the values are discarded, and the process is repeated over and over, thus implementing a *receding horizon* strategy.

There are different ways to formulate the MPC problem, depending on the specific technique used to solve the problem. Generally speaking, the figure of merit to be minimized is a quadratic function, which suitably weights the future deviations of the controlled variables from the set point and the intensity of the control action, as well as any other problem-specific performance index that has to be minimized, e.g. the financial cost of running the process. The constraints of the optimization problem are the dynamic relationship between the input and output variables, typically in the form (4), and possibly other constraints, such as upper and lower bounds of the state, control and output variables and of their rate of change.

The main advantage of MPC is its intrinsic ability to deal with highly interacting multivariable systems (many control inputs and controlled outputs), while keeping into account operating constraints such as actuator saturations or hard bounds on controlled variables, and at the same time meeting some problem-specific optimality criterion. The main drawback is the high computational load, since a (possibly non-linear and non-convex) constrained optimization problem must be solved at each sampling time; this makes MPC suitable for systems with slow dynamics, e. g. chemical plants, where there is plenty of time to carry out the required computations in real time. This limitation is likely to become less and less stringent in the future, thanks to Moore's law.

The second issue is the requirement that a suitable plant model is available, as the control system performance critically depends on the model quality. Models for linear MPC can be obtained either by linearization of analytical models, or by system identification from experimental and/or simulation data, e. g. step responses; both cases are already supported by current EOOL tools. Nonlinear MPC (NMPC) algorithms are preferably based upon analytical models in state-space form (1), which are derived from physical first-principles models. The conversion to discrete-time form (4) is often performed internally by the NMPC algorithm itself, by standard ODE integration routines. This means that the interface between the EOOL tool and the NMPC tool is similar to the one used for simulation problems, i.e. the state-space form (1), possibly augmented by the Jacobians of the right-hand-sides of (1).

The main requirement for NMPC-oriented models is that they must have the least possible number of state and algebraic variables, in order to keep the complexity of the optimization problem within acceptable limits, and that they have good smoothness properties, in order to avoid convergence problems of the iterative optimization algorithms. The development of those models can be very time consuming, and require highly skilled manpower; it is apparent how better tool support could be extremely useful in order to reduce the development effort and cost.

The potential of OOM for MPC was first noted by Maciejowski at the end of the '90 [24]. There are several reported case studies [14, 15, 3, 19], where the model used in the NMPC algorithm was derived from a Modelica model of the physical plant, using the tool Dymola to produce the code corresponding to the state-space form (1), i.e., the *dsmodel.c* code that is usually linked to ODE/DAE solvers. In order to derive suitably simplified models, the features of Modelica discussed in Section 3.3 have been extensively exploited. In general, this approach has proven much more satisfactory than writing the C-code of the model from scratch; however, it still requires a substantial investment of time and effort for each new application.

The application of the automatic MOR techniques described in section 4, possibly still combined with some manual intervention in terms of replaceable models, looks very promising in order to bring detailed simulation models into a form which is suitable for NMPC with a much more limited effort by the developer.

Furthermore, [19] correctly points out that, although the interface to NMPC algorithms is very similar to the interface to ODE/DAE solvers, the former requires some more flexibility. For example, advanced NMPC schemes can provide on-line estimation of uncertain parameters through the use of extended or unscented Kalman filters. This means that some model parameters are no longer constant throughout a transient, so that the C-code obtained for simulation purposes must be manually adapted. A better option would be to implement a code export interface which makes it possible to turn selected parameters appearing in (5) (which are going to be estimated on-line) into inputs, before transforming the system in state-space form (1).

6. Conclusions

After a brief review of the different uses of models in control system design, the current state of the art of EOOL-based tools for CACSD has been reviewed: apparently, currently available tools mainly focus on simulation tasks. Several further directions for research and development in EOOL tools where then discussed, which go beyond the mere simulation problem. Results in these directions could substantially improve the level of support to the control engineer willing to apply advanced, model-based control techniques to real-life problems, starting from object-oriented models of the plant.

References

[1] OpenModelica home page. URL: `http://www.ida.liu.se/labs/pelab/modelica/OpenModelica.html`.

[2] J. Åkesson. Optimica - An extension of Modelica supporting dynamic optimization. In *Proceedings 6th International Modelica Conference*, pages 57–66, Bielefeld, Germany, Mar. 3–4 2008.

[3] J. Åkesson and O. Slätteke. Modeling, calibration and control of a paper machine dryer section. In *Proceedings 5th International Modelica Conference*, pages 411–420, Vienna, Austria, Sep. 4–5 2006.

[4] M. Andersson, S. E. Mattsson, D. Brück, and T. Schöntal. OmSim - an integrated environment for object-oriented modelling and simulation. In *Proceedings of the IEEE/IFAC Joint Symposium on Computer-Aided Control System Design, CACSD'94*, pages 285–290, Tucson, Arizona, March 1994.

[5] P. Apkarian and R. J. Adams. Advanced Gain-Scheduling Techniques for Uncertain Systems. *IEEE Transactions on Control System Technology*, 6:21–32, 1998.

[6] P. I. Barton and C. C. Pantelides. The modeling of combined continuous and discrete processes. *AIChE Journal*, 40:966–979, 1994.

[7] D.A. van Beek, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Foundations of an interchange format for hybrid systems. In A. Bemporad, A. Bicchi, and G. Butazzo, editors, *Computation and Control, 10th International Workshop*, volume 4416 of *Lecture Notes in Computer Science*, pages 587–600. Springer Verlag, 2007.

[8] D.A. van Beek, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Concrete syntax and semantics of the compositional interchange format for hybrid systems. In *Proc. 17th IFAC World Congress*, Seoul, Korea, Jul 6–11 2008.

[9] E. Carpanzano and C. Maffezzoni. Symbolic manipulation techniques for model simplification in object-oriented modelling of large scale continuous systems. *Mathematics and Computers in Simulation*, 48(2):133–150, 1998.

[10] F. E. Cellier and H. Elmqvist. Automated formula manipulation supports object-oriented continuous-system modeling. *IEEE Control Systems Magazine*, 13(2):28–38, 1993.

[11] H. Elmqvist, S. E. Mattsson, and H. Olsson. New methods for hardware-in-the-loop simulation of stiff models. In *Proceedings 2nd International Modelica Conference*, pages 59–64, Oberpfaffenhofen, Germany, Mar. 18–19 2002.

[12] H. Elmqvist, M. Otter, and F. Cellier. Inline integration: A new mixed symbolic /numeric approach for solving differential–algebraic equation systems. In *Proc. ESM'95, European Simulation Multiconference*, pages xxiii–xxxiv, Prague, Czech Republic, Jun. 5–8 1995.

[13] The Fraunhofer-Institut für Techno-und Wirtschaftsmathematik. Analog Insydes. URL: `http://www.analog-insydes.de/`.

[14] R. Franke. Formulation of dynamic optimization problems using modelica and their efficient solution. In *Proceedings 2nd International Modelica Conference*, pages 315–323, Oberpfaffenhofen, Germany, Mar. 18-19 2002.

[15] R. Franke, M. Rode, and K. Krüger. On-line optimization of drum boiler startup. In *Proceedings 3rd International Modelica Conference*, pages 287–296, Nov. 3–4 2003.

[16] P. Fritzson, P. Aronsson, A. Pop, H. Lundvall, K. Nyström, L. Saldamli, D. Broman, and A. Sandholm. OpenModelica - A free open-source environment for system modeling, simulation, and teaching. In *Proceedings IEEE International Symposium on Computer-Aided Control Systems Design*, Munich, Germany, Oct. 4–6 2006.

[17] T. Halfmann and T. Wichmann. Symbolic methods in industrial analog circuit design. In *Scientific Computing in Electrical Engineering*, Mathematics in Industry. Springer Verlag, 2006.

[18] S. Hecker and A. Varga. Symbolic manipulation techniques for low order LFT-based parametric uncertainty modelling. *International Journal of Control*, 79(11):1485–1494, 2006.

[19] L. Imsland, P. Kittilsen, and T. S. Schei. Model-based optimizing control and estimation using modelica models. In *Proceedings 6th International Modelica Conference*, pages 301–310, Bielefeld, Germany, Mar. 3–4 2008.

[20] H. K. Khalil. *Nonlinear Systems*. Prentice Hall, 3rd edition, 2002.

[21] L. Lee and K. Poolla. Identification of linear parameter-varying systems using nonlinear programming. *Journal of Dynamic Systems, Measurement and Control - Transactions of the ASME*, 121(1):71–78, 1999.

[22] D. J. Leith and W. E. Leithead. Survey of gain-scheduling analysis and design. *International Journal of Control*, 73(11):1001–1025, 2000.

[23] M. Lovera and G. Mercere. Identification for gain-scheduling: a balanced subspace approach. In *2007 American Control Conference*, New York, USA, 2007.

[24] J. M. Maciejowski. Modelling and predictive control: enabling technologies for reconfiguration. *Annual Reviews in Control*, 23(1):13–23, 1999.

[25] J. M. Maciejowski. *Predictive control: with constraints*. Prentice Hall, 2002.

[26] C. Maffezzoni and R. Girelli. Modular modelling in an object-oriented database. *Mathematical Modelling of Systems*, 4:121–147, 1998.

[27] J.-F. Magni. Linear fractional representation toolbox. Technical Report TR 6/08162 DCSD, ONERA, 2004.

[28] A. Marcos and G. Balas. Development of linear-parameter-varying models for aircraft. *Journal of Guidance, Control and Dynamics*, 27(2):218–228, 2004.

[29] S. E. Mattsson, M. Andersson, and K. J. Åström. Modeling and simulation of behavioral systems. In *Proceedings of the 32nd IEEE Conference on Decision and Control*, volume 4, pages 3636–3641, San Antonio, Texas, Dec. 1993.

[30] S. E. Mattsson, M. Andersson, and K. J. Åström. Object-oriented modelling and simulation. In D. Linkens, editor, *CAD for Control Systems*, pages 31–69. Marcel Dekker Inc., New York, 1993.

[31] S. E. Mattsson and H. Elmqvist. Simulator for dynamical systems using graphics and equations for modeling. *IEEE Control Systems Magazine*, 9(1):53–58, Jan. 1989.

[32] S. E. Mattsson, H. Elmqvist, and M. Otter. Physical system

modeling with Modelica. *Control Engineering Practice*, 6(4):501–510, 1998.

[33] M. Otter, H. Elmqvist, and S. E. Mattsson. The new Modelica MultiBody library. In *Proceedings 3rd International Modelica Conference*, pages 311–330, Linköping, Sweden, Nov. 3–4 2003. URL: `http://www.modelica.org/events/Conference2003/papers/h37_Otter_multibody.pdf`.

[34] B. Paijmans, W. Symens, H. Van Brussel, and J. Swevers. A gain-scheduling-control technique for mechatronic systems with position-dependent dynamics. In *Proceedings of the 2006 American Control Conference*, Minneapolis, USA, 2006.

[35] A. Pop and P. Fritzson. ModelicaXML: A Modelica XML representation with applications. In *Proceedings of the 3rd International Modelica Conference*, pages 419–430, Linköping, Nov 3–4 2003.

[36] S. J. Qin and T. A. Badgwell. A survey of industrial model predictive control technology. *Control Engineering Practice*, 11:733–764, 2003.

[37] W. Rugh and J. Shamma. Research on gain scheduling. *Automatica*, 36(10):1401–1425, 2000.

[38] P. Schwarz, J. Bastians, C. Clauss, J. Haase, A. Köhler, G. Otte, and P. Schneider. A tool-box approach to computer-aided generation of reduced-order models. In *Proceedings EUROSIM 2007*, 2007.

[39] L. Sciavicco and B. Siciliano. *Modelling and Control of Robot Manipulators*. Springer Verlag, 2000.

[40] R. Sommer, T. Halfmann, and J. Broz. Automated behavioral modeling and analytical model-order reduction by application of symbolic circuit analysis for multi-physical systems. In *Proceedings of EUROSIM 2007*, Ljubljana, Slovenia, Sep 9–13 2007.

[41] The Modelica Association. Modelica - A unified object-oriented language for physical systems modeling - Language specification version 3.0. Online, Sep. 5 2007. URL: `http://www.modelica.org/news_items/documents/ModelicaSpec30.pdf`.

[42] The World Wide Web Consortium. Mathematical Markup Language (MathML) Version 2.0. Online, Oct 21 2003. URL: `http://www.w3.org/TR/MathML2/`.

[43] J.J.M. van Helvoort, M. Steinbuch, P.F. Lambrechts, and R. van de Molengraft. Analytical and experimental modelling for gain-scheduling of a double scara robot. In *Proceedings of the 3rd IFAC Symposium on Mechatronic Systems*, Sydney, Australia, 2004.

[44] A. Varga, G. Looye, D. Moormann, and G. Gräbel. Automated generation of LFT-based parametric uncertainty descriptions from generic aircraft models. *Mathematical and Computer Modelling of Dynamical Systems*, 4(4):249–274, 1988.

[45] V. Verdult. *Nonlinear system identification: a state space approach*. PhD thesis, University of Twente, 2002.

[46] K. Zhou, J. U. Doyle, and K. Glover. *Robust and optimal control*. Prentice-Hall, New Jersey, 1996.

A Static Aspect Language for Modelica Models

Malte Lochau Henning Günther

Institute for Programming and Reactive Systems, TU Braunschweig, Germany,
{m.lochau,h.guenther}@tu-bs.de

Abstract

With the introduction of the new Modelica major version 3, innovations mainly consist of further model restrictions for increased model quality. In addition, developers often want to ensure the compliance to further requirements early in the development cycle. Mostly emerging as domain specific conventions that often crosscut model structures, according checking mechanisms are required that are detached from the core language. In this paper, a declarative language is presented for specifying and evaluating quantified rules for static model properties. Based on aspect-oriented programming, the language allows for concise and expressive model inspections and a variable and typing concept facilitate subsequent model manipulations. A nascent implementation framework is proposed, based on the logic meta programming paradigm, thus leading to efficient and scalable aspect processing applicable as model query engine for an AOP Modelica Compiler.

Keywords Early Checking, Aspect Orientation, Modelica Model Inspection

1. Introduction

The Modelica description standard [2] proposes a modern multi-discipline language for component-based mathematical modeling and simulation of complex physical systems (see e.g. [12, 21]). Its equation-based, object-oriented, and declarative nature allows for hierarchical specification of system structure and behavior and smooth integration, evolution and reuse of developed components. The innovations of the version 3 of the Modelica Specification [2] mainly constitute further restrictions, e.g. the locally balanced model property [17], hence aiming at design rules for increased model quality.

Moreover, modeling follows established practices according to a substantial set of rules and properties. Generalizing, a developer wants to be able to specify, maintain, and analyze arbitrary structural model properties accompanying all development phases. A multitude of requirements, especially non-behavioral quality properties and properties not evident in testing cannot be adequately expressed solely using language constructs of Modelica as they exceed the expressiveness of object-oriented principles. As an example, the locations within a model to be considered for checking design policies such as

"For clarity reasons, inheritance hierarchies deeper than 4 are to be avoided"

are scattered throughout the code, i.e. the formulation of such a demand crosses model composition hierarchies. Also the balanced model concept of Modelica 3 [17] consists of a set of entangled demands, e.g.

"The number of flow variables in a connector must be identical to the number of non-causal non-flow variables"

which correlates properties of connectors.

Besides, design criteria considering chains of several models/connectors might be of interest, as well as further coding conventions such as correct library usage, domain specific patterns, and simple naming conventions, e.g.

"Flow variables shall be named with a flow postfix"

Therefore, an overall mechanism is desirable for expressing, modularizing, and finally ensuring compliance with especially application-domain specific design rules and patterns. Allowing for arbitrary model inspection with respect to those requirements, certain kinds of errors can be avoided from the start. For this purpose, aspect-oriented programming (AOP) offers a promising approach: Superposing the object-oriented paradigm and being oblivious with respect to the underlying core language, aspects allow for declarative quantification and modularization of concerns that crosscut the component structure and functionality of models.

As the multitude of properties of Modelica models are already fixed at definition/compile time – especially almost all structural characteristics – *static aspects* at source code level seem to provide a sufficient approach for handling a wide range of aspect-oriented concerns for an early and efficient checking process inside the development loop. Therefore, a domain-specific static language is required for specifying aspect rules that refer to static model entities constituted by fundamental Modelica language units. According to the aspect-oriented paradigm, this language must allow for intuitive and quantified formulation and integration (*weaving*) of aspects without explicitly affecting the underlying "host" language and existing models under consideration. Instead, a smooth description, isolation,

2nd International Workshop on Equation-Based Object-Oriented Languages and Tools. July 8, 2008, Paphos, Cyprus.

composition, and reuse of aspects is made possible for a collection of (connected) Modelica models, their inner components and behavioral descriptions, therefore specifying the properties emerging from the demanded requirements.

In this paper, syntax and semantics of a rule language for static aspects is presented in terms of expressions (point cuts) matching specific locations (join points) in Modelica models. Reflecting basic entities of such models, the language enables model queries at source code level, e.g. inspection and correlation of classes, connectors, and equations. The syntactical structure is based on predefined Modelica language primitives serving as "atoms" and operators for complex term construction. The design of the language aims at expressive and declarative encoding of a concise and succinct set of static design rules. The semantics includes proposals for the usage of variable bindings and point cut types for accessing and manipulating model elements. Although this paper focuses on a comprise definition of the static aspect language rather than an extensive case study, nevertheless some examples as well as a general discussion of its application is provided. An implementation framework for the aspect language is proposed that is under construction. Herein, logic programming principles are used for efficient rule processing by an integrated evaluation engine.

This paper is organized as follows: In Section 2, a brief overview on aspect-orientation and further related work is given. In Section 3, a complete and formalized specification of syntax and semantics is presented for a domain specific static aspect language for Modelica models, and an extension for a variable concept and type system is mentioned. The implementation framework of the language is proposed in Section 4, and some proposals for sample applications are depicted in Section 5. Section 6 concludes.

2. Aspect-Orientation and Related Work

Aspect-oriented programming (AOP) [10, 15] is motivated by the observation, that nowadays abstraction perceptions for logical system structuring and design mainly refer to the notions of hierarchy and (de-) composition. Accordingly, current programming and modeling language paradigms such as the object-oriented principles of Modelica are designed from this perspective. Nevertheless, in many cases there are concerns that crosscut a composition structure, so-called *aspects*, which are therefore contradicting the means of expression of such languages.

Concepts of aspect-oriented programming aim at capturing such crosscutting concerns by appropriate modularization constructs called *point cuts*. Point cuts are expressions matching well-defined combinations of correlated points within programs/models of the underlying component-based language, so-called *join points*. In the *advice* part of an aspect definition, actions/manipulations can be defined to be applied to the matching join points. Depending on the point in time at which aspects are considered, hence *weaved* to the host program/model, two categories can be distinguished: static and dynamic aspects, thus either at definition/compile time or run time. For instance, an implementation of dynamic aspects for Java is provided by AspectJ [14], whereas the JTL approach [7] allows for the specification of static aspects for Java using *query by example*.

The static aspect language for Modelica proposed in this paper is mainly inspired by the PDL described in [16] which is based on principles of description logics [3, 4]. The adoption of the approach to the Modelica language refers to the syntactical and semantic structure of Modelica version 3 [2]. As a major enhancement, a concept for variable bindings is proposed.

Previous efforts for checking properties of Modelica models have been made: In [22], mainly the technical issues are discussed for analyzing Modelica model properties such as naming conventions and inheritance complexity. However, no integrated approach for expressing such properties is mentioned. Furthermore, analysis techniques for specific facets of Modelica models can be found, e.g. in [5] and [6], where the determination of under and over constrained systems of equations is presented.

The implementation structure presented in this paper is based on the logic meta programming approach described in [23], where the intimate correlation between aspect orientation and logic programming is outlined. As already proposed by the OpenModelica Project [11] and the MetaModelica Language [20], an ANTLR parser is used to create an Abstract Syntax Tree (AST) for examining Modelica models under consideration. Thereupon, the RML/MetaModelica approach [19] can also be seen as a kind of strongly typed logic programming language for investigating Modelica models, but it is not linked to aspect oriented principles. Finally, a first attempt of an AOP Compiler focusing on merging AspectJ-like *inter-type declarations* into the AST of Modelica models can be found in [1].

3. Static Aspect Language for Modelica

In this Section, a domain-specific static aspect language is defined by giving a complete formalized syntax and related semantics for obtaining join points in Modelica models via quantified point cut expressions. Furthermore, variable capabilities are proposed for adequate binding of (typed) join points according to related Modelica language entities.

3.1 Applying Static Aspects to Modelica

First, the requirements for a static aspect language for Modelica is given to motivate the language design decisions. As an object-oriented language, Modelica benefits from component-based and inheritance principles fitting well with real structures from the problem domain thus being seamlessly transferable to the solution domain. The equation-based specifications of components' inertia allow for declarative and encapsulated behavioral descriptions detached from the system context. As a consequence, Modelica breaks a system down into smaller units of structure and behavior which is supported by according language constructs.

When aiming at a language for investigating static aspects of Modelica models, one starts with these units stating the *primitives* (atoms, terminals, etc.) as well-known starting points. A distinction between *unary* and *binary* primitives of Modelica models can be made, where the first category comprises high level units such as packages, classes, connectors, etc., therefore stating single, isolated entities. Instead, binary primitives are relations that group two kinds of units by a certain (semantic) criterion, e.g. all components of a model or all unknowns of an equation. On this basis, the aspect language must allow for expressing and modularization of combined, unit-spanning primitives by appropriate operators. Hereby, arbitrary complex relations (the static aspects) can be iteratively derived from simpler ones, always starting with the aforementioned primitives. Remind again the balanced model properties of Modelica 3: To ensure these properties, the inner structure of models and their connectors are to be considered on each hierarchy level.

To demonstrate the application of the syntactical constructs of the static aspect language introduced in the following, a simple Modelica model is provided as a running example consisting of a simple `Pin` for an electrical circuit as

```
connector Pin
  Voltage v;
  flow Current i;
end Pin;
```

For electrical components with two pins, a corresponding port "interface" model given as

```
partial model OnePort
  Pin p, n;
  Voltage v;
  Current i;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end OnePort;
```

defines the fundamental relations between quantities of electrical circuits. Being `partial`, the port model is to be derived by a concrete electrical component, e.g. an ideal resistor such as

```
model Resistor
  extends OnePort;
  parameter Real R(unit="Ohm");
equation
  R*i = v;
end Resistor;
```

characterized by a resistance parameter `R` and behavior according to Ohm's law. A simple `Circuit` model

```
model Circuit
  Resistor R1(R=100), R2(R=200);
equation
  R1.n = R2.p;
end Circuit;
```

consists of two resistors connected in series.

3.2 Syntax

The following language is specifically tailored to extend Modelica by static aspects and consists of two sublanguages for modularizing aspect definitions:

- A language for specifying a set of *join points*, hence static elements of interest within the model under consideration. The set of join points to be obtained is defined by a *point cut* expression that can be composed out of Modelica primitives (fundamental types of language units) and appropriate operators for combination.
- An action language for defining *advices*, therefore stating what to be done with the join points previously calculated as result of the point cut. The actual content of an advice might vary from the simple output of an error message in case of using the language for checking design rules, to arbitrary complex modifications of the join points. The latter entails static aspect weaving capabilities affecting the inspected model, e.g. by refactorings.

Accordingly, the language allows for specifying a series of rules of the form:

```
<Point cut> => <Advice>;
```

constituting *point cut – advice*-pairs with well-defined syntactical structure as will be given in this Section, thereby focusing on the first part.

Point cuts describe specific classes of elements in Modelica models referring to the language's basic constructs such as class definitions (types), components (members of classes), and equations. According to static aspects, point cuts constitute definite points within the model code to be considered in the aspect advice. The *join points* that match to a point cut are those complying with the predicates the point cut is composed of by combinations of logical and quantified expressions. Through this declarative approach of describing "requirements" for model elements under investigation, these point cut expressions are decoupled from a specific model and allow for a quantified model inspection that can be applied to arbitrary Modelica models without knowing inner details.

Due to the obvious relation of the concept to the logic paradigm, a point cut expression can be construed as a set of predicate definitions to be evaluated on a Modelica source of interest (set of model definitions). The therein contained set of model-specific join points is adjusted stepwise by iteratively processing the point cut subterms, and finally resulting in a set of join points fulfilling the overall point cut claims. The complete syntax for the point cut language is depicted in Figure 1. The syntactical structure is inspired by the PDL in [16], but modified in some parts, especially concerning parameterized point cuts which will be introduced in detail below. Meaning and application of the different constituents will be described in the following Sections.

$$
\begin{array}{rcl}
p & ::= & u \\
 & | & b(p) \\
 & | & p \ \textbf{and} \ p \\
 & | & p \ \textbf{or} \ p \\
 & | & \textbf{not} \ p \\
 & | & p \ \textbf{equals} \ p \\
 & | & p \ \textbf{less} \ p \\
 & | & p \ \textbf{subset} \ p \\
 & | & \textbf{exists} \ b : p \\
 & | & \textbf{forall} \ b : p \\
 & | & \textit{relop} \ \ n \ b \\
 & | & [v := p]\textit{relop} \ b \ b \\
u & ::= & \textit{id} \\
 & | & '\textit{pattern}' \\
b & ::= & \textit{id} \\
 & | & b+ \\
 & | & b+n \\
 & | & p \ \textbf{product} \ p \\
 & | & p \ \textbf{product-d} \ p \\
p & : & \textit{Point Cut} \\
u & : & \textit{Unary Point Cut} \\
b & : & \textit{Binary Relation} \\
id & : & \textit{Identifier} \\
n & : & \textit{Natural Number} \\
relop & : & \textit{Relational Operator} \\
pattern & : & \textit{Name Pattern}
\end{array}
$$

Figure 1. Syntax of the Point Cut Language

3.2.1 Primitives

Starting the inspection of a Modelica input source by considering the set of all join points present, the point cut language provides some fundamentals *primitives* for a first limitation of the join point set. These predefined primitive terms constitute subsets of the overall join point set to those of a certain kind and/or within a certain context with respect to elementary structuring constructs of Modelica.

As mentioned in [16], primitives might either be classified as *unary* or *binary* which depends on the number of join points to be considered for a match. Binary primitives express some property of join points. The choice of appropriate unary primitives focuses on major entities in which models are organized: packages and class types. Such individuals represent first class members, hence key paradigm concepts of Modelica. Tables 1 and 2 list sets of unary primitives which are simply accessed by their names. The most general primitive `class` includes all specialized class type definitions, namely `model`, `connector`, `package`, `block`, `type`, `record` and `function` [2]. All of whom are Modelica primitives on their part, therefore partitioning the set of class types into disjoint subsets. Applied to the `Circuit` example, `class` matches to `Pin`, `OnePort`, `Resistor`, and `Circuit`, whereas `connector` only selects the `Pin`. The (sub-) set of partial class type definitions can be obtained by the primitive `partialType`, and the final types correspondingly. Hence, `partialType` matches to `OnePort`, whereas the predicate `finalType` is matched nowhere in the sample model. The `localType` primitive matches local class instances nested within a model, e.g. consider an additional local model declaration

Primitive	Matches
`class`	all class types defined in the source
`model`	model types defined in the source
`connector`	connector types defined in the source
`package`	package types defined in the source
`block`	block types defined in the source
`type`	data types defined in the source
`record`	record types defined in the source
`function`	function types defined in the source

Table 1. Unary Modelica Primitives for basic Class Types

Primitive	Matches
`partialType`	partial types defined in the source
`finalType`	final types defined in the source
`localType`	local types defined within a type

Table 2. Unary Modelica Primitives for specialized Types

```
replaceable model Res = Resistor;
```

within `Circuit` for parameterized typing of circuit elements. Primitive `localType` matches to the type of `Res` which actually refers to the global type `Resistor` in this example.

Binary primitives match pairs of join points, therefore expressing binary relations. They can be used for incrementally deriving interrelations, therefore inspecting further properties of model elements, e.g. relating a member variable or nested component and its surrounding class type. As unary Modelica primitives were defined on the high-level type structure, now binary primitives allow for a detailed inspection of the internal properties of a class, namely the parts for structure (variables/component members, inheritance, accessibility etc.) and behavior (equations). Again, binary primitives are given by a name followed by a parameter `p` stating the kind of join point, it is related to. Tables 3 to 7 contain structural, and Table 8 behavioral unary primitives for Modelica models. The primitives in Table 3 allow for structural insights of a given type `p` by accessing its members. These might either be `primitiveMember` or components, thus class-typed variables. For the circuit example, `primitiveMember(model)` results in the resistance variable `R` from model `Resistor`[1]. Further partitioning of members is done by their dimension (vectors,

[1] Note: The variables v and i from Pin and OnePort are typed by an according (non-primitive) Type declaration, thus stating component members.

Primitive	Matches
`member(p)`	all members of a type that matches p
`primitiveMember(p)`	primitive members of a type that matches p
`componentMember(p)`	components of a type that matches p
`vectorMember(p)`	vector members of a type that matches p
`matrixMember(p)`	matrix members of a type that matches p
`publicMember(p)`	public members of a type that matches p
`protectedMember(p)`	protected members of a type that matches p
`replMember(p)`	polymorphic members of a type that matches p
`replType(p)`	local class member of a type that matches p
`constrType(p)`	upper bound type of a replaceable type that matches p

Table 3. Binary Modelica Primitives for Type Members

matrices), and visibility. Note that members not explicitly stated as public or protected in a model are assumed to be public, e.g. `public(connector)` results in `v,i`. A Modelica specific concept is that of *replaceable* members, explicit polymorphic members, and replaceable local types for type parameterization. The `replMember(model)` primitive for example would match components such as:

```
replaceable OnePort op;
```

within `Circuit`. This component can for instance be replaced by `Resistor` via a modifier on instantiation. For `replType` consider again the replaceable resistor model example and the aforementioned `localType` primitive: In contrast, `replType(model)` matches the member variable `Res` itself instead of its referenced type which *is* actually the type of components within `Resistor` typed as `Res`. Finally, `constrType(model)` matches types constraining replaceable types, i.e. for

```
replaceable model Res
          = Resistor extends OnePort;
```

the point cut expression

```
constrType(replType(model))
```

results in `OnePort`. The `primModifier(p)` primitive in Table 4 matches modifier values for parameter members `p` applied when its surrounding class is instantiated, e.g.

```
primModifier(primitiveMember(model))
```

matches `100,200` as `Resistor` is instantiated twice within `Circuit` with corresponding modifier values for the parameter `R`. The primitive `compModifier(p)` matches types used for redeclaration of replaceable components `p` when its surrounding class type is being instantiated, e.g. in

Primitive	Matches
`primModifier(p)`	modifier of a primitive parameter member that matches p
`compModifier(p)`	modifier of a replaceable component that matches p
`typeModifier(p)`	redeclaration type of a replaceable type

Table 4. Binary Modelica Primitives for Modification and Redeclaration

```
Circuit circuit(
        redeclare OnePort Resistor);
```

the modifier `Resistor` for the replaceable `OnePort` matches this predicate. In case of redeclaration of local class types `p` such as

```
Circuit circuit(
        redeclare Model Res =
        Capacitor);
```

the new type parameter assigned to `Res` can be obtained by

```
typeModifier(replType(model))
```

thus resulting in `Capacitor`. Table 5 lists primitives

Primitive	Matches
`derivedType(p)`	types that are derived from a type that matches p
`baseType(p)`	types that are derived by a type that matches p
`subType(p)`	types that are subtypes of a type that matches p
`varDerivedType(p)`	types that are variably derived from a local type that matches p

Table 5. Binary Modelica Primitives for Inheritance Hierarchies

for obtaining inheritance relations between types. The result of `derivedType(partial)` is `Resistor` (direct subclass relation), and `baseType(model)` inversely matches `OnePort` as the direct super class of `Resistor`. In Modelica, a distinction is made between explicit subclasses and implicit subtypes within the type hierarchy of a system model. The primitive `subType` matches all types with public interfaces being compatible with that of type `p`. Note that this primitive is quite powerful as it is not directly extractable from a model source, but rather requires additional computational efforts. A further advanced construct of Modelica is variable inheritance. Consider the modified `OnePort` model:

```
model OnePort
  replaceable model Res = Resistor;
  ...
protected
  extends Res;
```

```
  ...
end OnePort;
```

Here, the local class type `Res` can be used to redeclare the base class of `OnePort` which is initially stated as `Resistor`. Thus, the result of the expression

```
varDerivedType(localType)
```

where `localType` matches `Resistor`, is `OnePort`. The primitives depicted in Table 6 allow for inspection of

Primitive	Matches
`flow(p)`	flow members of a type that matches p
`input(p)`	input members of a type that matches p
`output(p)`	output members of a type that matches p
`constant(p)`	constant members of a type that matches p
`parameter(p)`	parameter members of a type that matches p
`inner(p)`	inner members of a type that matches p
`outer(p)`	outer members of a type that matches p
`final(p)`	final members of a type that matches p
`discrete(p)`	discrete members of a type that matches p

Table 6. Binary Modelica Primitives for Member Properties

further member properties. Being mostly self-explanatory, a detailed description shall be omitted at this point.

Primitive	Matches
`startValue(p)`	start value of a member that matches p
`fixedStartValue(p)`	fixed value of a member that matches p

Table 7. Binary Modelica Primitives for Member Initialization

Modelica allows for the propagation of start values and initialization values for primitive member variables. The primitives in Table 7 can be used to obtain such values. The primitives listed in Table 8 can be used to delve into model bodies and explore the behavioral specifications in equations[2]. As `equation(p)` matches all kinds of equations defined for type `p`, e.g. `R*i = v;` for `Resistor`, further primitives for partitioning the set of equations are given, i.e. initial equations, equations for connecting two connector components, equations containing `if` or `when` clauses (potentially causing events), and `for` loops. After having selected a certain equation, the `unknown` primitive can be used to get the set of variables appearing in an equation, hence

[2] Note: Algorithm parts can also constitute model behavior, but are no further considered at this point.

Primitive	Matches
`equation(p)`	equations defined in a type that matches p
`initEquation(p)`	initial equations in a type that matches p
`connectEquation(p)`	connect equations in a type that matches p
`ifEquation(p)`	equations containing `if` in a type that matches p
`whenEquation(p)`	eq. containing `when` in a type that matches p
`forEquation(p)`	equations containing `for` in a type that matches p
`unknown(p)`	unknown variables in an equation p
`derivated(p)`	unknown variables derivated in an equation p

Table 8. Binary Modelica Primitives for Model Behavior

```
unknown(equation(p))
```

for `p` matching the `Resistor` results in `R,i,v`. Variables being derivated within an equation match the primitive `derivated`. On this basis, even more arbitrary complex primitives concerning equation details might be useful, e.g. inspecting operators, but shall be omitted at this point.

3.3 Operators

For the definition of extensive aspects, thus concerns that crosscut the entities of models, operators for complex term construction are provided for correlating Modelica primitives of initially separated model units. Being closed under the set of join points of the overall model, such operators allow for iterative combinations of point cut expressions permitting rules of any complexity. Some criteria to take into account when choosing appropriate operators:

- The resulting language's expressiveness must be sufficient for capturing a wide range of possibly occurring requirements.
- The operators must allow for an "atomic" conversion and efficient evaluation.
- The operators usage must be concise and intuitive.

As the operators are applied to *sets* of join points, they are mainly of set oriented nature inspired by logic programming and query languages. For operator precedences, the usage of appropriate parentheses is recommended as usual.

3.3.1 Logical Operators

For simple interrelations of join point sets, logical connectors are provided. For instance, classes, that are both subclasses derived from other classes *and* subtype of another type, can be searched via

```
derivedType(class) and subType(class)
```

As another example, the expression

```
input(class) or output(class)
```

matches all directed member variables. As one of the most powerful operators, the logical negation allows for expressions matching all join points *not* being contained in a given set of join points[3]. For instance

```
partial and not baseType(class)
```

matches all partial class types not being derived by any other class type. Moreover, the equality of two sets of join points can be stated, e.g.

```
class equals (model or connector)
```

requires systems only consisting of models and connectors. Two more operators for more intuitive descriptions of stepwise refinements for intermediate point cuts are provided:

```
model less partial
```

matches all model types *but* partial ones, whereas

```
partial subset model
```

matches *only those* model types that are also partial, which can actually also be expressed by the `and` operator.

3.3.2 Pattern

In order to examine join points that refer to named model elements, a pattern operator is provided. Using arbitrary pattern expressions, the set of join points can be reduced to the ones whose names match the given pattern. By

```
model and 'Resistor'
```

the resistor model can be obtained. Moreover, patterns can be used to check naming conventions, e.g. the demand *"Flow variables shall be named with a flow postfix"* can be expressed by

```
(flow(class) less '*_flow')
```

matching those flow members whose names do not have the required postfix. Besides wild cards `*` matching arbitrary sequences of symbols, further operators inspired e.g. by *regular expressions* can be used such as ranges `[a,b,c]` demanding one of the listed symbols. As there are primitives referring to unnamed join point types, e.g. equations, the pattern operator is only allowed at innermost position of expressions and no explicit comparison operator is provided.

3.3.3 Quantification

Point cut quantification can be used to check some condition on a *range* of values in a binary relation of join points. Therefore, point cut expressions can be constructed whose matching join points are based on properties of related join points. The expression

```
forall primitiveMember : output(block)
```

matches block types, whose primitive members are *all* output variables, i.e. sources. In this example, the property `output` (of a block) is postulated for all primitive members of that block (stated as a binary primitive relation). In contrast, the expression

```
exists primitiveMember : output(block)
```

matches blocks with *at least* one output variable.

3.3.4 Cardinality

Operators dealing with the cardinality of join point sets allow for evaluation of model metrics. Style conventions for structuring Modelica libraries such as *"The number of models defined in an own package must be at least 5"* can be expressed by

```
package and (> 5 componentMember)
```

resulting in "malformed" packages containing less than 5 type declarations. A cardinality expression can be parameterized by an optional point cut `[v:=p]`, hence a set of join points. Note that this concept will be generalized in Section 3.5 for application to all point cut expressions. In this way, comparisons of cardinalities concerning further properties between join points can be enforced. The complex balanced model demand *"The number of flow variables in a connector must be identical to the number of non-causal non-flow variables"* [17] can be stated as

```
[v:=connector](= flow(v)
(primitiveMember(v)
less (flow(v) or input(v) or output(v)
or parameter(v) or constant(v)))
```

constituting an iterative "foreach" loop over the set of connectors.

3.3.5 Composition

New binary relations b can be created by composing two point cuts by building the Cartesian product, hence combining all join points matching these point cuts. For instance, the expression

```
[v:=class](derivedType(v) product
derivedType(v))
```

relates types having the same (direct) super class. Here, again the parameterization syntax is used to relate both subclasses to the same super class. Note that the `product` operator also relates identical join points, thus creating reflexive relations. For avoiding such self-references, the `product-d` (disjoint) operator can be used. As a further example, the expression

```
[v:=connectEquation(model)]
(unknown(v) product-d unknown(v))
```

relates components being connected within models.

3.3.6 Transitive Closures

The deduction of (anti-symmetric) transitive closures of a binary relations can be obtained by

```
derivedType+
```

for instance, resulting in the subclass relation, hence relating two classes (indirectly) associated via arbitrary chains of derivations. The "bounded" transitive closure operator creates chains limited to at most n links, e.g.

```
derivedType+4
```

[3] Depending on the type system applied, this complemented set could be further limited to only those join points of the same type as the given ones.

can be used to check whether inheritance hierarchies deeper than 4 are present in the model by comparing the result to that of the unbounded case. The closure operator can also be used for examining connector traces of Modelica models.

3.4 Semantics

Partly taken from [16], the point cut evaluation is reduced to element-wise reasoning of join point sets considering the stipulated conditions. The evaluation of a point cut expression $p \in P$ from a set of rules P with respect to a Modelica model specification $\mathcal{M}$ (a collection of conjugated class type definitions) is stated as:

$$p : J_{\mathcal{M}} \rightarrow \mathcal{P}(J_{\mathcal{M}})$$

resulting in a (sub-) set of join points of the model under consideration, where $J_{\mathcal{M}}$ is assumed to be

$$J_{\mathcal{M}} = \{\text{sets of all types of join points present in } \mathcal{M}\}$$

The evaluation of a rule expression precedes from inwards to outwards, therefore calculating temporary join point sets as intermediate stages that are gradually refined toward the overall result set. Being composites of unary and binary expressions, point cuts are calculated through sequences of mappings

$$U : \textit{Unary Point Cut} \rightarrow \mathcal{P}(J_{\mathcal{M}})$$

for unary primitives u, and

$$B : \textit{Binary Relation} \rightarrow \mathcal{P}(J_{\mathcal{M}} \times J_{\mathcal{M}})$$

for binary primitives $b(p)$, respectively. The evaluation of u in a point cut expression can be simply stated as $P[\![u]\!] = U[\![u]\!]$, where u might be either a unary primitive denoted by *id* and will therefore be replaced by those join points $j \in J_{\mathcal{M}}$ matching *id*, or it is some kind of regular expression, thus

$$U[\!['pattern']\!] = \{j \mid j \textit{ matches 'pattern'}\}$$

is to be applied. For the evaluation of binary primitives $b(p)$, the given parameter p is to be taken into account:

$$P[\![b(p)]\!] = \{j_1 \mid (j_1, j_2) \in B[\![b]\!], j_2 \in P[\![p]\!]\}$$

The relation set b between the parameter join point p and the binary primitive is constructed by trying out all possible combinations and keeping those fulfilling b. The result, again is a set of single join points "fitting" to the parameter p. Unary and binary primitives can be nested (composed) at will, e.g. $b(b(p))$. The operators for logical combinations of join point sets can be reduced to according set operators:

$$\begin{aligned} P[\![p_1 \textbf{ and } p_2]\!] &= P[\![p_1]\!] \cap P[\![p_2]\!] \\ P[\![p_1 \textbf{ or } p_2]\!] &= P[\![p_1]\!] \cup P[\![p_2]\!] \\ P[\![p_1 \textbf{ less } p_2]\!] &= P[\![p_1]\!] \setminus P[\![p_2]\!] \\ P[\![\textbf{not } p_1]\!] &= \{j \mid j \notin P[\![p_1]\!]\} \end{aligned}$$

The operators for comparing sets of join points (equality and subsets) are evaluated as follows:

$$\begin{aligned} P[\![p_1 \textbf{ subset } p_2]\!] &= \{j \mid \forall j \in p_1 : j \in p_2\} \\ P[\![p_1 \textbf{ equals } p_2]\!] &= (p_1 \textbf{ subset } p_2) \textbf{ and } (p_2 \textbf{ subset } p_1) \end{aligned}$$

Hence, `subset` results in the join point p_1, iff $p_1 \subseteq p_2$ and in the empty set, otherwise. The equality of two join point sets can then be ensured via `equals` by checking the result set not being empty. For the quantification operators, the semantics are given as

$$\begin{aligned} P[\![\textbf{forall } b : p]\!] &= \{j_2 \mid \forall (j_1, j_2) \in B[\![b]\!] : j_1 \in P[\![p]\!]\} \\ P[\![\textbf{exists } b : p]\!] &= \{j_2 \mid \exists (j_1, j_2) \in B[\![b]\!] : j_1 \in P[\![p]\!]\} \end{aligned}$$

Binary relations constitute relationships between two join points and can be obtained by according primitives *id*:

$$B[\![id]\!] = \textit{<primitive>}$$

Further relations can be constructed as Cartesian products:

$$P[\![p_1 \textbf{ product } p_2]\!] = \{(j_1, j_2) \in P[\![p_1]\!] \times P[\![p_2]\!]\}$$

For the `product-d` operator, the requirement $j_1 \neq j_2$ must be satisfied. The semantics for transitive closure is defined to be

$$P[\![b+]\!] = \{(j_1, j_k) \mid \exists (j_1, j_2), \ldots, (j_{k-1}, j_k) \in B[\![b]\!]\}$$

where $k \leq n$ must hold in the bounded case for $P[\![p + n]\!]$.

3.5 Variable Binding and Type System

As already used for the cardinality comparison operator, for further examinations of elements within other subterms, variable bindings are introduced. They allow for binding of join points to variables which can be used as parameters in subsequent terms. This concept shall now be enhanced to all kinds of point cut expressions. Generally, a point cut expression can be parameterized by an arbitrary set of point cut variables ϕ that are then visible within the point cut:

$$[\phi]p : \textit{Point cut}_{\phi}, \text{ where } \phi = \{v_1 := p_1, \ldots, v_n := p_n\}$$

is a set of n point cut variables v_i, whose content is again defined by point cut expressions p_i. In the modified semantics, these parameters are passed to all subterms of p, e.g.

$$P[\![p_1 \textbf{ and } p_2]\!]_{\phi} = P[\![p_1]\!]_{\phi} \cap P[\![p_2]\!]_{\phi}$$

These enhanced evaluation semantics constitutes a (nested) "for-each" loop over the set of join point combinations depicted by the parameter point cut(s) to be likewise adopted to all point cut subterms.

As proposed in [16], the adoption of *types* for point cuts allows for sound expressions with respect to the types expected for the matching join points. Such types reflect the basic kinds of Modelica constructs the join points refer to, namely types, members (primitive, components), scalar values, and equations. The integration of a type system for point cuts into the aforementioned semantics allows for

exact determination of the join point types in the result set. Therefore, the matching join points are restricted to those, that are in the static type calculated for the point cut. Due to lack of space, a formalization of this approach, especially considering types of parameter variables for point cuts, is deferred to future work.

3.6 Advices

The *advice* part of a rule shall only be discussed informally at this point. Generally, it is considered to be "executable" and it is applied for each join point matching the point cut part of a static aspect. For the simplest case, e.g. rule checking, a report string can be put out as an error description:

```
<point cut> =>
"Error: violated naming convention";
```

For providing a more expressive report, access to the resulting join point set should be made possible, e.g. by the following syntax for iterating the result set:

```
<point cut> =>
"Error: violated naming convention in "
+ ResultSet.nextItem().getName();
```

returning the name of the join point within the result set currently iterated. Note that the join points within `ResultSet` must be of a type that refers to a named element in Modelica. As a next step, arbitrary code as well as access to additional variables defined in the point cut could be permitted, e.g. for obtaining the corresponding AST Nodes for manipulation within a compiler environment.

4. Implementation Framework

A sample implementation framework of the static aspect language that is currently under construction is proposed in the following. It is based on the principles of *logic meta programming* [23]. The application of two languages is proposed for processing static aspects on given Modelica models: (1) A *User* language for specifying static aspects, such as the previously defined point cut language which serves as an inspection API for Modelica models, and (2) an efficient *Implementation* language for processing of the aspects, therefore stating a point cut evaluation engine. Due to the similarities of aspect principles and the logic paradigm [23], the first order logic programming language Prolog [9] can serve as an evaluation engine. Prolog allows for seamless conceptual representation and efficient evaluation of aspect queries on Modelica models in terms of logical facts and rules. A structural overview of the resulting implementation framework architecture is depicted in Figure 2. The front end for user input parsing and transformation consists of two interfaces:

1. A Modelica model input interface realized as a conventional ANTLR [11, 18] Modelica Parser that constructs the Abstract Syntax Tree (AST) representation of the model under consideration and extracts primitives in terms of Prolog facts. A similar approach is taken in MetaModelica [20].

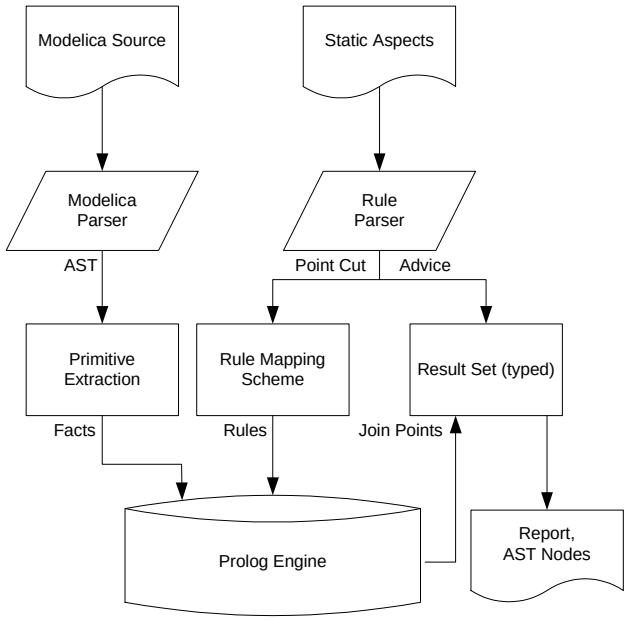


Figure 2. Architecture of the Static Aspects Framework for Modelica Models

2. An aspect rule input interface accepting syntactically well-formed static aspects expressions according to the aforementioned point cut language grammar. The point cut expression parts are then transformed to suitable Prolog rules according to a mapping scheme for iteratively composing primitives and operators.

The middle end forms the aspect processing engine in terms of a Prolog interpreter implemented on top of the Modelica source code parser. Applying the Prolog rules generated for point cut expressions to the fact basis containing the model primitives, the engine integrates both, the input models and the related aspect rules for join point result set processing. The back end interface conducts advice processing with respect to the resulting join point set, i.e.

- Error reports for simple design rule evaluation,
- References to the originating AST nodes of the resulting join points, thus allowing for arbitrary post-processing of complex aspect advices detached from the framework, e.g. as described in [8].

A simple example for mapping model structures to corresponding Prolog rules shall be given. Consider the aforementioned `Resistor` model inheriting from the partial `OnePort` model. First, the "existence" of both models can be expressed by appropriate Prolog facts:

```
model(m1,'OnePort').
model(m2,'Resistor').
```

Next, the interrelation of both models with respect to the implied inheritance hierarchy can be stated as:

```
derive(m2,m1).
```

According to these facts, implications can be derived by appropriate rules, e.g.:

```
derivedType(Sub,Sup)
            :- derive(Sub,Sup).
derivedType(Sub,Sup)
            :- derive(Sub,X),
               derivedType(X,Sup).
```

for calculation of the transitive closure of the inheritance hierarchy.

The decoupling of the *User* and *Implementation* language aims at a high grade of extensibility and adaptability. Being Turing complete, the Prolog engine allows for integrating point cut language constructs of any complexity, and does not dictate the concrete representation of the aspect language implementation. The implementation can either be used as a "stand-alone" rule checker for systematic model inspection, or it can be integrated to an ambient Modelica compiler/development environment accomplishing static aspect *weaving* e.g. AST transformations.

5. Application

On the basis of the adaptable and scalable framework implementation and the flexibility of the proposed static aspect language for Modelica, various possible areas of application are conceivable:

1. Rule checking "by negation": Describing point cuts matching join points that are not desired to appear in the models as proposed in [16]. In case of non empty result sets, the rule is violated by the join points calculated and corresponding error reports can be generated in the advice part. By expressing new restrictions of the Modelica 3 specification as static aspects, the compatibility of legacy code such as libraries can be examined automatically.
2. Model inspection: Searching for model elements or patterns matching criteria of interest, either "off-line" (e.g. metrics calculations), or "on-line" as a model inspection tool within a Modelica IDE. Further applications in the realm of the object oriented paradigm might be that of *concepts* [13], thus checking whether a given set of types are applicable as parameters for a generic construct (upper bound resolution for constraining types).
3. Join point manipulations within the advice part, e.g. renaming of certain elements with respect to naming conventions or model maintenance.
4. Arbitrary model restructuring by join points referencing nodes of the AST. Therefore, static aspect weaving can be done by graph transformation, e.g. context aware refactorings.

6. Conclusion

The formal syntax and semantics definition and sample implementation framework of a static aspect language for Modelica was presented. The language design aims at sufficient expressiveness and extensibility, but yet still provides intuitive usage. Language extensions for variable bindings of (typed) join points were mentioned for enhanced rule precision.

The framework proposed can serve as a foundation for a wide range of applications, e.g. simple rule checking up to source code manipulations. An integration into existing environments is aimed at, either as a basis for a point cut evaluation engine for a Modelica AOP Compiler, or as a programmer's on-line assistance tool for code inspection queries, e.g. providing an interactive search engine with Eclipse IDE integration.

In future work, after having finished the implementation, the application of the language in various case studies can indicate, whether the language boundaries defined up to this point are sufficient. For this purpose, the formulation of balanced model requirements of Modelica version 3 in terms of static aspects is assumed to be a convenient case study. Hereby, the performance of the implementation can be investigated concerning the number and complexity of rules and models under investigation. Optimizations can lead to increased efficiency, e.g. by dynamical and cached AST access on demand during rule evaluation. Moreover, a detailed survey of the advice part is aimed at, especially in view of conflicts analysis between different aspects. On this basis, studies of applying dynamic aspect approaches to Modelica can be promising, although Modelica-like language are not *procedural* ones as demanded e.g. in [15].

Acknowledgments

The authors' thank goes to Dr. Michaela Huhn and TLK Thermo for helpful discussions supporting this paper.

References

[1] Johan Akesson. *Languages and Tools for Optimization of Large-Scale Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, November 2007.

[2] The Modelica Association. *The Modelica Language Specification 3.0*. http://www.modelica.org, September 2007. http://www.modelica.org.

[3] Franz Baader, Diego Calvanese, Deborah L. McGuiness, Daniele Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[4] Alexander Borgida. Description Logics in Data Management. In *IEEE Transactions on Knowledge and Data Engineering*, volume 7, pages 671–682, 1995.

[5] David Broman, Kaj Nyström, and Peter Fritzson. Determining Over- and Under-Constrained Systems of Equations using Structural Constraint Delta. In *Proceedings of the Fifth International Conference on Generative Programming and Component Engineering (GPCE'06)*, pages 151–160, Portland, Oregon, USA, 2006. ACM Press.

[6] Peter Bunus and Peter Fritzson. Automated Static Analysis of Equation-Based Components. *SIMULATION*, 80(7-8):321–345, July-August 2004.

[7] Tal Cohen, Joseph Gil, and Italy Maman. JTL - The Java Tools Language. In *OOPSLA'06: Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2006.

[8] Roger F. Crew. ASTLOG: A Language for Examinging Abstract Syntax Trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 229–242, 1997.

[9] P. Deransart, A. Ed-Dbali, and L. Ceravoni. *Prolog: The Standard*. Springer-Verlag, NewYork, 1996.

[10] R. Filman and D. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness, 2000.

[11] P. Fritzson, P. Aronsson, A. Pop, H. Lundvall, K. Nystrom, L. Saldamli, D. Broman, and A. Sandholm. OpenModelica - A Free Open-Source Environment for System Modeling, Simulation, and Teaching. In *IEEE International Symposium on Computer-Aided Control Systems Design*, pages 1588–1595, October 2006.

[12] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. IEEE Press, 2004.

[13] J. Järvi, J. Willcock, and A. Lumsdaine. Associated Types and Constrain Propagation for Mainstream Object-Oriented Generics. In *Proceedings of the 20th OOPSLA*, pages 327–355. Springer Verlag, June 2001.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. *Lecture Notes in Computer Science*, 2027:327–355, 2001.

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland*. Springer-Verlag, June 1997.

[16] Clint Morgan, Kris De Volder, and Eric Wohlstadter. A Static Aspect Language for Checking Design Rules. In *AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, pages 63–72, New York, NY, USA, 2007. ACM.

[17] Hans Olsson, Martin Otter, Sven Erik Mattsson, and Hilding Elmqvist. Balanced Models in Modelica 3.0 for Increased Model Quality. In Prof. Dr. B. Bachmann, editor, *Proceedings of the 6th International Modelica Conference*, volume 1, University of Applied Sciences Germany, Bielefeld, 2008. The Modelica Association.

[18] Terence Parr. *The Definitive ANTLR Reference Guide: Building Domain-specific Languages*. Pragmatic Programmers, 2007.

[19] Adrian Pop and Peter Fritzson. Debugging Natural Semantics Specifications. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 77–82, New York, NY, USA, 2005. ACM.

[20] Adrian Pop and Peter Fritzson. MetaModelica: A Unified Equation-Based Semantical and Mathematical Modeling Language. In *Joint Modular Languages Conference (JMLC2006)*, Jesus College, Oxford, England, September 2006.

[21] Michael Tiller. *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers, 2001.

[22] Michael Tiller. Parsing and Semantics Analysis of Modelica Code for Non-Simulation Applications. In Peter Fritzson, editor, *Proceedings of the 3rd International Modelica Conference*, volume 1, pages 411–418, Linköping, November 2003.

[23] Kris De Volder and Theo D'Hondt. Aspect-Oriented Logic Meta Programming. In *Reflection '99: Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*, pages 250–272, London, UK, 1999. Springer-Verlag.

Higher-Order Acausal Models

David Broman Peter Fritzson

Department of Information and Computer Science, Linköping University, Sweden
{davbr,petfr}@ida.liu.se

Abstract

Current equation-based object-oriented (EOO) languages typically contain a number of fairly complex language constructs for enabling reuse of models. However, support for model transformation is still often limited to scripting solutions provided by tool implementations. In this paper we investigate the possibility of combining the well known concept of higher-order functions, used in standard functional programming languages, with acausal models. This concept, called Higher-Order Acausal Models (HOAMs), simplifies the creation of reusable model libraries and model transformations within the modeling language itself. These transformations include general model composition and recursion operations and do not require data representation/reification of models as in metaprogramming/metamodeling. Examples within the electrical and mechanical domain are given using a small research language. However, the language concept is not limited to a particular language, and could in the future be incorporated into existing commercially available EOO-languages.

Keywords Higher-Order, Acausal, Modeling, Simulation, Model Transformation, Equations, Object-Oriented, EOO

1. Introduction

Modeling and simulation have been an important application area for several successful programming languages, e.g., Simula [6] and C++ [24]. These languages and other general-purpose languages can be used efficiently for discrete time/event-based simulation, but for continuous-time simulation, other specialized tools such as Simulink [15] are commonly used in industry. The latter supports causal block-oriented modeling, where each block has defined input(s) and output(s). However, during the past two decades, a new kind of language has emerged, where differential algebraic equations (DAEs) can describe the continuous-time behavior of a system. Moreover, such languages often support hybrid DAEs for modeling combined continuous-time and discrete-time behavior.

2nd International Workshop on Equation-Based Object-Oriented Languages and Tools. July 8, 2008, Paphos, Cyprus.

These languages enable modeling of complex physical systems by combining different domains, such as electrical, mechanical, and hydraulic. Examples of such languages are Modelica [10, 17], Omola [1], gPROMS [3, 20], VHDL-AMS [5], and χ (Chi) [13, 27].

A fundamental construct in most of these languages is the *acausal model*. Such a model can encapsulate and compose both continuous-time behavior in form of DAEs and / or other interconnected sub-models, where the direction of information flow between the sub-models is not specified. Several of these languages (e.g., Modelica and Omola) support object-oriented concepts that enable the composition and reuse of acausal models. However, the possibilities to perform *transformations* on models and to create generic and reusable transformation libraries are still usually limited to tool-dependent scripting approaches [7, 11, 26], despite recent development of metamodeling/metaprogramming approaches like MetaModelica [12].

In functional programming languages, such as Haskell [23] and Standard ML [16], standard libraries have for a long time been highly reusable, due to the basic property of having functions as first-class values. This property, also called *higher-order functions*, means that functions can be passed around in the language as any other value.

In this paper, we investigate the combination of acausal models with higher-order functions. We call this concept *Higher-Order Acausal Models (HOAMs)*.

A similar idea called *first-class relations on signals* has been outlined in the context of functional hybrid modeling (FHM)[18]. However, the work is still at an early stage and it does not yet exist any published description of the semantics. By contrast, our previous work's main objective has been to define a formal operational semantics for a subset of a typical EOO language [4]. From the technical results of our earlier work, we have extracted the more general ideas of HOAM, which are presented in this paper in a more informal setting.

An objective of this paper is to be accessible both to engineers with little functional language programming background, as well as to computer scientists with minimal knowledge of physical acausal modeling. Hence, the paper is structured in the following way to reflect both the broad intended audience, as well as presenting the contribution of the concept of HOAMs:

- The fundamental ideas of traditional higher-order functions are explained using simple examples. Moreover, we give the basic concepts of acausal models when used for modeling and simulation (Section 2).
- We state a definition of higher order acausal models (HOAMs) and outline motivating examples. Surprisingly, this concept has not been widely explored in the context of EOO-languages (Section 2).
- The paper gives an informal introduction to physical modeling in our small research language called Modeling Kernel Language (MKL) (Section 3).
- We give several concrete examples within the electrical and mechanical domain, showing how HOAMs can be used to create highly reusable modeling and model transformation/composition libraries (Section 4).

Finally, we discuss future perspectives of higher-order acausal modeling (Section 5), and related work (Section 6).

2. The Basic Idea of Higher-Order

In the following section we first introduce the well established concept of anonymous functions and the main ideas of traditional higher-order functions. In the last part of the section we introduce acausal models and the idea of treating models with acausal connections to be higher-order.

2.1 Anonymous Functions

In functional languages, such as Haskell [23] and Standard ML [16], the most fundamental language construct is functions. Functions correspond to partial mathematical functions, i.e., a function $f : A \rightarrow B$ gives a mapping from (a subset of) the domain A to the codomain B.

In this paper we describe the concepts of higher-order functions and models using a tiny untyped research language called *Modeling Kernel Language (MKL)*. The language has similar modeling capabilities as parts of the Modelica language, but is primarily aimed at investigating novel language concepts, rather than being a full-fledged modeling and simulation language. In this paper an informal example-based presentation is given. However, a formal operational semantics of the dynamic elaboration semantics for this language is available in [4].

In MKL, similar to general purpose functional languages, functions can be defined to be *anonymous*, i.e., the function is defined without an explicit naming. For example, the expression

```
func(x){x*x}
```

is an anonymous function that has a formal parameter `x` as input parameter and returns `x` squared[1]. Formal parameters are written within parentheses after the `func` keyword, and the expression representing the body of the function is given within curly parentheses; in this case `{x*x}`.

An anonymous function can be applied by writing the function before the argument(s) in a parenthesized list, e.g. `(3)`:

```
func(x){x*x}(3)
→ 3*3
→ 9
```

The lines starting with a left arrow (→) show the evaluation steps when the expression is executed.

However, it is often convenient to name values. Since anonymous functions are treated as values, they can be defined to have a name using the `def` construct in the same way as constants.

```
def pi = 3.14
def power2 = func(x){x*x}
```

Here, both `pi` and function `power2` can be used within the defined scope. Hence, the definitions can be used to create new expressions for evaluation, for example:

```
power2(pi)
→ power2(3.14)
→ 3.14 * 3.14
→ 9.8596
```

2.2 Higher-Order Functions

In many situations, it is useful to pass a function as an argument to another function, or to return a function as a result of executing a function. When functions are treated as values and can be passed around freely as any other value, they are said to be *first-class citizens*. In such a case, the language supports *higher-order functions*.

DEFINITION 1 (Higher-Order Function).
A higher-order function is a function that

1. *takes another function as argument, and/or*
2. *returns a function as the result.*

Let us first show the former case where functions are passed as values. Consider the following function definition of `twice`, which applies the function `f` two times on `y`, and then returns the result.

```
def twice = func(f,y){
    f(f(y))
};
```

The function `twice` can then be used with an arbitrary function `f`, assuming that types match. For example, using it in combination with `power2`, this function is applied twice.

```
twice(power2,3)
→ power2(power2(3))
→ power2(3*3)
→ power2(9)
→ 9*9
→ 81
```

Since `twice` can take any function as an argument, we can apply `twice` to an anonymous function, passed directly as an argument to the function `twice`.

[1] In programming language theory, an anonymous function is called a *lambda abstraction*, written $\lambda x.e$, where x is the formal parameter and e is the expression representing the body of the function. The corresponding syntactic form in MKL for a lambda abstraction is `func` p{e}, where p is a *pattern*. A pattern can be a n-ary tuple enclosed in parenthesis, e.g., a tuple pattern with one parameter can have the form `(x)` and one with two parameters `(x,y)`.

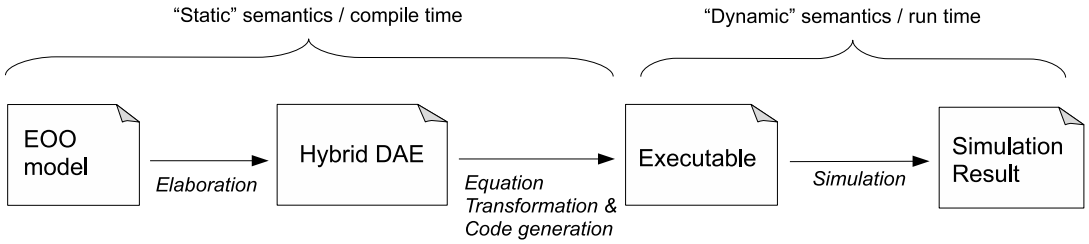


Figure 1. Outline of a typical compilation and simulation process for an EOO language tool.

```
twice(func(x){2*x-3},5)
→ func(x){2*x-3}(func(x){2*x-3}(5))
→ func(x){2*x-3}(2*5-3)
→ func(x){2*x-3}(7)
→ 2*7-3
→ 11
```

Let us now consider the second part of Definition 1, i.e., a function that returns another function as the result.

In mathematics, functional composition is normally expressed using the infix operator $\circ$. Two functions $f : X \to Y$ and $g : Y \to Z$ can be composed to $g \circ f : X \to Z$, by using the definition $(g \circ f)(x) = g(f(x))$.

The very same definition can be expressed in a language supporting higher-order functions:

```
def compose = func(g,f){
    func(x){g(f(x))}
};
```

This example illustrates the creation of a new anonymous function and returning it from the `compose` function. The function composes the two functions given as parameters to `compose`. Hence, this example illustrates both that higher-order functions can be applied to functions passed as arguments (using formal parameters `f` and `g`), and that new functions can be created and returned as results (the anonymous function).

To illustrate an evaluation trace of the composition function, we first define another function `add7`

```
def add7 = func(x){7+x};
```

and then compose `power2` and `add7` together, forming a new function `foo`:

```
def foo = compose(power2,add7);
→ def foo = func(x){power2(add7(x))};
```

Note how the function `compose` applied to `power2` and `add7` evaluates to an anonymous function. Now, the new function `foo` can be applied to some argument, e.g.,

```
foo(4)
→ func(x){power2(add7(x))}(4)
→ power2(add7(4))
→ power2(7+4)
→ power2(11)
→ 11*11
→ 121
```

The simple numerical examples given here only show the very basic principle of higher-order functions. In functional programming other more advanced usages, such as list manipulation using functions `map` and `fold`, are very common.

2.3 Elaboration and Simulation of Acausal Models

In conventional object-oriented programming languages, such as Java or C++, the behavior of classes is described using methods. On the contrary, in equation-based object-oriented languages, the continuous-time behavior is typically described using differential algebraic equations and the discrete-time behavior using constructs generating events. This behavior is grouped into abstractions called classes or models (Modelica) or entities and architectures (VHDL-AMS). From now on we refer to such an abstraction simply as *models*.

Models are blue-prints for creating *model instances* (in Modelica called components). The models typically have well-defined interfaces consisting of ports (also called connectors), which can be connected together using *connections*. A typical property of EOO-languages is that these connections usually are *acausal*, meaning that the direction of information flow between model instances is not defined at modeling time.

In the context of EOO languages, we define acausal (also called non-causal) models as follows:

DEFINITION 2 (Acausal Model).
An acausal model is an abstraction that encapsulates and composes

1. *continuous-time behavior in form of differential algebraic equations (DAEs).*
2. *other interconnected acausal models, where the direction of information flow between sub-models is not specified.*

In many EOO languages, acausal models also contain conditional constructs for handling discrete events. Moreover, connections between model instances can typically both express potential connections (across) and flow (also called through) connections generating sum-to-zero equations. Examples of acausal models in both MKL and Modelica are given in Figure 2 and described in Section 3.1.

A typical implementation of an EOO language, when used for modeling and simulation, is outlined in Figure 1. In the first phase, a hierarchically composed acausal model is *elaborated* (also called flattened or instantiated) into a hybrid DAE, describing both continuous-time behavior (DAEs) and discrete-time behavior (e.g., when-equations). The second phase performs *equation transformations and*

code generation, which produces executable target code. When this code is executed, the actual simulation of the model takes place, which produces a simulation result. In the most common implementations, e.g., Dymola [7] or OpenModelica [26], the first two phases occur during compile time and the simulation can be viewed as the run-time. However, this is not a necessary requirement of EOO languages in general, especially not if the language supports structurally dynamic systems (e.g., Sol [29], FHM [18], or MOSILAB [8]).

2.4 Higher-Order Acausal Models

In EOO languages models are typically treated as compile time entities, which are translated into hybrid DAEs during the elaboration phase. We have previously seen how functions can be turned into first-class citizens, passed around, and dynamically created during evaluation. Can the same concept of higher-order semantics be generalized to also apply to acausal models in EOO languages? If so, does this give any improved expressive power in such generalized EOO language?

In the next section we describe concrete examples of acausal modeling using MKL. However, let us first define what we actually mean by higher-order acausal models.

DEFINITION 3 (Higher-Order Acausal Model (HOAM)). *A higher-order acausal model is an acausal model, which can be*

1. *parametrized with other HOAMs.*
2. *recursively composed to generate new HOAMs.*
3. *passed as argument to, or returned as result from functions.*

In the first case of the definition, models can be parametrized by other models. For example, the constructor of a automobile model can take as argument another model representing a gearbox. Hence, different automobile instances can be created with different gearboxes, as long as the gearboxes respects the interface (i.e., type) of the gearbox parameter of the automobile model. Moreover, an automobile model does not necessarily need to be instantiated with a specific gearbox, but only *specialized* with a specific gearbox model, thus generating a new more specific model.

The second case of Definition 3 states that a model can reference itself; resulting in a recursive model definition. This capability can for example express models composed of many similar parts, e.g., discretization of flexible shafts in mechanical systems or pipes in fluid models.

Finally, the third case emphasizes the fact that HOAMs are first-class citizens, e.g., that models can be both passed as arguments to functions and created and returned as results from functions. Hence, in the same way as in the case of higher-order functions, generic reusable functions can be created that perform various tasks on arbitrary models, as long as they respect the defined types (interfaces) of the models' formal parameters. Consequently, this property enables *model transformations* to be defined and executed within the modeling language itself. For example, certain discretizations of models can be implemented as a generic function and stored in a standard library, and then reused with different user defined models.

Some special and complex language constructs in currently available EOO languages express part of the described functionality (e.g., the redeclare and for-equation constructs in Modelica). However, in the next sections we show that the concept of acausal higher-order models is a small, but very powerful and expressive language construct that subsumes and/or can be used to define several other more complex language constructs. If the end user finds this more functional approach of modeling easy or hard depends of course on many factors, e.g., previous programming language experiences, syntax preferences, and mathematical skills. However, from a semantic point of view, we show that the approach is very expressive, since few language constructs enable rich modeling capabilities in a relatively small kernel language.

3. Basic Physical Modeling in MKL

To concretely demonstrate the power of HOAMs, we use our tiny research language Modeling Kernel Language (MKL). The higher-order function concept of the language was briefly introduced in the previous section. In this section we informally outline the basic idea of physical modeling in MKL; a prerequisite for Section 4, which introduces higher-order acausal models using MKL.

3.1 A Simple Electrical Circuit

To illustrate the basic modeling capabilities of MKL, the classic simple electrical circuit model is given in Figure 2. Part (I) shows the graphical layout of the model and (II) shows the corresponding textual model given in MKL. For clarity to the readers familiar with the Modelica language, we also compare with the same model given as Modelica textual code (III).

In MKL, models are always defined anonymously. In the same way as for anonymous functions, an anonymous model can also be given a name, which is in this example done by giving the model the name `circuit`. The model takes zero formal parameters, given by the empty tuple (parenthesized list) to the right of the keyword `model`. The contents of the model is given within curly braces. The first four statements define four new *wires*, i.e., connection points from which the different components (model instances) can be connected.

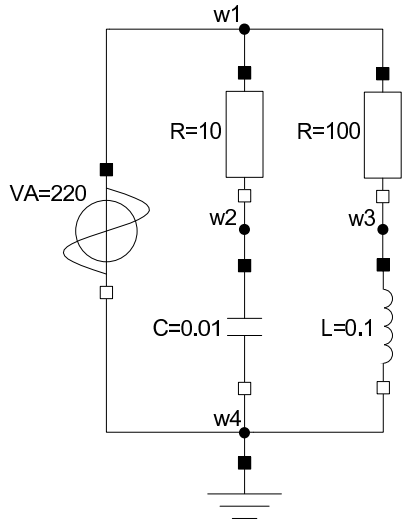
The six components defined in this circuit correspond to the layout given in part (I) in Figure 2. Consider the first resistor instantiated using the following:

```
Resistor(w1,w2,10);
```

The two first arguments state that wires `w1` and `w2` are connected to this resistor. The last argument expresses that the resistance for this instance is 10 Ohm. Wire `w2` is also given as argument to the capacitor, stating that the first resistor and the capacitor are connected using wire `w2`.

Modeling using MKL differs in several ways compared to Modelica (Figure 2, part III). First, models are not defined anonymously in Modelica and are not treated as first-class citizens. Second, the way acausal connections are de-

(I)



(II)

```
def Circuit = model(){
    def w1 = Wire();
    def w2 = Wire();
    def w3 = Wire();
    def w4 = Wire();
    Resistor(w1,w2,10);
    Capacitor(w2,w4,0.01);
    Resistor(w1,w3,100);
    Inductor(w3,w4,0.1);
    VSourceAC(w1,w4,220);
    Ground(w4);
};
```

(III)

```
model Circuit
    Resistor  R1(R=10);
    Capacitor C(C=0.01);
    Resistor  R2(R=100);
    Inductor  L(L=0.1);
    VsourceAC AC(VA=220);
    Ground    G;
equation
    connect(AC.p, R1.p);
    connect(R1.n, C.p);
    connect(C.n,  AC.n);
    connect(R1.p, R2.p);
    connect(R2.n, L.p);
    connect(L.n,  C.n);
    connect(AC.n, G.p);
end Circuit;
```

Figure 2. Model of a simple electrical circuit. Figure part (I) shows the graphical model of the circuit, (II) gives the corresponding MKL model definition, and (III) shows a Modelica model of the same circuit.

fined between model instances differs. In MKL, the connection (in this electrical case a wire), is created and then connected to the model instances by giving it as arguments to the creation of sub-model instances. In Modelica, a special `connect`-equation construct is defined in the language. This construct is used to define binary connections between connectors of sub-model instances. From a user point of view, both approaches can be used to express acausal connections between model instance. Hence, we let it be up to the reader to judge what is the most natural way of defining interconnections. However, from a formal semantics point of view, in regards to HOAMs, we have found it easier to encode connections using ordinary parameter passing style[2].

3.2 Connections, Variables, and Flow Nodes

The concept of wire is not built into the language. Instead, it is defined using an anonymous function, referring to the built-in constructs `var()` and `flow()`:

```
def Wire = func(){
    (var(),flow())
};
```

Here, a function called `Wire` is defined by using the anonymous function construct `func`. The definition states that the function has an empty formal parameter list (i.e., takes an empty tuple `()` as argument) and returns a tuple `(var(),flow())`, consisting of two elements. A tuple is expressed as a sequence of terms separated by commas and enclosed in parentheses.

[2] In the technical report [4], we have been able to define the elaboration semantics with HOAMs using an effectful small-step operational semantics. The main challenge of handling HOAMs and acausal connections concerns the treatment of flow variables and sum-to-zero equation. By using the parameter passing style, we avoid Modelica's informal semantic approach of using connection-sets. Moreover, by using this approach, the generated sum-to-zero equations implicitly gets the right signs, without the need of keeping track of outside/inside connectors.

The first element of the defined tuple expresses the creation of a new unknown continuous-time variable using the syntax `var()`. The variable could also been assigned an initial value, which is used as a start value when solving the differential equation system. For example, creating a variable with initial value 10 can be written using the expression `var(10)`. Variables defined using `var()` correspond to *potential* variables, i.e., the voltage in this example.

The second part of the tuple expresses the current in the wire by using the construct `flow()`, which creates a new flow-node. This construct is the essential part in the formal semantics of [4]. However, in this informal introduction, we just accept that Kirchhoff's current law with sum to zero at nodes is managed in a correct way.

In the circuit definition (Figure 2, part II) we used the syntax `Wire()`, which means that the function is invoked without arguments. The function call returns the tuple `(var(),flow())`. Hence, the `Wire` definition is used for encapsulating the tuple, allowing the definition to be reused without the need to restate its definition over and over again.

3.3 Models and Equation Systems

The main model in this example is already given as the `Circuit` model. This model contains instances of other models, such as the `Resistor`. These models are also defined using model definitions. Consider the following two models:

```
def TwoPin = model((pv,pi),(nv,ni),v){
    v = pv - nv;
    0 = pi + ni;
};
```

```
def Resistor = model(p,n,R){
    def (_,pi) = p;
    def v = var();
    TwoPin(p,n,v);
    R*pi=v;
};
```

In the same way as for `Circuit`, these sub-models are defined anonymously using the keyword `model` followed by a formal parameter and the model's content stated within curly braces. A formal parameter can be a pattern and *pattern matching* [3] is used for decomposing arguments. Inside the body of the model, definitions, components, and equations can be stated in any order within the same scope.

The general model `TwoPin` is used for defining common behavior of a model with two connection points. `TwoPin` is defined using an anonymous model, which here takes one formal parameter. This parameter specifies that the argument must be a 3-tuple with the specified structure, where `pv`, `pi`, `nv`, `ni`, and `v` are pattern variables. Here `pv` means positive voltage, and `ni` negative current. Since the illustrated language is untyped, illegal patterns are not discovered until run-time.

Both models contain new definitions and equations. The equation `v = pv - nv;` in `TwoPin` states the voltage drop over a component that is an instance of `TwoPin`. The definition of the voltage `v` is given as a formal parameter to `TwoPin`. Note that the direction of the causality of this formal parameter is not defined at modeling time.

The resistor is defined in a similar manner, where the third element `R` of the input parameter is the resistance. The first line `def (_,pi) = p;` is an alternative way of pattern matching where the current `pi` is extracted from `p`. The pattern _ states that the matched value is ignored. The second row defines a new variable `v` for the voltage. This variable is used both as an argument to the instantiation of `TwoPin` and as part of the equation `R*pi=v;` stating Ohm's law. Note that the wires `p` and `n` are connected directly to the `TwoPin` instance.

The inductor model is defined similarly to the `Resistor` model:

```
def Inductor = model(p,n,L){
    def (_,pi) = p;
    def v = var(0);
    TwoPin(p,n,v);
    L*der(pi) = v;
};
```

The main difference to the `Resistor` model is that the `Inductor` model contains a differential equation `L*der(pi) = v;`, where the `pi` variable is differentiated with respect to time using the built-in `der` operator.

The other sub-models shown in this example (`Ground`, `VSourceAC`, and `Capacitor`) is defined in a similar manner as the one above.

[3] A pattern can be a variable name, an underscore, or a tuple. When argument values are passed, each value is matched against its corresponding pattern. A variable is bound to the corresponding argument value, an underscore matches anything, i.e., nothing happens; a tuple is matched against a tuple value resulting in that each variable name in the tuple pattern is bound to the corresponding value in the tuple.

3.4 Executing the Model

Recall Figure 1, which outlined the compilation and simulation process for a typical EOO language. When a model is evaluated (executed) in MKL, this means the process of elaborating a model into a DAE. Hence, the steps of equation transformation, code generation, and simulation are not part of the currently defined language semantics. This latter steps can be conducted in a similar manner as for an ordinary Modelica implementation. Alternatively, the resulting equation system can be used for other purposes, such as optimization [14]. In the next section we illustrate several examples of how HOAMs can be used. Consequently, these examples concern the use of HOAMs during the elaboration phase, and not during the simulation phase. Further discussion on future aspects of HOAMs during these latter phases is given in Section 5.

4. Examples of Higher-Order Modeling

In Definition 3 (Section 2.4) we defined the meaning of HOAMs, giving three statements on how HOAMs can be used. This section is divided into sub-sections, where we exemplify these three kinds of usage by giving examples in MKL.

4.1 Parameterization of Models with Models

A common goal of model design is to make model libraries extensible and reusable. A natural requirement is to be able to parameterize models with other models, i.e., to reuse a model by replacing some of the submodels with other models. To illustrate the main idea of parameterized acausal models, consider the following oversimplified example of an automobile model, where we use `Connection()` with the same meaning as the previous `Wire()`:

```
def Automobile = model(Engine, Tire){
   def c1 = Connection();
   def c2 = Connection();
   Engine(c1);
   Gearbox(c1,c2);
   Tire(c2); Tire(c2); Tire(c2); Tire(c2)
};
```

In the example, the automobile is defined to have two formal parameters; an `Engine` model and a `Tire` model. To create a model instance of the automobile, the model can be applied to a specific engine, e.g., a model `EngineV6` and some type of tire, e.g. `TireTypeA`:

```
Automobile(EngineV6,TireTypeA);
```

If later on a new engine was developed, e.g., `EngineV8`, a new automobile model instance can be created by changing the arguments when the model instance is created, e.g.,

```
Automobile(EngineV8,TireTypeA);
```

Hence, new model instances can be created without the need to modify the definition of the `Automobile` model. This is analogous to a higher-order function which takes a function as a parameter.

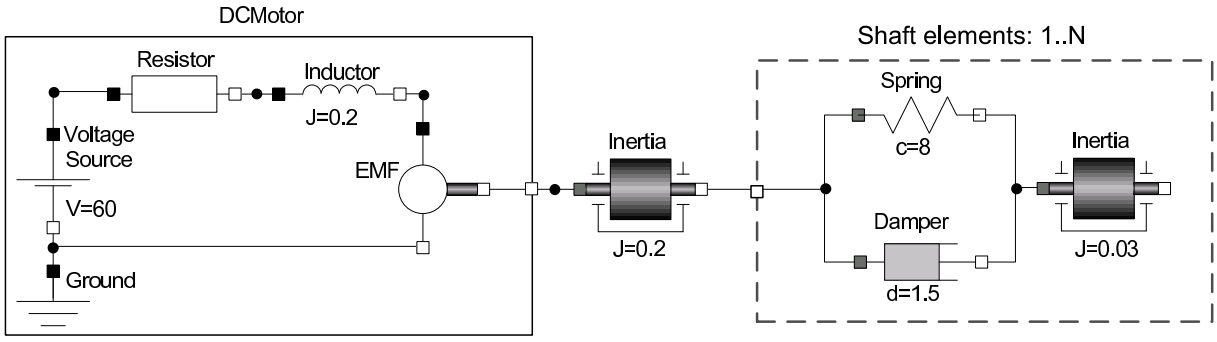


Figure 3. A mechatronic system with a direct current (DC) motor to the left and a flexible shaft to the right. The flexible shaft consists of 1 to N elements, where each element includes an inertia, a spring, and a damper.

In the example above, the definition of `Automobile` was not parametrized on the `Gearbox` model. Hence, the `Gearbox` definition must be given in the lexical scope of the `Automobile` definition. However, this model could of course also be defined as a parameter to `Automobile`.

This way of reusing acausal models has obvious strengths, and it is therefore not surprising that constructs with similar capabilities are available in some EOO languages, e.g., the special `redeclare` construct in Modelica. However, instead of creating a special language construct for this kind of reuse, we believe that HOAMs can give simpler and a more uniform semantics of a EOO language.

4.2 Recursively Defined Models

In many applications it is enough to hierarchically compose models by explicitly defining model instances within each other (e.g., the simple `Circuit` example). However, sometimes several hundreds of model instances of the same model should be connected to each other. This can of course be achieved manually by creating hundreds of explicit instances. However, this results in very large models that are hard to maintain and get an overview of.

One solution could be to add a loop-construct to the EOO language. This is the approach taken in Modelica, with the `for`-equation construct. However, such an extra language construct is actually not needed to model this behavior. Analogously to defining recursive functions, we can define *recursive models*. This gives the same modeling possibilities as adding the `for`-construct. However, it is more declarative and we have also found it easier to define a compact formal semantics of the language using this construct.

Consider Figure 3 which shows a Mechatronic model, i.e., a model containing components from both the electrical and mechanical domain. The left hand side of the model shows a simple direct current (DC) motor. The electromotoric force (EMF) component converts electrical energy to mechanical rotational energy. If we recall from Section 2, the connection between electrical components was defined using the `Wire` definition. However, in the rotational mechanical domain, the connection is instead defined by using the angle for the potential variable and the torque for flow. The rotational connection is defined as follows:

```
def RotCon = func(){(var(),flow())};
```

In the middle of the model in Figure 3 a rotational body with Inertia *J=0.2* is defined. This body is connected to a flexible shaft, i.e., a shaft which is divided into a number of small bodies connected in series with a spring and a damper in parallel in between each pair of bodies. N is the number of shaft elements that the shaft consists of.

A model of the mechatronic system is described by the following MKL source code.

```
def MechSys = model(){
    def c1 = RotCon();
    def c2 = RotCon();
    DCMotor(c1);
    Inertia(c1,c2,0.2);
    FlexibleShaft(c2,RotCon(),120);
};
```

The most interesting part is the definition of the component `FlexibleShaft`. This shaft is connected to the Inertia to the left. To the right, an empty rotational connection is created using the construction `RotCon()`, resulting in the right side not being connected. The third argument states that the shaft should consist of 120 elements.

Can these 120 elements be described without the need of code duplication? Yes, by the simple but powerful mechanism of recursively defined models. Consider the following self-explanatory definitions of `ShaftElement`:

```
def ShaftElement = model(ca,cb){
    def c1 = RotCon();
    Spring(ca,c1,8);
    Damper(ca,c1,1.5);
    Inertia(c1,cb,0.03);
};
```

This model represents just one of the 120 elements connected in series in the flexible shaft. The actual flexible shaft model is recursively defined and makes use of the `ShaftElement` model:

```
defrec FlexibleShaft = model(ca,cb,n){
    if(n==1)
        ShaftElement(ca,cb)
    else{
        def c1 = RotCon();
        ShaftElement(ca,c1);
        FlexibleShaft(c1,cb,n-1);
    };
};
```

The recursive definition is analogous to a standard recursively defined function, where the `if`-expression evaluates to false, as long as the count parameter `n` is not equal to 1. For each recursive step, a new connection is created by defining `c1`, which connects the shaft elements in series. Note that the last element of the shaft is connected to the second port of the `FlexibleShaft` model, since the shaft element created when the `if`-expression is evaluated to true takes parameter `cb` as an argument.

When the `MechSys` model is elaborated using our MKL prototype implementation, it results in a DAE consisting of 3159 equations and the same number of unknowns. It is obviously beneficial to be able to define recursive models in cases such as the one above, instead of manually creating 120 instances of a shaft element.

However, it is still a bit annoying to be forced to write the recursive model definition each time one wants to serialize a number of model instances. Is it possible to capture and define this serialization behavior once and for all, and then reuse this functionality?

4.3 Higher-Order Functions for Generic Model Transformation

In the previous section we have seen how models can be reused by applying models to other models, or to recursively define models. In this section we show that it is indeed possible to define several kinds of *model transformations* by using higher-order functions. These functions can in turn be part of a modeling language's standard library, enabling reuse of model transformation functions.

Recall the example from Section 2.2 of higher-order functions returning other anonymously defined functions. Assume that we want to create a generic function, which can take any two models that have two ports defined (`Resistor`, `Capacitor`, `ShaftElement` etc), and then compose them together by connecting them in parallel, and then return this new model:

```
def composeparallel = func(M1,M2){
    model(p,n){
        M1(p,n);
        M2(p,n);
    }
};
```

However, our model `Resistor` etc. does not take two arguments, but 3, where the last one is the value for the particular component (resistance for the `Resistor`, inductance for the `Inductor` etc.). Hence, it is convenient to define a function that sets the value of this kind of model and returns a more *specialized* model[4]:

```
def set = func(M,val){
    model(p,n){
        M(p,n,val);
    }
};
```

[4] In these examples we are using tuples as argument to the function, which makes it necessary to introduce a set function. The same kind of specialization can of course also be performed using *currying*. However, we have chosen to use the tuple notation, since it is likely to be more accessible for the reader with little experience of functional languages.

For example, a new model `Foo` that composes two other models can be defined as follows:

```
def Foo = composeparallel(set(Resistor, 100),
                          set(Inductor, 0.1));
```

A standard library can then further be enhanced with other generic functions, e.g., a function that composes two models in series:

```
def composeserial = func(M1,M2,Con){
    model(p,n){
        def w = Con();
        M1(p,w);
        M2(w,n);
    }
};
```

However, this time the function takes a third argument, namely a connector, which is used to create the connection between the models created in series. Since different domains have different kinds of connections (Wires, RotCon etc.), this must be supplied as an argument to the function. These connections are defined as higher-order functions and can therefore easily be passed as a value to the `composeserial` function.

We have now created two simple generic functions which compose models in parallel and in series. However, can we create a generic function that takes a model M, a connector C, and an integer n, and then returns a new model where n number of models M has been connected in series, using connector C? If this is possible, we do not have to create a special recursive model for the `FlexibleShaft`, as shown in the previous section.

Fortunately, this is indeed possible by combining a generic recursive model and a higher-order function. First, we define a recursive model `recmodel`:

```
defrec recmodel = model(M,C,ca,cb,n){
    if(n==1)
        M(ca,cb)
    else{
        def c1 = C();
        M(ca,c1);
        recmodel(M,C,c1,cb,n-1);
    };
};
```

Note the similarities to the recursively defined model `FlexibleShaft`. However, in this version an arbitrary model `M` is composed in series, using connector parameter `C`.

To make this model useful, we encapsulate it in a higher-order function, which takes a model `M`, a connector `C`, and an integer number `n` of the number of wanted models in series as input:

```
def serialize = func(M,C,n){
    model(ca,cb){
        recmodel(M,C,ca,cb,n);
    }
};
```

Now, we can once again define the mechatronic system given in Figure 3, but this time by using the generic function `serialize`:

```
def MekSys2 = model(){
    def c1 = RotCon();
    def c2 = RotCon();
    DCMotor(c1);
    Inertia(c1,c2,0.2);
    def FlexibleShaft =
        serialize(ShaftElement,RotCon,120);
    FlexibleShaft(c2,RotCon());
};
```

Even if the serialize function might seem a bit complicated to define, the good news is that such functions usually are created by library developers and not end-users. Fortunately, the end-user only has to call the serialize function and then use the newly created model. For example, to create a new model, where 50 resistors are composed in series is as easy as the following:

```
def Res50 =
   serialize(set(Resistor,100), Wire, 50);
```

5. Future Perspectives of Higher-Order Modeling

Our current design of higher-order acausal modeling capabilities as presented here is restricted to executing during the compiler (or interpreter) model elaboration phase, i.e., it cannot interact with run-time objects during simulation. However, removing this restriction gives interesting possibilities for run-time higher-order acausal modeling:

- The run-time results of simulation can be used in conjunction with models as first-class objects in the language, i.e., run-time creation of models, composition of models, and returning models. This is also useful in applications such as model-based optimization or model-based control, influenced by results from (on-line) simulation of models, e.g., [9].
- Structural variability [8, 18, 19, 29] of models and systems of equations means that the model structure can change at run-time, e.g., change in causality and/or number of equations. Run-time support for higher-order acausal model can be seen as a general approach to structurally variable systems. These ideas are discussed in [18, 19] in the context of Functional Hybrid Modeling (FHM).

These run-time modeling facilities provide more flexibility and expressive power but also give rise to several research challenges that need to be addressed:

- How can static strong type checking be preserved?
- How can high performance from compile-time optimizations be preserved? One example is index reduction, which requires symbolic manipulation of equations.
- How can we define a formal sound semantics for such a language?

Another future generalization of higher-order acausal modeling would be to allow models to be propagated along connections. For example, a water source could be connected to a generic flow connection structure with unspecified media. The selection of a media of type water in the source would automatically propagate to other objects.

6. Related Work

The main emphasis of this work is to explore the language concept of HOAMs in the context of EOO languages. In the following we briefly discuss three aspects of work which is related to this topic.

6.1 Functional Hybrid Modeling

As mentioned in the introduction, our notation of HOAMs has similarities to *first-class relations on signals*, as outlined in the context of Functional Hybrid Modeling (FHM) [18, 19]. The concepts in FHM are a generalization of Functional Reactive Programming (FRP) [28], which is based on reactive programming with causal hybrid modeling capabilities. Both FHM and FRP are based on *signals* that conceptually are functions over time. While FRP supports causal modeling, the aim of FHM is to support acausal modeling with structurally dynamic systems. However, the work of FHM is currently at an early stage and no published formal semantics or implementation currently exist.

HOAMs are similar to FHM's relations on signals in the sense that they are both first-class and that they can recursively reference themselves. In this paper we have showed how recursion can be used to define large structures of connected models, while in [19] ideas are outlined how it can be used for structurally dynamic systems.

One difference is that FHM's relations on signals are as its name states only relations on signals, while MKL acausal models can be parameterized on any type, e.g., other HOAMs or constants. By contrast, FHM's relation on signals can be parameterized by other relations or constants using ordinary functional abstraction, i.e., free variables inside a relation can be bound by a surrounding function abstraction. There are obvious syntactic differences, but the more specific semantic differences are currently hard to compare, since there are no public semantic specification available for any FHM language.

The work with MKL has currently focused on formalizing a kernel language for the elaboration process of typical EOO languages, such as Modelica. Hence, the formal semantics of MKL defined in [4] investigates the complications when HOAMs are combined with flow variables, generating sum-to-zero equations. How this kind of issue is handled in FHM is currently not published.

6.2 Metaprogramming and Metamodeling

The notion of higher-order models is related to, but different from metamodeling and metaprogramming. A metaprogram is a program that takes other programs/models as data and produces programs/models as data, i.e., meta-programs can manipulate *object programs* [21]. A metamodel may also have a subset of this functionality, i.e., it may specify the structure of other models represented as data, but not necessarily be executable and produce other models.

Staged metaprogramming can be achieved by quoting/unquoting operations applied in two or more stages, e.g., as in MetaML [25] and Template Haskell [22].

We have earlier developed a simple metaprogramming facility for Modelica by introducing quoting/unquoting mechanisms [2], but with limited ability to perform operations on code. A later extension [12] introduced general metaprogramming operations based on pattern-matching and transformations of abstract-syntax tree representations of models/programs similar to those found in many functional programming languages.

By contrast, the notion of higher-order models in this paper allows direct access to models in the language, e.g., passing models to models and functions, returning models, etc, without first representing (or viewing, reifying) models as data. This allows more integrated access to such facilities within the language including integration with the type system. Moreover, it often implies simpler usage and increased re-use compared to what is typically offered by metaprogramming approaches.

Metaprogramming, on the other hand, offers the possibility of greater generality on the allowed operations on models, e.g., symbolic differentiation of model equations, and the possibility of compile-time only approaches without any run-time penalty.

We should also mention the common usage of interpretive scripting languages, e.g., Python, or add-on interpretive scripting facilities using algorithmic parts of the modeling language itself such as in OpenModelica [12] and Dymola [7]. This works in practice, but is less well integrated and typically a bit ad hoc. This either requires two languages (e.g., Python and Modelica), or a separate interpretive implementation of a subset of the same language (e.g., Modelica scripting) which often give some differences in semantics, ad hoc restrictions, and inconsistent or partially missing integration with a general type system.

6.3 Modelica Redeclare and For-Equations

Modelica [17] provides a powerful facility called redeclaration, which has some capabilities of higher order models. Using redeclare, models can be passed as arguments to models (but not to functions using ordinary argument passing mechanisms e.g., at run-time), and returned from models in the context of defining a new model. For example:

```
model RefinedResistorCircuit =
  GenericResistorCircuit
  (redeclare model ResistorModel =
   TempResistor);
```

Redeclaration can also be used to adapt a model when it is inherited:

```
extends GenericResistorCircuit
  (redeclare model ResistorModel =
   TempResistor)
```

Redeclare is a compile-time facility which operates during the model elaboration phase. Moreover, using redeclare it is not possible to pass a model to a function, or to return a model from a function. Redeclaration is similar to C++ templates and Java Generics in that it allows passing types/models, but is more closely integrated in the language since it part of the class/model concept rather than being a completely separate feature. The Modelica redeclare can be seen as a special case of the more general concept of higher-order acausal models.

Modelica also provides the concept of for-equations to express repetitive equations and connection structures. Since iteration can be expressed as recursion, also for models as shown in Section 4.2, the concept of for-equations can be expressed as a special case of the more general concept of recursive models included in higher-order acausal models.

Even though EOO languages, such as Modelica, does not support HOAMs at the syntax level, HOAMs can still be very useful as a semantic concept for describing a precise formal semantics of the language. Language constructs, such as `for`-equations, can then be transformed down to a smaller kernel language. Having a small precisely defined language semantics can then make the language specification less ambiguous, enable better formal model checking possibilities, as well as providing more accurate model exchange.

7. Conclusions

We have in this paper informally presented how the concept of higher-order functions can be combined with acausal models. This concept, which we call higher-order acausal models (HOAMs), has been shown to be a fairly simple and yet powerful construct, which enables both parameterized models and recursively defined models. Moreover, by combining it with functions, we have briefly shown how it can be used to create reusable model transformation functions, which typically can be part of a model language's standard library. The examples and the implementation were given in a small research language called Modeling Kernel Language (MKL), and it was illustrated how HOAMs can be used during the elaboration phase. However, the concept is not limited to the elaboration phase, and we believe that future research in the area of HOAMs at runtime can enable both more declarative expressiveness as well as simplified semantics of EOO languages.

Acknowledgments

We would like to thank Jeremy Siek and the anonymous reviewers for many useful comments on this paper. This research work was funded by CUGS (the National Graduate School in Computer Science, Sweden) and by Vinnova under the NETPROG Safe and Secure Modeling and Simulation on the GRID project.

References

[1] Mats Andersson. *Object-Oriented Modeling and Simulation of Hybrid Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, December 1994.

[2] Peter Aronsson, Peter Fritzson, Levon Saldamli, Peter Bunus, and Kaj Nyström. Meta Programming and Function Overloading in OpenModelica. In *Proceedings of the 3rd International Modelica Conference*, pages 431–440, Linköping, Sweden, 2003.

[3] Paul Inigo Barton. *The Modelling and Simulation of Combined Discrete/Continuous Processes*. PhD thesis, Department of Chemical Engineering, Imperial Collage of Science, Technology and Medicine, London, UK, 1992.

[4] David Broman. Flow Lambda Calculus for Declarative Physical Connection Semantics. Technical Reports in Computer and Information Science No. 1, LIU Electronic Press, 2007.

[5] Ernst Christen and Kenneth Bakalar. VHDL-AMS - A Hardware Description Language for Analog and Mixed-Signal Applications. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 46(10):1263–1272, 1999.

[6] Ole-Johan Dahl and Kristen Nygaard. SIMULA: an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[7] Dynasim. Dymola - Dynamic Modeling Laboratory (Dynasim AB). `http://www.dynasim.se/` [Last accessed: April 30, 2008].

[8] Christoph Nytsch-Geusen et. al. MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics. In *Proceedings of the 4th International Modelica Conference*, Hamburg, Germany, 2005.

[9] Rüdiger Franke, Manfred Rode, and Klaus Krüger. On-line Optimization of Drum Boiler Startup. In *Proceedings of the 3rd International Modelica Conference*, pages 287–296, Linköping, Sweden, 2003.

[10] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, New York, USA, 2004.

[11] Peter Fritzson, Peter Aronsson, Adrian Pop, Håkan Lundvall, Kaj Nyström, Levon Saldamli, David Broman, and Anders Sandholm. OpenModelica - A Free Open-Source Environment for System Modeling, Simulation, and Teaching. In *IEEE International Symposium on Computer-Aided Control Systems Design*, Munich, Germany, 2006.

[12] Peter Fritzson, Adrian Pop, and Peter Aronsson. Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica. In *Proceedings of the 4th International Modelica Conference*, pages 519–525, 2005.

[13] Georgina Fábián. *A Language and Simulator for Hybrid Systems*. PhD thesis, Institute for Programming research and Algorithmics, Technische Universiteit Eindhoven, Netherlands, Netherlands, 1999.

[14] Johan Åkesson. *Languages and Tools for Optimization of Large-Scale Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, November 2007.

[15] MathWorks. The Mathworks - Simulink - Simulation and Model-Based Design. `http://www.mathworks.com/products/simulink/` [Last accessed: November 8, 2007].

[16] Robin Milner, Mads Tofte, Robert Harper, and David MacQuee. *The Definition of Standard ML - Revised*. The MIT Press, 1997.

[17] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.0*, 2007. Available from: `http://www.modelica.org`.

[18] Henrik Nilsson, John Peterson, and Paul Hudak. Functional Hybrid Modeling. In *Practical Aspects of Declarative Languages : 5th International Symposium, PADL 2003*, volume 2562 of *LNCS*, pages 376–390, New Orleans, Lousiana, USA, January 2003. Springer-Verlag.

[19] Henrik Nilsson, John Peterson, and Paul Hudak. Functional Hybrid Modeling from an Object-Oriented Perspective. In *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools*, pages 71–87, Berlin, Germany, 2007. Linköping University Electronic Press.

[20] M. Oh and Costas C. Pantelides. A modelling and Simulation Language for Combined Lumped and Distributed Parameter Systems. *Computers and Chemical Engineering*, 20(6–7):611–633, 1996.

[21] Tim Sheard. Accomplishments and research challenges in meta-programming. In *Proceedings of the Workshop on Semantics, Applications, and Implementation of Program Generation*, volume 2196 of *LNCS*, pages 2–44. Springer-Verlag, 2001.

[22] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, USA, 2002. ACM Press.

[23] Simon Peyton Jones. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.

[24] Bjarne Stroustrup. A history of C++ 1979–1991. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 271–297, New York, USA, 1993. ACM Press.

[25] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.

[26] The OpenModelica Project. `www.openmodelica.org` [Last accessed: May 8, 2008].

[27] D.A. van Beek, K.L. Man, MA. Reniers, J.e. Rooda, and R.R.H Schiffelers. Syntax and consistent equation semantics of hybrid Chi. *The Journal of Logic and Algebraic Programming*, 68:129–210, 2006.

[28] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 242–252, New York, USA, 2000. ACM Press.

[29] Dirk Zimmer. Enhancing Modelica towards variable structure systems. In *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools*, pages 61–70, Berlin, Germany, 2007. Linköping University Electronic Press.

Type-Based Structural Analysis for Modular Systems of Equations

Henrik Nilsson

School of Computer Science, University of Nottingham, UK
nhn@cs.nott.ac.uk

Abstract

This paper investigates a novel approach to a type system for modular systems of equations; i.e., equation systems constructed by composition of individual equation system fragments. The purpose of the type system is to ensure, to the extent possible, that the composed system is solvable. The central idea is to attribute a *structural type* to equation system fragments that reflects which variables occur in which equations. In many instances, this allows over- and underdetermined system fragments to be identified separately, without first having to assemble all fragments into a complete system of equations. The setting of the paper is equation-based, non-causal modelling, specifically Functional Hybrid Modelling (FHM). However, the central ideas are not tied to FHM, but should be applicable to equation-based modelling languages in general, like Modelica, as well as to applications featuring modular systems of equations outside the field of modelling and simulation.

Keywords Equation-based, non-causal modelling; Modelica; Functional Hybrid Modelling; structural analysis; types; type-based analysis; dependent types

1. Introduction

An important question in the context of equation-based modelling is whether or not the system of equations describing the modelled entity is solvable. In general, this can only be answered by studying the complete system of equations, and often not even then, except by attempting to solve the equations through simulation.

This is problematic. Models are usually modular, i.e. described by combining small systems of equations into larger ones. Being able to detect problems with individual parts or their combinations without first having to put together a complete system model is generally desirable. Moreover, a system may be *structurally dynamic*, meaning that the system of equations describing its behaviour *changes* over time. This implies that the question of the solvability cannot be addressed prior to simulation.

2nd International Workshop on Equation-Based Object-Oriented Languages and Tools. July 8, 2008, Paphos, Cyprus.

However, establishing that a system of equations *definitely is not solvable* can be almost as helpful. Fortunately there are criteria necessary (but not sufficient) for solvability that can be checked more easily and that are applicable to model fragments. A simple example is that the number of variables (unknowns) and equations must agree. For example, Modelica as of version 3.0 [12] enforces this constraint for model fragments (and thus for a model as a whole) so as to enable early detection of common modelling mistakes. Keeping track of the variable and equation balance is also the idea behind the structural constraint delta type system [2] with similar aims.

This paper is a preliminary investigation into an improved type-based (and thus compile-time) analysis for determining when (fragments of) systems of equations *cannot* be solved. The goal is to provide improved precision compared with just counting variables and equations by attributing a *structural type* to systems of equations reflecting which variables occur in which equations. A type-based approach is adopted as that is a natural way of ensuring that model fragments can be checked in isolation. This is particularly important for structurally dynamic systems where parts of the system change over time. However, as long as the *types* of the parts remain unchanged, and are reasonably informative, a meaningful analysis can still be carried out statically, at compile-time.

The development is carried out in the context of Functional Hybrid Modelling (FHM) [14, 15], as this provides a small and manageable modelling language framework that helps keeping the focus on the essence of the problem. FHM itself is still in an early stage of development. However, the central ideas put forward in this paper are not tied to FHM, but should be applicable to equation-based modelling languages like Modelica in general, as well as to applications featuring modular systems of equations outside the field of modelling and simulation. In effect, FHM is mainly used as a convenient and concise notation for modular systems of equations.

The rest of the paper is organised as follows. Section 2 provides general background and discusses related work. Section 3 provides an overview of FHM in the interest of making this paper relatively self-contained. Section 4 then develops the idea of structural types for modular systems of equations. As an example, this is applied to a simple electrical circuit in Section 5. Finally, Section 6 discusses future work and Section 7 gives conclusions.

2. Background and Related Work

Object-oriented modelling languages like Modelica [12] allow models to be developed in a *modular* fashion: systems of equations describing individual components are composed into larger systems of equations describing aggregates of components, and ultimately into a complete model of the system under consideration. As with software in general, such modularity is key to addressing the complexity of large-scale development as it allows large problems to be broken down into smaller ones that can be addressed independently, enables reuse, etc.

Of course, it is possible that mistakes are made during the development of a model. If so, it is desirable to catch such mistakes early. In a modular setting, this means checking whether a component in isolation is inherently faulty, and whether two or more components are being composed appropriately. As a result, mistakes can be localised effectively, meaning it becomes a lot easier to find and correct them. In contrast, mistakes that only become evident once a system has been fully assembled are usually a lot harder to pinpoint as the symptom in itself often is not enough to suggest any particular part of the system as the root of the problem. Even more problematic is a situation where problems only reveal themselves in use, as this means the system is unreliable.

A good way to catch errors early is to employ the notion of *types*. An entity has some particular type if it satisfies the properties implied by that type. A *type system* then governs under which conditions typed entities may be combined, and determines what properties the combined entity satisfies, i.e. its type.

As a simple example, consider the type $Integer$. If an entity has type $Integer$, this means that this entity satisfies the property of being an integer. Moreover, a rule of the type system would establish that *any* two entities satisfying the property of being integers can be combined using arithmetic addition into a new entity that also is an integer. This example is trivial, but as we will see, it is possible to capture much more complex properties through suitably defined types.

An important aspect of a type system is that it works solely on the basis of the *types* of the combined entities, without referring to any specific entity *instances*. This makes it possible to establish various properties of a combined entity before knowing exactly what all its parts are. This in turn allows for all manner of useful parametrisations, systems with dynamically evolving structure, etc.

This paper is concerned with equation systems properties for establishing whether a system can be solved or not. One necessary but not sufficient condition for solvability is the variable and equation balance: globally, the number of variables to solve for and the number of equations must be equal. Languages like Modelica naturally enforce this. Since version 3.0 [12], Modelica has adopted the even stricter criterion that (in essence) variables and equations must be *locally* balanced, i.e. balanced on a per component basis. Thus, in a sense, the property of being balanced is implicitly part of the type of a component in Modelica 3.0, as all well-typed components are balanced. Naturally, if all components of a model are locally balanced, this implies that the model is globally balanced.

Of course, a locally imbalanced model might still be globally balanced. To allow such models (without deferring *all* checking until a model has been fully assembled), it is necessary to *explicitly* make the variable and equation imbalance part of the type of a component. This was suggested by Nilsson *et al.* [14] and, independently, by Broman *et. al.* [2], who developed the idea in detail by integrating the notion of a "structural constraint delta" into the types of components.

Unfortunately, ensuring that the number of variables and equations agree only gives relatively weak assurances. As a simple example, consider the following system of equations, where f, g, and h are known functions, and x, y, and z are variables:

$$\begin{aligned} f(x, y, z) &= 0 \\ g(z) &= 0 \\ h(z) &= 0 \end{aligned}$$

The number of equations and variables agree. Yet it is clear that we cannot hope to solve this system of equations: x and y occur only in one equation, but we need two equations to have a chance to determine both of them. Moreover, z occurs alone in two of the equations, meaning that it may be impossible to find a value of z that satisfies them both. What we have in this case is an *underdetermined* system of equations for x and y (one equation, two variables), and an *overdetermined* system of equations for z (two equations, one variable).

Note that it was possible to establish the unsolvability of this system by just considering its *structure*: which variables occurs in which equations. This can be formalised through the notion of a *structurally singular* system of equations:

DEFINITION 1 (Structurally singular system of equations). *A system of equations is* structurally singular *iff it is not possible to put the variables and equations in a one-to-one correspondence such that each variable occurs in the equation it is related to.*

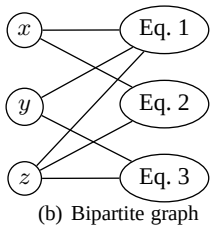
We now simply observe that a system of equations that is structurally singular is unsolvable.

Languages like Modelica ensure that models are not structurally singular as simulation is not possible if this is the case. However, in Modelica, this check is not carried out on a per component basis, but only once the system has been fully expanded into a "flat" system of equations. To the best of this author's knowledge, this is also the case for all similar languages. As a result, if it turns out that the final model is structurally singular, it can be very difficult to find out what the origin of the problem is.

To help overcome this difficulty, Bunus and Fritzson proposed a method to help localising the cause of any structural singularity [3, 4]. Their idea is to view the system of equations as a bipartite graph where the variables constitute one set of nodes, the equations the other set of nodes, and

$$f(x, y, z) = 0 \quad (1)$$
$$g(x, z) = 0 \quad (2)$$
$$h(y, z) = 0 \quad (3)$$

(a) System of equations



(b) Bipartite graph

	x	y	z
Eq. 1	1	1	1
Eq. 2	1	0	1
Eq. 3	0	1	1

(c) Incidence matrix

Figure 1. A system of equations and its corresponding structural representations.

there is an edge between a variable and an equation if the former occurs in the latter. See Figure 1(a) and 1(b). They then use the Dulmage and Mendelsohn canonical decomposition algorithm [6] to partition the flat system of equations into three parts: one overdetermined, one underdetermined, and one where the variables and equations match up. This information is then used to help diagnose the problem and suggest remedies.

Still, it would be an advantage if mistakes that *inevitably* are going to lead to structural singularities can be flagged up early, without first having to fully expand a model. This is true in particular for structurally dynamic systems: since the system of equations describing the behaviour of the system change over time, there is no one fully expanded system in this case. This is the kind of systems we ultimately hope to address in the context of our work on Functional Hybrid Modelling [14, 15].

This paper investigates an approach to early detection of structural singularities. The basic idea is to attribute types to components such that these types characterise the *structure* of the underlying system of equations used to represent a component, or more precicely, the structure of the equations that constitute its *interface*. We refer to this as the *structural type* of the component. The fundamental idea is similar to the structural constraint delta approach suggested by Broman *et al.*. However, the structural type is much richer: instead of a single number reflecting the variable and equation imbalance, the structural type details which variables occur in which equations. That is, the structural type is essentially a bipartite graph as in the work by Bunus and Fritzson, or it can be viewed as an *incidence matrix*: see Figure 1(c). We will freely switch between these two points of view in the following.

It turns out, though, that it often will be necessary to approximate the information on which variables occur in which equations. Thus the approach of this paper is not a complete alternative to error diagnosis on the final, flat system of equations as suggested by Bunus and Fritzson, but rather complementary to it.

3. Functional Hybrid Modelling

Functional Hybrid Modelling (FHM) [14, 15] is a generalisation of the central ideas of Functional Reactive Programming (FRP) [18]. In FRP, a functional programming language is extended with constructs for reactive programming and *causal*, hybrid, modelling, specifically *signals* (time-varying values) and functions on signals. This has proved to yield a very flexible and expressive framework for many different kinds of reactive and modelling applications [13, 9, 5, 8]. The FHM approach is similar, but *relations on signals* are added to address *non-causal* modelling.

The salient features of FRP and FHM relevant for this paper are covered in the rest of this section. The ideas are illustrated with a simple circuit example. This example is also used later in this paper. Note that FHM is currently being developed: no complete implementation exists yet. However, as explained earlier, it provides a convenient setting for this work.

3.1 Fundamental Concepts

FRP is a conceptual framework. A number of concrete implementations exists. Here, we will briefly consider Yampa [13], which is most closely related to FHM. Yampa is based on two central concepts: *signals* and *signal functions*. A signal is a function from time to a value; conceptually:

$$Signal\ \alpha \approx Time \rightarrow \alpha$$

(The conceptual nature of this definition is indicated by $\approx$. $\rightarrow$ is the infix type constructor for function types.) $Time$ is continuous, and is represented as a non-negative real number. The type parameter α specifies the type of values carried by the signal. For example, the type of a varying electrical voltage might be $Signal\ Voltage$.

A *signal function* is a function from $Signal$ to $Signal$:

$$SF\ \alpha\ \beta \approx Signal\ \alpha \rightarrow Signal\ \beta$$

When a value of type $SF\ \alpha\ \beta$ is applied to an input signal of type $Signal\ \alpha$, it produces an output signal of type $Signal\ \beta$. Signal functions are *first class entities* in Yampa. Signals, however, are not: they only exist indirectly through the notion of signal function. Additionally, signal functions satisfies a causality[1] requirement: at any point in time, the output must not depend on future input.

The output of a signal function at time t is uniquely determined by the input signal on the interval $[0, t]$. If a signal function is such that the output at time t only depends on the input at the very same time instant t, it is called *stateless*. Otherwise it is *stateful*.

3.2 First-Class Signal Relations

A natural mathematical description of a continuous signal function is that of an ODE in explicit form. A function is just a special case of the more general concept of a *relation*. While functions usually are given a causal interpretation, relations are inherently non-causal. Differential Algebraic Equations (DAEs), which are at the heart of

[1] This is *temporal* causality, a notion distinct from the notion of causality in "non-causal modelling."

non-causal modelling, express dependences among signals without imposing a causality on the signals in the relation. Thus it is natural to view the meaning of a DAE as a non-causal *signal relation*, just as the meaning of an ODE in explicit form can be seen as a causal signal function. Since signal functions and signal relations are closely connected, this view offers a clean way of integrating non-causal modelling into an Yampa-like setting.

Similarly to the signal function type SF of Yampa (Section 3.1), the type $SR\ \alpha$ stands for a relation on a signal of type α. Like signal functions, signal relations are first class entities, as will become clear in the following. Specific relations use a more refined type; e.g., for the derivative relation **der** we have the typing:

$$\mathbf{der} :: SR\ (Real, Real)$$

Since a signal carrying pairs is isomorphic to a pair of signals, we can understand **der** as a binary relation on two real-valued signals.

Signal relations are constructed as follows:

$$\mathbf{sigrel}\ pattern\ \mathbf{where}\ equations$$

The pattern introduces *signal variables* that at each point in time are bound to the *instantaneous* value of the corresponding signal. Given a pattern p of type t, $p :: t$, we have:

$$\mathbf{sigrel}\ p\ \mathbf{where} \ldots :: SR\ t$$

Consequently, the equations express relationships between instantaneous signal values. This resembles the standard notation for differential equations in mathematics. For example, consider $x' = f(y)$, which means that the instantaneous value of the derivative of (the signal) x at every time instant is equal to the value obtained by applying the function f to the instantaneous value of y.

There are two styles of basic equations:

$$e_1 = e_2$$
$$sr \diamond e_3$$

where e_i are expressions (possibly introducing new signal variables), and sr is an *expression* denoting a signal relation. We require equations to be well-typed. Given $e_i :: t_i$, this is the case iff $t_1 = t_2$ and $sr :: t_3$.

The first kind of equation requires the values of the two expressions to be equal at all points in time. For example:

$$f\ x = g\ y$$

where f and g are ordinary, pure, functions.[2]

The second kind allows an arbitrary relation to be used to enforce a relationship between signals. The symbol $\diamond$ can be thought of as *relation application*; the result is a constraint which must hold at all times. The first kind of equation is just a special case of the second in that it can be seen as the application of the identity relation. Thus, with I denoting the identity relation, an equation $e_1 = e_2$ could also be written $I \diamond (e_1, e_2)$.

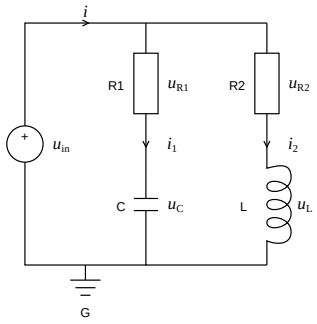


Figure 2. A simple electrical circuit.

For another example, consider a differential equation like $x' = f(x, y)$. Using the notation above, this equation can be written:

$$\mathbf{der} \diamond (x, f\ x\ y)$$

where **der** is the relation relating a signal to its derivative. For notational convenience, we will often use a notation closer to standard mathematical practice:

$$\mathbf{der}\ x = f\ x\ y$$

The meaning is exactly as in the first version. Thus, in the second form, **der** is *not* a pure function operating only on instantaneous signal values. It is a (stateful) signal function operating on the underlying signal.

We illustrate the ideas above by modelling the electrical circuit in Figure 2 (adapted from [11]). The type Pin is a record type describing an electrical connection. It has fields v for voltage and i for current.[3]

$$\begin{aligned}
&twoPin :: SR\ (Pin, Pin, Voltage)\\
&twoPin = \mathbf{sigrel}\ (p, n, u)\ \mathbf{where}\\
&\quad u = p.v - n.v\\
&\quad p.i + n.i = 0\\
&resistor :: Resistance \to SR\ (Pin, Pin)\\
&resistor\ r = \mathbf{sigrel}\ (p, n)\ \mathbf{where}\\
&\quad twoPin \diamond (p, n, u)\\
&\quad r * p.i = u\\
&inductor :: Inductance \to SR\ (Pin, Pin)\\
&inductor\ l = \mathbf{sigrel}\ (p, n)\ \mathbf{where}\\
&\quad twoPin \diamond (p, n, u)\\
&\quad l * \mathbf{der}\ p.i = u\\
&capacitor :: Capacitance \to (Pin, Pin)\\
&capacitor\ c = \mathbf{sigrel}\ (p, n)\ \mathbf{where}\\
&\quad twoPin \diamond (p, n, u)\\
&\quad c * \mathbf{der}\ u = p.i
\end{aligned}$$

The resistor, inductor and capacitor models are defined as extensions of the $twoPin$ model. This is accomplished

[2] We follow standard functional programming practice and denote ordinary function application simply by juxtapositioning, without any parentheses.

[3] The name Pin is perhaps a bit misleading since it just represents a pair of physical quantities, *not* a physical "pin component"; i.e., Pin is the type of *signal variables* rather than *signal relations*.

using functional abstraction rather than any Modelica-like class concept. Note how parameterized models are defined through functions *returning* relations, e.g. $resistor$. Since the parameters (like r of $resistor$) are normal function arguments, *not* signal variables, their values remain unchanged throughout the lifetime of the returned relations.[4] As signal relations are first class entities, signal relations can be parameterized on other signal relations in the same way.

To assemble these components into the full model, a Modelica-inspired **connect**-notation is used as a convenient abbreviation for connection equations. In FHM, this is just syntactic sugar that is expanded to basic equations: equality constraints for connected potential quantities and a sum-to-zero equation for connected flow quantities.[5] In the following, **connect** is only applied to Pin records, where the voltage field is declared as a potential quantity whereas the current field is declared as a flow quantity.

We assume that a voltage source model $vSourceAC$ and a ground model $ground$ are available in addition to the component models defined above. Moreover, we are only interested in the total current through the circuit, and, as there are no inputs, the model thus becomes a *unary* relation:

$$
\begin{array}{l}
simpleCircuit :: SR\ Current \\
simpleCircuit = \mathbf{sigrel}\ i\ \mathbf{where} \\
\quad resistor\ 1000 \quad \diamond\,(r1p, r1n) \\
\quad resistor\ 2200 \quad \diamond\,(r2p, r2n) \\
\quad capacitor\ 0.00047 \diamond\,(cp, cn) \\
\quad inductor\ 0.01 \quad \diamond\,(lp, ln) \\
\quad vSourceAC\ 12 \quad \diamond\,(acp, acn) \\
\quad ground \quad \diamond\,gp \\
\quad \mathbf{connect}\ acp\ r1p\ r2p \\
\quad \mathbf{connect}\ r1n\ cp \\
\quad \mathbf{connect}\ r2n\ lp \\
\quad \mathbf{connect}\ acn\ cn\ ln\ gp \\
\quad i = r1p.i + r2p.i
\end{array}
$$

There is no need to declare variables like $r1p$, $r1n$: their types are inferred. Note the signal relation expressions like $resistor\ 1000$ to the left of the signal relation application operators $\diamond$.

As an illustration of signal relation application, let us expand $resistor\ 1000 \diamond (r1p, r1n)$ using the definitions of $twoPin$ and $resistor$. The result is is the following three equations, where $u1$ is a fresh variable:

$$
\begin{array}{ll}
u1 & = r1p.n - r1n.v \\
r1p.i + r1n.i & = 0 \\
1000 * r1p.i & = u1
\end{array}
$$

3.3 Dynamic Structure

Yampa can express highly structurally dynamic systems. Ultimately, we hope to integrate as much of that functionality as possible into FHM. As a basic example, switching among two different sets of equations as a Boolean signal changes value might be expressed as follows:

$$
\begin{array}{l}
\mathbf{switch}\ b \\
\mathbf{when}\ False \\
\quad equations_1 \\
\mathbf{when}\ True \\
\quad equations_2
\end{array}
$$

If the type system approach outlined in this paper is to work for FHM, we need to consider how to handle such constructs from a type perspective. This is done in Section 4.4. There are many other outstanding problems related to implementation of structurally dynamic systems. But those are outside the scope if this paper.

4. Structural Types for Signal Relations

We now define the notion of structural type and show how it enables structural analysis to be carried out in a modular way, without having to first expand out signal relations to "flat" systems of equations. The key difficulty is abstraction of structural types, and consequently the section mostly focuses on that aspect.

4.1 The Structural Type

In essence, a signal relation is an *encapsulated* system of equations. When a signal function is applied, these equations impose constraints on signals in scope at the point of application through the variables of the signal relation *interface*. A larger system of equations is thus formed, composed from equations contributed by each applied signal relation.

Let us consider a simple example:

$$
\begin{array}{l}
foo :: SR\ (Real, Real, Real) \\
foo = \mathbf{sigrel}\ (x_1, x_2, x_3)\ \mathbf{where} \\
\quad f_1\ x_1\ x_2\ x_3 = 0 \\
\quad f_2\ x_2\ x_3 \quad\ = 0
\end{array}
$$

Let us assume a context with five variables, u, v, w, x, y, and let us apply foo twice in that context:

$$
\begin{array}{l}
foo \diamond (u, v, w) \\
foo \diamond (w, u + x, v + y)
\end{array}
$$

The result, obtained by substituting the variables u, v, w, x, y into the equations of foo, is the following system of equations:

$$
\begin{array}{ll}
f_1\ u\ v\ w & = 0 \\
f_2\ v\ w & = 0 \\
f_1\ w\ (u + x)\ (v + y) & = 0 \\
f_2\ (u + x)\ (v + y) & = 0
\end{array}
$$

Note that each application of foo contributed two equations to the composed system, each for a subset of the variables to the right of the relation application operator $\diamond$.

As discussed in Section 2, the aim is now to analyse the structure (which variables occur in which equations)

[4] In Modelica terms, they are **parameter**-variables.

[5] This simple treatment of **connect** has been sufficient for our small examples thus far. It is not clear if all aspects of the more comprehensive Modelica notion of **connect** could be handled in the same way.

of the composed system in order to identify situations that definitely will result in over- or underdetermined systems of equations.

However, for a variety of reasons, it is not desirable to assume that this can be done by simply unfolding the applied relations as was done above. In the context of FHM, what goes to the left of $\diamond$ is a signal relation *expression* that may involve parameters that are not known at compile time, thus preventing the expression from being evaluated statically. Or the exact contribution of the applied signal relation might not be known for other reasons, for example due to separate compilation or because it is structurally dynamic.

Thus, we are only going to assume that the *type* of the applied signal relation is known. To enable structural analysis, the type of signal relations is enriched by a component reflecting its structure. We refer to this as the *structural type* of the signal relation.

DEFINITION 2 (Structural type of system of equations). *The* structural type *of a system of equations is the incidence matrix of that system. It has one row for each equation, and one column for each variable in scope*[6]*. An occurrence of a variable in an equation is indicated by 1, a non-occurrence by 0.*

Note that definition 2 concerns systems of equations. For a *signal relation*, i.e. an *encapsulated* system of equations, the structural type is limited to the equations *contributed* by the signal relation and the variables of its interface. If the interface includes records of signal variables, like Pin of the simple circuit example in Section 3.2, then each field counts as an independent variable. We defer a precise definition until section 4.3.

As an example, consider the signal relation foo above. Its type, including the structural part, is:

$$foo :: SR\ (Real, Real, Real) \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

4.2 Composition of Structural Types

Now let us consider composition of structural types. The overall structural type for a sequence of equations is obtained by simply joining the incidence matrices for the individual equations as the same set of variables is in scope across all equations.

The structural type for a basic equation of the form

$$e_1 = e_2$$

is a single-row matrix indicating which variables occurs in expressions e_1 and e_2.

The structural type for the second form of equation, signal relation application, is more interesting. The general form of this kind of equation is:

$$sr \diamond (e_1, e_2, \ldots, e_i)$$

where e_1, e_2, ..., e_i are expressions over the signal variables that are in scope. These expressions and their relation to the variables in scope can also be represented by an incidence matrix, with one row for each expression and one column for each variable. The incidence matrix of the signal relation *application* is then obtained by Boolean matrix multiplication[7] of the structural type of the applied signal relation and the incidence matrix of the right-hand side expressions.

Returning to the example from the previous section, the incidence matrix of the right-hand side of the application

$$foo \diamond (u, v, w)$$

in a context with five signal variables u, v, w, x, y is

$$\begin{array}{c} u\ \ v\ \ w\ \ x\ \ y \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

(where the columns have been labelled for clarity). Multiplying the structural type of foo with this matrix yields:

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{array}{c} u\ \ v\ \ w\ \ x\ \ y \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array} = \begin{array}{c} u\ \ v\ \ w\ \ x\ \ y \\ \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Similarly, for

$$foo \diamond (w, u + x, v + y)$$

we obtain

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{array}{c} u\ \ v\ \ w\ \ x\ \ y \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix} \end{array} = \begin{array}{c} u\ \ v\ \ w\ \ x\ \ y \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

The complete incidence matrix for the two applications of foo is thus

$$\begin{array}{c} u\ \ v\ \ w\ \ x\ \ y \\ \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

Compare with the fully expanded system of equations in the previous section.

4.3 Abstraction over Structural Types

In the previous section, we saw how to obtain the overall structural type of a composition of signal relations given the structural types of the involved signal relations. The

[6] Only "unknown" signal variables are of interest here, not parameters or "known" (input) signal variables.

[7] Multiplication is understood as Boolean conjunction, $\wedge$ (logical "and"), and addition as Boolean disjunction, $\vee$ (logical "or").

next step is to consider how to encapsulate a system of equations in a signal relation. It is often the case that the set of variables in the interface of a signal relation, the *interface variables* is a proper subset of the variables that are in scope. A signal relation may thus abstract over a number of *local variables*. This, in turn, means that a number of the equations at hand *must* be used to solve for the local variables: the local variables are not going to be in scope outside the signal relation, and thus it is not possible to add further equations for them later.

The available equations are thus going to be partitioned into *local equations*, those that are used to solve for local variables, and *interface equations*, those that are contributed to the "outside" when the signal relation is applied. This immediately presents an opportunity to detect instances of over- and underdetermined systems of equations for the local variables on a per signal relation basis. However, it also presents a very hard problem as the partitioning is not uniquely determined, which in general implies that a signal relation does not have unique best structural type.

To illustrate, consider encapsulating the example from the previous section in a signal relation where only the variables u and y appear in the interface:

$$\begin{array}{l} bar = \mathbf{sigrel}\ (u, y)\ \mathbf{where} \\ \quad foo \diamond (u, v, w) \\ \quad foo \diamond (w, u + x, v + y) \end{array}$$

Recall the incidence matrix of the encapsulated system:

$$\begin{array}{c} \begin{array}{ccccc} u & v & w & x & y \end{array} \\ \left(\begin{array}{ccccc} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{array} \right) \end{array}$$

Three of the underlying equations are needed to solve for the local variables v, w, and x, the remaining one is the interface equation. But the only equation that *cannot* be chosen as the interface equation is number 2, as no interface variable occurs in this equation. Projecting out the columns for the interface variables for the the incidence matrices for the three possible choices of interface equation yields

$$\begin{array}{ccc} \begin{array}{cc} u & y \end{array} & \begin{array}{cc} u & y \end{array} & \begin{array}{cc} u & y \end{array} \\ \left(\begin{array}{cc} 1 & 0 \end{array} \right) & \left(\begin{array}{cc} 1 & 1 \end{array} \right) & \left(\begin{array}{cc} 1 & 1 \end{array} \right) \end{array}$$

The last two possibilities are equivalent, so this leaves us with two possible structural types: the signal relation bar can either provide a single equation in which the first variable of the interface occurs, or it can provide an equation in which both interface variables occurs, depending on the chosen equation partitioning in bar.

A modelling language compiler will decide on a specific partitioning. But this choice is typically dictated by intricate numerical considerations and often also by the usage context[8]. As it is essential that typechecking is compositional, it is clear that the partitioning must be done independently of usage context. And to ensure that the type system is independent of arbitrary implementation choices, as well as reasonably easy to understand for the end user, it is clear that the partitioning should not depend on low-level numerical considerations either.

There are two approaches for dealing with the situation. One is to *accept* that a signal relation can have more than one structural type. This paper does not explore that avenue as there is a risk that it would lead to a combinatorial explosion of possibilities to consider. Still, it should not be ruled out. The other approach is to decide on a suitable notion of "best" structural type. Then, if a signal relation has more than one possible structural type, choose the best one, if this is a uniquely determined choice, otherwise *approximate* all best types with a type that is better than them all, but still as informative as possible, and take this approximation as the structural type of the signal relation.

We are going to adopt the a notion of "best" that reflects the observation that an equation is more useful the more variables that occur in it (as this gives more flexibility when choosing which equation to use to solve for which variable). We are further going to assume that an implementation is free to make such a best choice. The latter might not be the case, but we should then keep in mind that the objective of the type system is *not* to guarantee that a system of equations *can* be solved, but to detect cases where a system of equations definitely *cannot* be solved. Assuming a freedom of choice is thus a safe approximation.

DEFINITION 3 (Subsumed (variables)). *Let V_1 and V_2 be sets of variables. V_1 is* subsumed by *V_2 iff $V_1 \setminus V_2 = \emptyset$.*

DEFINITION 4 (Subsumed (structural types)). *Let s_1 and s_2 be structural types. s_1 is* subsumed by *s_2 iff there exists a permutation of the rows of the incidence matrix for s_2 such that the variables of each row of the incidence matrix for s_1 are subsumed by the variables of the corresponding row of the permuted incidence matrix for s_2. The subsumed relation on structural types is denoted by the infix symbol $\leq$.*

DEFINITION 5 (Best Structural Types). *Let S be a set of structural types. The* best structural types *in S is the set*

$$\{s \mid s \in S \wedge \neg(\exists s' \in S \,.\, s \leq s')\}$$

Returning to the signal relation bar above, we find that it actually has a single best structural type since

$$\begin{array}{ccc} \begin{array}{cc} u & y \end{array} & & \begin{array}{cc} u & y \end{array} \\ \left(\begin{array}{cc} 1 & 0 \end{array} \right) & \leq & \left(\begin{array}{cc} 1 & 1 \end{array} \right) \end{array}$$

The complete type of bar is thus:

$$bar :: SR\ (Real, Real)\ \left(\begin{array}{cc} 1 & 1 \end{array} \right)$$

As an example of a case where there is not any best type, consider

$$s_1 = \left(\begin{array}{ccc} 1 & 0 & 1 \\ 1 & 0 & 0 \end{array} \right),\ s_2 = \left(\begin{array}{ccc} 0 & 0 & 1 \\ 1 & 1 & 1 \end{array} \right)$$

[8] Tools usually expand the model to a flat system of equations first. These equations are then "sorted", meaning deciding on which equation to use to solve for a particular variable [7].

Note that $s_1 \not\leq s_2$ and $s_2 \not\leq s_1$. Neither is better than the other, and the best structural types of $S = \{s_1, s_2\}$ is S.

What is needed if there is more than one best type is to find an approximation in the form of an upper bound that subsumes them all. Clearly such a bound exists: just take the incidence matrix with all 1s, for example. That corresponds to an assumption that each equation can be used to solve for any variable, meaning that we are back to the approach of counting equations and variables. However, to avoid loosing precision unnecessarily, a *smallest upper bound* should be chosen. As the following example shows, there may be more than one such bound, in which case one is chosen arbitrarily.

Consider the two structural types

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

Upper bounds can be constructed by taking the union of the first incidence matrix and all possible row permutations of the second one. As there are only two rows, we get two upper bounds:

$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

Neither is smaller than the other. However, they are both as small as possible, as removing a single 1 from any matrix means it will not subsume one or the other of the original matrices. Thus, in general, the least upper bound of structural types under the subsumed ordering is not uniquely determined.

We can now give a definition of the structural type of a signal relation:

DEFINITION 6 (Structural type of a signal relation). *The* structural type of a signal relation *with a body of m equations over n variables, of which i variables occur in the interface, if that type exists, is an $(m - (n - i)) \times i$ incidence matrix that is a least upper bound of the structural types of all possible choices of interface equations.*

The following algorithm determines the structural type of a signal relation when one exists, or reports an error otherwise. We claim this without proof, leaving that as future work:

Arguments:

1. Structural type s for the system of equations of the body of the signal relation in the form of an $m \times n$ incidence matrix (m equations, n variables).
2. The set V of variables, $|V| = n$, and a mapping from variables to the corresponding column number of the incidence matrix.
3. The set I of interface variables of the signal relation.

Result:

- If successful, an $(m - (n - |I|)) \times |I|$ incidence matrix representing the structural type of the signal relation.
- Otherwise, an indication of the problem(s): under- or overdetermined system of local equations; overdetermined system of interface equations.

Algorithm:

1. Let $L = V \setminus I$ be the set of local variables. Partition s into three parts:
   - s_L: rows corresponding to equations over variables in L only, the *a priori local equations*;
   - s_I: rows corresponding to equations over variables in I only the *a priori interface equations*;
   - s_M: the remaining rows, corresponding to equations over mixed interface and local variables.

   Let m_L, m_I, m_M be the number of rows of s_L, s_I, and s_M respectively. (Note that the a priori local equations can *only* be used to solve for local variables, whereas the a priori interface equations can *only* be used to solve for interface variables.)
2. Let $k = |L| - m_L$. k is the number of equations in addition to local ones that are needed to solve for all local variables.
   - If $k < 0$, report "overdetermined local system of equations".
   - If $k > m_M$, report "underdetermined local system of equations".
3. Initialise $S_{I'}$ to $\emptyset$
4. Choose k rows from s_M in all possible ways ($\binom{m_M}{k}$ possibilities, $m_M \geq k$). For each such choice:
   (a) Partition s_M into $s_{L'}$ containing the k chosen rows and $s_{I'}$ containing the remaining rows.
   (b) Consider s_L and $s_{L'}$ restricted to the local variables L as a bipartite graph and compute a maximum matching using the standard augmenting path algorithm [1, pp. 246–250]. Check if the size of the matching is equal to $|L|$. If yes, this means that each variable in L can be paired with a row from s_L or $s_{L'}$ in which it occurs, which is a necessary condition for using the equations corresponding to the rows from s_L or $s_{L'}$ to solve for the local variables.
   (c) Consider s_I and $s_{I'}$ restricted to the interface variables I as a bipartite graph and compute a maximum matching using the standard augmenting path algorithm. Check if the size of the matching is equal to the number of rows of s_I and $s_{I'}$, i.e. $m_I + m_M - k$. If yes, then this means that all equations corresponding to the rows of s_I and $s_{I'}$ can be used simultaneously to solve for one of the interface variables. This is a necessary condition for ensuring that the interface equations contributed by the signal relation does not constitute an overdetermined system.
   (d) If both checks above passed, then this particular choice of k rows is *valid*.
   (e) For each valid choice, add $s_{I'}$ restricted to the variables I to $S_{I'}$.
5. If $S_{I'} = \emptyset$, it is not possible to solve for the local variables and/or the interface equations contributed by the

signal relation are going to be overdetermined. Report the problem.

6. Determine the best structural types $S_{I''}$ of $S_{I'}$.
7. Let $s_{I''}$ be a least upper bound of $S_{I''}$.
8. The incidence matrix obtained by joining s_I and s''_I is the structural type of the signal relation, i.e. a least upper bound of the structural types of all possible choices of interface equations.

4.4 Structurally Dynamic Systems

To conclude the development, we briefly consider how to handle structurally dynamic systems, for example of the type illustrated in section 3.3. Clearly, the structural types of the equations in the different branches could be different. However, at any point in time, the choice of which equations that are active is determined by the condition of the **switch**-construct. Thus, the structural type of the entire **switch**-construct is the *greatest lower bound* of the structural types of the branches, as that is the only thing which is guaranteed at all points in time. One may also want to impose additional consistency constraints between the branches to avoid unpleasant surprises at run-time, e.g. due to the system of equations all of a sudden becoming overdetermined. But this has not yet been investigated.

4.5 Implementation

The algorithm for computing the structural type for a signal relation has been prototyped in Haskell. It implements all aspects of the described algorithm, except that it has not been verified whether the computation of upper bounds indeed yields one of the least upper bounds.

The time complexity of the algorithm is a concern. For example, the $\binom{m_M}{k}$ possible partitionings of the mixed equations that need to be investigated could, in adverse circumstances, be a large number. However, there may be ways to exploit more of the structure of the equations in order to limit the number of alternatives to consider. It is also easy to check how many partitioning there are before starting to enumerate them, and if they are judged to be too many, one can simply default to a safe over approximation of the type.

5. Structural Types for a Simple Electrical Circuit

As an example, let us apply the structural type system developed in section 4 to the simple electrical circuit from section 3.2.

Let us first consider the resistor. Recall that Pin is a record of two fields v and i, and that the signal relation interface thus consists of *four* variables: $p.v$, $p.i$, $n.v$, and $n.i$:

$$
\begin{aligned}
&resistor :: Resistance \to SR\ (Pin, Pin)\\
&resistor\ r = \mathbf{sigrel}\ (p, n)\ \mathbf{where}\\
&\quad twoPin \diamond (p, n, u)\\
&\quad r * p.i = u
\end{aligned}
$$

Before approximation, the two possible structural types for $resistor$ are

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

reflecting a choice between using $u = p.v - n.v$ or $r * p.i = u$ for solving for the local variable u. (The equation $u = p.v - n.v$ is contributed by $twoPin$. However, note that only its *structural type* is of interest here, not the exact equation.) This gets approximated with a least upper bound to:

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Of course, $resistor$ cannot provide a single equation in which all of $p.v$, $p.i$, and $n.v$ occur. But as the equation can only be used to solve for one of the variables, and as an equation can be provided for either two of the variables or the third, this is not too bad.

Let us now consider $inductor$:

$$
\begin{aligned}
&inductor :: Inductance \to SR\ (Pin, Pin)\\
&inductor\ l = \mathbf{sigrel}\ (p, n)\ \mathbf{where}\\
&\quad twoPin \diamond (p, n, u)\\
&\quad l * \mathbf{der}\ p.i = u
\end{aligned}
$$

The possible structural types before approximation are the same as for resistor, but this time reflecting a choice between using $u = p.v - n.v$ or $p.i = \int p.i' \,\mathrm{d}t$, where $p.i'$ is the state derivative, for solving for the local variables. Note that the equation $l * p.i' = u$ is local, as neither the state derivative nor u occurs in the interface of $inductor$. After approximation, the structural type of $inductor$ becomes the same as that of $resistor$.

The case for $capacitor$ is also very similar, and both the possible structural types prior to approximation and the final structural type are again the same.

For a final example, suppose a mistake has been made in the definition of $simpleCircuit$: instead of

$$
\begin{aligned}
&\mathbf{connect}\ r1n\ cp\\
&\mathbf{connect}\ r2n\ lp
\end{aligned}
$$

the equations read

$$
\begin{aligned}
&\mathbf{connect}\ r2n\ lp\\
&\mathbf{connect}\ r2n\ lp
\end{aligned}
$$

Note that the number of equations and variables remain exactly the same in the two cases (each **connect** above is expanded to one equality constraint and one sum-to-zero equation).

The structural type checking algorithm presented in this paper correctly reports that $simpleCircuit$ is a locally underdetermined system. If only variables and equations had been counted, this error would not have been detected.

6. Future Work

It should be emphasised that what has been presented in the present paper is only a preliminary investigation into the

basics of a type-based structural analysis for modular systems of equations. It is not yet yet a full-featured type system. In particular, we have only considered the structural aspect in isolation, and to that end it was tacitly assumed that the structural types of composed signal relations were known, enabling the overall structural type of signal relations to be computed in a bottom-up manner.

However, FHM aims at treating signal relations as first class entities. One consequence of this is that signal relations can be *parametrised*, including on other signal relations. In FHM, a parametrised signal relation is simply a function that computes a signal relation given values of the parameters, which could include other signal relations. The question then is how to determine the structural type of any signal relation parameters.

One option would be to insist that the structural types of signal relation parameters is always declared. This could be cumbersome, but there is always the possibility of making a permissible (imprecise) default assumption in the absence of explicit declarations. Another option might be to try to infer suitable structural constraints for the parameters from how they are being used in Hindley-Milner fashion. A third option would be to move to a framework of *dependent types* [17, 16] where types are indexed by (can depend on) *terms*. In our case, the incidence matrices that represent the structural type would be considered term-level data, and the output structural type of a parametrised signal relation is then allowed to depend on the input structural type(s), or even the values of other parameters, meaning that the output structural type will be given as a function of the parameter values.

Incidentally, Modelica effectively also provides parametrised signal relations through its mechanism of replaceable components. Here the problem is addressed by syntactically requiring a default value for the replaceable component, which is used for typechecking, and additionally insisting that any replacement conforms with the type of the default value in such a way that the the result after any replacement is still guaranteed to be well-typed.

Another aspect that was not considered is how to handle equations on arrays. If the sizes of the arrays are manifestly known, it would be possible to consider an array equation simply as a shorthand notation for equations between the individual elements. But that is not very attractive, and it would inevitably lead to unwieldy structural types, bloated with lots of repetitive information. And, of course, if the array sizes are not manifest but parameters of the relation, it would be even more problematic. The most feasible approach is likely to restrict array equations in such a way that each such equation can be considered a single equation for the purpose of the structural types. Again, moving to a setting of dependent types might be helpful, as the typechecking depends on term-level data, i.e. the sizes of the arrays. Dependent type systems supporting explicitly sized data has been studied extensively. One good example is Dependent ML [19, 20].

We would also like to integrate checking of physical dimensions [10] into the FHM type system. We observe that this is another reason to look closer at dependent types since the types become dependent on term-level data. For example, if an entity with a dimension type is subject to iterated multiplication, the resulting dimension depends on *how many times* the multiplication was iterated.

Finally, there are usability aspects that needs to be considered. While the type errors that are reported should be attributed fairly precisely to the component that is faulty, it is not clear how to phrase the error messages such that the problem becomes evident to the end user. Also, we need to keep in mind the conservative nature of the type system: there is no guarantee that further errors will not be discovered when a complete system of equations has been assembled. Combining the approach developed here with that of Bunus and Fritzson [3, 4] might help on both counts.

7. Conclusions

This paper presented a preliminary investigation into type system for modular systems of equations. The setting of the paper is equation-based, non-causal modelling, but the central ideas should have more general applicability. The paper showed how attributing a *structural type* to equation system fragments allows over- and underdetermined system fragments to be identified separately, without first having to assemble all fragments into a complete system of equations. The central difficulty was handling abstraction of systems of equations. The paper presented an algorithm for determining the best possible type for an abstracted system, although this may involve approximation.

It should be emphasised that was has been presented is not yet a complete type system. The paper only considers the structural aspect, and it was tacitly assumed that these structural types essentially could be determined in a straightforward bottom-up manner. The goal of treating signal relations as first class entities raises a number of further challenges, some of which were discussed in Section 6.

Acknowledgments

This work was supported by EPSRC grant EP/D064554/1. The author wishes to thank the anonymous referees for many useful suggestions that helped improve the paper.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Company, 1983.

[2] David Broman, Kaj Nyström, and Peter Fritzson. Determining over- and under-constrained systems of equations using structural constraint delta. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 151–160, Portland, Oregon, USA, 2006. ACM.

[3] Peter Bunus and Peter Fritzson. A debugging scheme for declarative equation based modeling languages. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002)*, volume

2257 of *Lecture Notes in Computer Science*, pages 280–298, Portland, OR, USA, January 2002. Springer-Verlag.

[4] Peter Bunus and Peter Fritzson. Methods for structural analysis and debugging of Modelica models. In *Proceedings of the 2nd International Modelica Conference*, pages 157–165, Oberpfaffenhofen, Germany, March 2002.

[5] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18, Uppsala, Sweden, August 2003. ACM Press.

[6] A. L. Dulmage and N. S. Mendelsohn. Coverings of bipartite graphs. *Canadian Journal of Mathematics*, 10:517–534, 1958.

[7] Hilding Elmqvist. *A Structured Model Language for Large Continuous Systems*. PhD thesis TFRT-1015, Department of Automatic Control, Lund Institute of Technology, 1978.

[8] George Giorgidze and Henrik Nilsson. Switched-on Yampa: Declarative programming of modular synthesizers. In Paul Hudak and David S. Warren, editors, *Practical Aspects of Declarative Languages (PADL) 2008*, volume 4902 of *Lecture Notes in Computer Science*, pages 282–298, San Francisco, CA, USA, January 2008. Springer-Verlag.

[9] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming, 4th International School 2002*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[10] Andrew Kennedy. Dimension types. In *Proceedings of the 5th European Symposium on Programming*, number 788 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[11] The Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling: Tutorial version 1.4*, December 2000.

[12] The Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling: Language Specification Version 3.0*, September 2007.

[13] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

[14] Henrik Nilsson, John Peterson, and Paul Hudak. Functional hybrid modeling. In *Proceedings of PADL'03: 5th International Workshop on Practical Aspects of Declarative Languages*, volume 2562 of *Lecture Notes in Computer Science*, pages 376–390, New Orleans, Lousiana, USA, January 2003. Springer-Verlag.

[15] Henrik Nilsson, John Peterson, and Paul Hudak. Functional hybrid modeling from an object-oriented perspective. In Peter Fritzson, François Cellier, and Christoph Nytsch-Geusen, editors, *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools*, number 24 in Linköping Electronic Conference Proceedings, pages 71–87. Linköping University Electronic Press, 2007.

[16] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[17] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley Publishing Company, 1991.

[18] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of PLDI'01: Symposium on Programming Language Design and Implementation*, pages 242–252, June 2000.

[19] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

[20] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.

Introducing Messages in Modelica for Facilitating Discrete-Event System Modeling

Victorino Sanz Alfonso Urquia Sebastian Dormido

Dpto. Informática y Automática, ETSII Informática, UNED
Juan del Rosal 16, 28040 Madrid, Spain
{vsanz,aurquia,sdormido}@dia.uned.es

Abstract

The work performed by the authors to provide to Modelica more discrete-event system modeling functionalities is presented. These functionalities include the replication of the modeling capacities found in the Arena environment, the SIMAN language and the DEVS formalism. The implementation of these new functionalities is included in three free Modelica libraries called ARENALib, SIMANLib and DEVSLib. These libraries also include capacities for random number and variates generation, and dynamic memory management. They are freely available for download at http://www.euclides.dia.uned.es/.

As observed in the work performed, discrete-event system modeling with Modelica using the process-oriented approach is difficult and complex. The convenience to include a new concept in the Modelica language has been observed and is discussed in this contribution. This new concept corresponds to the model communication mechanism using messages. Messages help to describe the communication between components in a discrete-event system. They do not substitute the current discrete-event modeling capabilities of Modelica, but extend them. The proposed messages mechanism in Modelica is discussed in the manuscript. An implementation of the messages mechanism is also proposed.

Keywords discrete events, process-oriented modeling, Modelica, Arena, SIMAN, DEVS, messages

1. Introduction

Several Modelica libraries have been developed by the authors in order to provide to Modelica more discrete-event system modeling capabilities. The work performed is specially based in modeling systems using the process-oriented approach, reproducing the modeling functionalities of the Arena simulation environment [10] in a Modelica library called ARENALib. The functionalities of the SIMAN modeling language [18], used to describe components in Arena, have also been reproduced in a Modelica library called SIMANLib. One objective of the development of this library is to take advantage of the Modelica object-oriented capabilities to modularize as much as possible the development of discrete-event system models. Also, the use of a formal specification to describe SIMANLib components helped to understand, develop and maintain them. SIMANLib blocks can be described using DEVS specification formalism [21]. Event communication in DEVS and block communication in SIMANLib match perfectly. An implementation of the Parallel DEVS formalism [23] has been developed in a Modelica library called DEVSLib, and used to describe the components in SIMANLib. All the performed work with Modelica has been developed using the Dymola modeling environment [1]. The problems encountered during the development of the ARENALib, SIMANLib and DEVSLib Modelica libraries, and the solutions applied to those problems are discussed.

The Modelica language includes several functionalities for discrete-event management, such as `if` expressions to define changes in the structure of the model, or `when` expressions to define event conditions and the actions associated with the defined events [16].

Other authors have contributed to the discrete-event system modeling with Modelica. Depending on the formalism used to define the discrete-event system, contributions can be found using finite state machines [7, 14, 17], Petri nets [15] or the DEVS formalism [2, 3, 4, 8]. On the other hand, other authors have developed tools to simulate discrete-event systems in conjunction with Modelica. For example, translating models developed using a subset of the Modelica language to the DEVS formalism. The translated models are then simulated using the CD++ DEVS simulator [5]. Also, other authors describe the discrete-event system with an external tool that translates a block diagram to Modelica code [19].

All these contributions use the event-scheduling approach for describing the discrete-event systems [12]. Events are scheduled to occur in a future time instant. The

2nd International Workshop on Equation-Based Object-Oriented Languages and Tools. July 8, 2008, Paphos, Cyprus.

simulation evolves executing the actions associated with the occurrence of the events.

Due to the difficulties and problems encountered during the development of the mentioned Modelica libraries, the convenience of introducing a new concept in Modelica has been identified. This new concept will facilitate the development of discrete-event systems, extending the current Modelica capacities. This new concept is the model communication using the messages mechanism. The main characteristics and functionalities of this mechanism are also discussed in this manuscript.

2. Process-Oriented Modeling in Modelica

A discrete-event system modeled using the process-oriented approach is described from the point of view of entities [10]. These entities flow through the components of the system, and some processes are applied to them using the available resources of the system. Some of the information associated with the entities are the serial number, the type, the statistical indicators, the attributes, the creation time, and the processing time among others. An example of this kind of system can be a beverage manufacturing system. The entities of this system are the bottles. A tank fills bottles with the beverage. Once filled, the bottles are labeled and quality controlled before they are accepted for distribution (first and second class bottles). Bottles without the required quality are cleaned and re-labeled. The components of this kind of systems are usually stochastic. For example, the labeling and cleaning processes are modeled using the Triangular probability distribution. The quality controls are represented by two-way decisions whose percentage is based on the values of uniform random variates.

The process-oriented approach is supported by the Arena simulation environment to model discrete-event systems. Arena has *data modules*, that represent the entities, the resources, and some other static elements of the system, and *flowchart modules*, that represent the processes performed on the entities across the system. The implementation of the beverage manufacturing system using Arena is shown in Figure 1a. It is modeled as a hybrid system, because the tank is represented by a continuous-time model.

Arena allows some simple hybrid modeling by describing level variables, that change continuously over time, and rate variables, that represent how fast the level variable changes its value. Each pair of level/rate variables represents a differential equation that is simulated using Euler, RKF or any user-implemented integration method.

2.1 ARENALib

ARENALib reproduces the Arena data and flowchart modules that have to be combined and connected to model the system. This library is freely available for download at [6]. At the moment, the Create, Process, Dispose and Decide flowchart modules and the Entity, Queue, Resource and Variable data modules, of the Arena Basic Process panel, have been implemented.

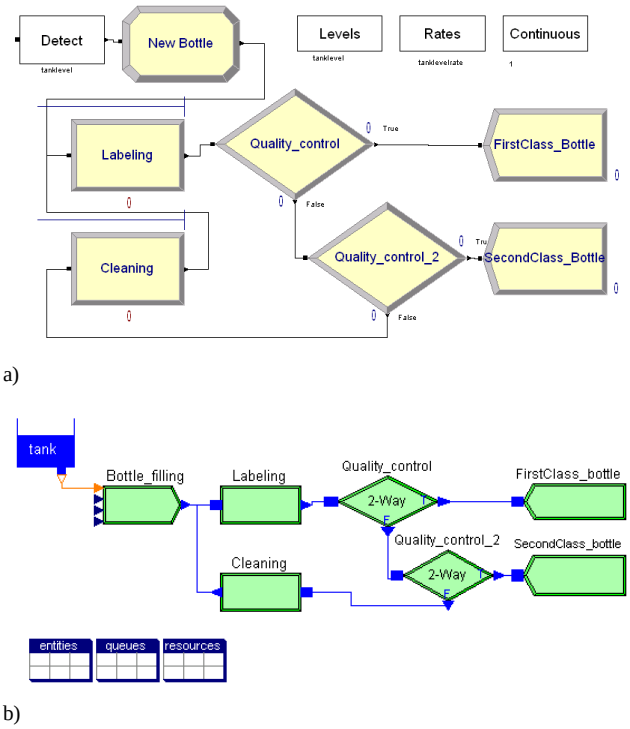


Figure 1. Beverage manufacturing system. An example of hybrid discrete-event system developed using: a) Arena; and b) ARENALib.

The library also allows hybrid system modeling, combining the current Modelica continuous-time system modeling functionalities with the components of ARENALib. A detailed description of the library can be found in [20]. The model of the beverage manufacturing system composed using ARENALib is shown in Figure 1b. In this figure, the *Bottle_filling* module corresponds to a Create module, *Labeling* and *Cleaning* correspond to Process modules, *Quality_control* and *Quality_control_2* are Decide modules and the *FirstClass_bottle* and *SecondClass_bottle* are Dispose modules. *Entities*, *queues* and *resources* contain the data modules required for this system.

The main tasks accomplished during the development of the ARENALib library were: a) the model communication mechanism; b) the entity management; c) the management of the statistical information and; d) the generation of stochastic data. These tasks and the solutions proposed and implemented to the problems encountered during the development of the ARENALib library are discussed below.

2.2 Model Communication Mechanism

Entities are generated in the system during the simulation, flow across the components of the system and, if necessary, are disposed. Generally, the number of entities in the system changes during the simulation run, depending on the behavior of the system.

Usually an entity arrives to a module, is processed and sent to the following module. Entity communication is an important part of the simulation process.

Model interaction in Modelica can be performed using connectors. A connector is an special class and contains some variables that are linked with the ones in another connector using a connect equation. The connect equation relates variables either equaling them, or summing them and equaling the sum to zero.

Several approaches have been studied, implemented and evaluated during the development of ARENALib in order to perform the entity transmission between modules. The approach used to perform the entity transmission is completely transparent for the end user. At the user level, the communication is just defined by connecting the output ports of some modules to the input ports of other modules. The mentioned approaches are discussed next.

2.2.1 Direct transmission

It consists of specifying all the variables that define a type of entity inside the connector. The values assigned to the variables of one connector represent an entity. These values are assigned, because of the connect equation, to the connector of the next model. In this way, an entity is directly transmitted from one model to another. Different types of entities require different connectors, one for each type. This is the simplest way for communicating models, but presents a problem: the simultaneous reception of several entities at one model. There are three possible situations for this problem:

- One-to-one connection: one model sends several entities to another model at the same time.
- Many-to-one connection: several models simultaneously send one entity to another model.
- A combination of the previous cases: several models simultaneously send one, or more, entities to another model.

The two following solutions have been applied to this problem.

1. Synchronizing the entity transmission between models using semaphores. The synchronization allows the sender and receiver to manage the flow of entities between both models, using a send/ACK mechanism like in the TCP/IP communication. Thus, the sender model will send an entity to the receiver and wait for an ACK. On the other hand, the receiver model will receive entities when it is ready to process them, and only send the ACK back if still ready to continue processing more entities. A model of the semaphore synchronization mechanism, based on a previous work by Lundvall and Fritzson [9], has been implemented and is freely available for download at [6]. A disadvantage of this solution is the performance degradation due to the event iteration that takes place during the synchronization phase of the entity transmission.
2. Including in the connector a `flow` variable that represents the number of entities sent from a model. So, the model receiving the entities will know the number of entities received, even with many senders. However, the information that describes several entities can not be transmitted simultaneously using the direct transmission approach. The variables of the connector that describe the entity can not be assigned with different values, that represent the different transmitted entities, at the same time. Anyway, the text file storage and dynamic memory storage approaches, discussed below, allow to solve this problem using the flow variable.

2.2.2 Text file storage

The idea is to define an intermediate storage for the transmitted entities. This storage behaves as a communication buffer between two or more modules.

The storage is implemented in a text file, that stores in each line of text the information related to each transmitted entity. The connector contains a reference to the text file, its file-name, and the flow variable indicating the number of entities received. This reference is shared between the models connected to that connector, allowing them to access the file. Each module able to receive entities, creates an storage text file and sets the reference to that file in the connector. Functions to read/write entities from/to the file have been developed. A model writes one or several entities to the file using the write function. Another function is used by the receiver to check the number of entities in the file. When there is any entity to be read, the receiver reads the entities and processes them. Thus, this approach allows the simultaneous reception of several entities.

A disadvantage associated with this approach is the poor performance due to the high usage of I/O operations to access the files. Also, the structure of the information stored in the files is not very flexible if any additional information has to be included. If new types of entities need to be used, or the attributes of an entity have to be changed, the file management functions (i.e. read and write) have to be reimplemented to correctly parse the text file to support these new changes.

2.2.3 Dynamic memory storage

In order to improve the performance of the text file approach, the intermediate storage was moved from the file-system to the main memory. Using the Modelica external functions interface, a library in C was created to manage the intermediate storage using dynamic memory allocation. An entity is represented in Modelica using a `record` class, and in C using its equivalent `struct` data structure. Entities are stored using linked-lists structures during their transmission from one model to another. This library is freely distributed together with the ARENALib Modelica library.

Instead of a reference to the file, the connector contains a reference to the memory space that stores the entities, together with the flow variable that indicates the number of entities received. That reference is the memory address pointing to the beginning of the linked-list. It is stored in an integer variable in the connector. Similarly to the text file approach, each model able to receive entities initializes the linked-list and sets the reference to it in the connector. Entities can be transferred to the queue using the write

function, and can be extracted using the read function. Another function is used to check the availability of received entities, in order to process them.

This approach also allows the simultaneous reception of several entities. The performance is highly increased compared to the text file approach. And, the structure of the information only depends on the data structures managed by the functions. To modify any attribute or entity type, it is only necessary to change a data structure and not all the functions used to manage that structure.

2.3 Entity Management

Regarding the entity management, it has to be mentioned that an additional problem appears when implementing processes that delay the entity. Arena process module can include a delay time that represents the time spent processing the entity. This delay time is usually randomly selected from a probability distribution. It has to be noticed that since the delay time is usually random, the order of the arrived entities need not correspond to the order of the entities leaving the process. These processes have to include a temporal storage for the entities that are being delayed. This problem can be solved using the text file storage or the dynamic memory storage as an additional storage for delayed entities. Due to performance reasons, the dynamic memory approach was used to manage entity storage during delays in ARENALib and SIMANLib.

Together with the initialization of the linked-queue for entity communication, a process module initializes a temporary storage, represented by a linked-list in memory, for delayed entities. The reference to that list is also stored in an integer variable. Every time the process module has to delay an entity, it stores the entity in the list using a write function. Entities are inserted in the list in increasing order, according to the time they must leave the process. The insertion of an entity in the list returns the leaving-time for the first entity in the queue. When the simulation time reaches the next leaving-time, the entity or entities leaving the process are extracted from the list and sent to the next module.

2.4 Stochastic Data Generation

Discrete-event models usually contain some kind of stochastic information. Random processing times, delays or interarrival times help to construct a more realistic model of a given system.

The Modelica language specification does not include any functionality for random number generation. Dymola, the modeling environment used to develop and test the mentioned Modelica libraries, includes two functions for generating random uniform and random normal variates [1]. The generation of random variates following other probability distributions is not covered by these random number generation functions. Also, the application of variance reduction techniques is not supported by these functions.

A random number generator (RNG) was developed by the authors. The RNG algorithm selected for its implementation in Modelica is the same that is used in the Arena environment. This allows the validation of the ARENALib models using the Arena environment, because both use the same source of random numbers. This RNG algorithm was proposed by Pierre L'Ecuyer and is called Combined Multiple Recursive Generator. A detailed description of the RNG is given in [13].

Additionally to the implementation of the RNG, some functions for generating random variates were also developed by the authors of this manuscript. The new RNG and the random variates generation functions are packaged in a Modelica library called RandomLib. This library is freely available for download at [6].

2.5 Statistical Information Management

Simulation results are usually reported using statistical indicators, due to the stochastic nature of discrete-event systems. Some of these statistical indicators have to be calculated during the simulation and some others at the end. The amount of data that has to be stored to calculate some of these indicators changes depending on the length of the simulation.

Modelica does not allow to declare variables with an undefined length or size, which are required to store the statistical data. A mechanism to declare variables of undefined length in Modelica needs to be defined, giving the possibility to increase or decrease the size of the variable during the simulation run.

This problem is very similar to the previously mentioned one about intermediate entity storage for transmission or delay management. So, the mentioned dynamic memory storage has been used in ARENALib to record the information regarding the statistical indicators of the simulation. The indicators calculated in each ARENALib module are shown in Table 1. Statistical indicators calculated include the number of entities arrived, the number of entities departed, processing times, the number of entities in queue, and the number of entities in the system, among others. The information calculated for each indicator is the mean, the maximum value, the minimum value, the final value and the number of observations. These values are updated during the simulation. On the other hand, all the intermediate values have to be recorded and used to calculate the confidence interval at the end of the simulation. A variable in Modelica stores a reference to the memory space that contains the stored data for each indicator. That space is managed using external functions written in C.

2.6 SIMANLib

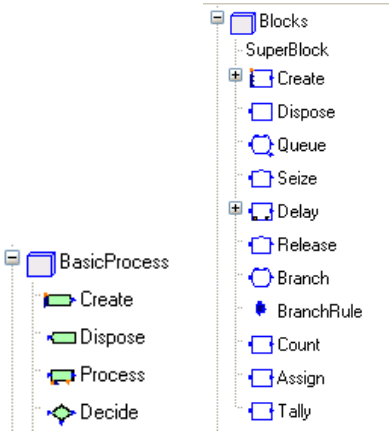
The first approach for the development of ARENALib was to write all its components, except the mentioned external functions and data types which are written in C, in plain Modelica code. This generated large and complex models that were difficult to understand, maintain and extend.

The idea then was to divide the actions performed by each module into simpler actions that combined will offer the same functionality than the original module.

The same structure can be observed in the Arena environment, where the modules are based and constructed

Module	Indicator	Values
Create	System.NumberIn	Obs
Process	NumberIn	Obs
	NumberOut	Obs
	VATime Per Entity	Avg, Min, Max, Final, Obs
	NVATime Per Entity	Avg, Min, Max, Final, Obs
	TotalTime Per Entity	Avg, Min, Max, Final, Obs
	Queue.NQ	Avg, Min, Max, Final
	Queue.WaitTime	Avg, Min, Max, Final, Obs
Dispose	System.NumberOut	Obs
EntityType	NumberIn	Obs
	NumberOut	Obs
	VATime	Avg, Min, Max, Final, Obs
	NVATime	Avg, Min, Max, Final, Obs
	TranTime	Avg, Min, Max, Final, Obs
	WaitTime	Avg, Min, Max, Final, Obs
	OtherTime	Avg, Min, Max, Final, Obs
	Work In Progress	Avg, Min, Max, Final

Table 1. Statistical indicators and values calculated in the ARENALib modules.



a) b)

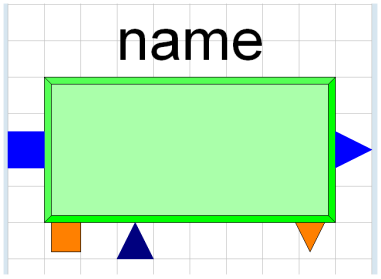
Figure 2. Flowchart modules: a) ARENALib; and b) SIMANLib.

using a lower level simulation language called SIMAN [18].

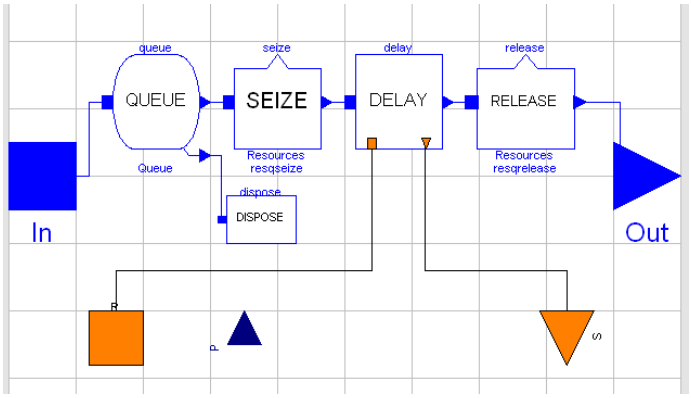
SIMANLib contains low-level components for discrete-event system modeling and simulation. These are low-level components compared to the modules in ARENALib, which represent the high-level modules for system modeling. Flowchart modules of both libraries are shown in Figure 2. ARENALib modules can be described using a combination of SIMANLib components. For example, the process module of ARENALib is composed by the Queue, Seize, Delay and Release blocks of SIMANLib, as shown in Figure 3.

Components in SIMANLib are divided, as well as in the SIMAN language, in two groups: blocks and elements. The blocks represent the dynamic part of the system, and are used to describe its structure and define the flow of entities from their creation to their disposal. The elements represent the static part of the system, and are used to model different components such as entities, resources, queues, etc.

An example of a model developed using SIMANLib is shown in Figure 4. This system is very similar to the



a)



b)

Figure 3. ARENALib process module: a) icon; and b) internal structure composed using SIMANLib components.

beverage manufacturing system mentioned above. The entities are pieces to be machined. The pieces arrive to the system and are processed by a machine, one at a time. After processed, the pieces are inspected by a supervisor and classified as Good, Reject and Repair. Repair pieces are sent back for re-processing.

3. Parallel DEVS in Modelica

The main objective of the implementation of the DEVSLib library has been to closely follow the definition of the Parallel DEVS formalism and implement all its features without restrictions. The functionalities of DEVSLib are similar to the ones offered by other DEVS environments such as DEVSJAVA [24] or CD++ [22]. These similarities include the new atomic and coupled models construction based on predefined classes, the redefinition of the internal, external, output and time advance functions in each atomic model as required by the user and the management of model input and output ports as needed. However, due to the capacities of the Modelica language, DEVSLib still presents some restrictions that will be discussed below.

3.1 DEVSLib Architecture

The architecture of the library is rather simple. It is shown in Figure 5a. It contains two main models, atomicDraft and coupledDraft, that represent the basic structures for building any new atomic or coupled DEVS models. Together with the main models there are several auxiliary models and functions for managing event transmission. Additionally, some examples of atomic and coupled sys-

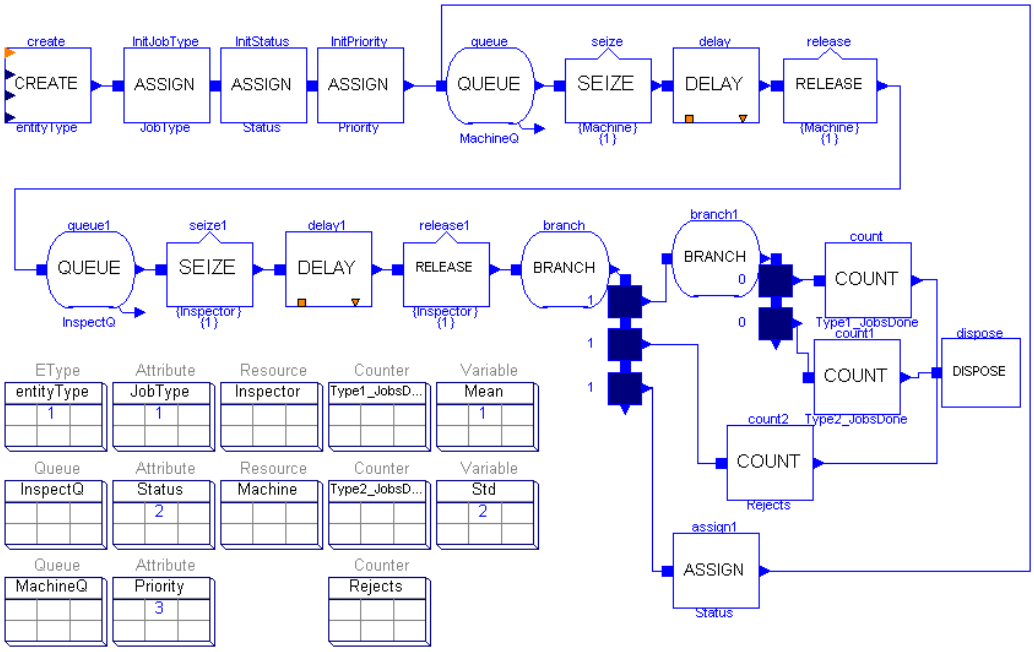


Figure 4. Manufacturing system model composed using SIMANLib components.

tems have been included. One of the included examples is the hybrid model of a pendulum clock [11], which is shown in Figure 5b. In this system a continuous-time model of a pendulum generates tics, acting as the motor of the clock. The rest of the clock receives the tics, calculates the current time (in hours and minutes) and manages the alarm of the clock.

3.2 Model Development with DEVSLib

When building a new atomic model, the user has to specify the actions to be performed by the external transition, internal transition, output and time advance functions. This can be performed redeclaring the functions `Fext`, `Fint`, `Fout` and `Fta`, initially declared in the atomicDraft model. The user can specify any desired behavior for these functions, while maintaining the defined function declaration. Any new atomic model has to extend the AtomicDEVS model and to redeclare the mentioned functions. The Modelica code of a processor system [23] developed using DEVSLib is shown in Listings 1, 2 and 3.

The desired number of input and output ports can also be included in the new model and managed with the mentioned functions. The user can drag and drop new input and output ports into the model. The prototypes of the external transition and the output function allow the user to check the port where an incoming event has been received, or to specify the output port to send the event. All these ports could be connected later to other models.

A coupled DEVS model, like the one shown in Figure 5b, can be easily build using previously defined atomic or coupled models, and connecting them as required. The input and output ports have to be included and connected to any of the model components

```
model processor
  extends AtomicDEVS(
      redeclare record State = st);
      redeclare function Fcon = con;
      redeclare function Fint = int;
      redeclare function Fext = ext;
      redeclare function Fta = ta;
      redeclare function initState =
                  initst(dt=processTime);
  parameter Real processTime = 1;
  Interfaces.outPortManager outPortManager1(
      redeclare record State = st,
      redeclare function Fout = out,
      n=1);
  Interfaces.outPort outPort1; // output port
  Interfaces.inPort inPort1;   // input port
equation
    iEvent[1] = inPort1.event;
    iQueue[1] = inPort1.queue;
    connect(outPortManager1.port, outPort1);
end processor;
```

Listing 1. Modelica code of a processor system modeled using DEVSLib.

3.3 DEVSLib Modeling Restrictions

One restriction in DEVSLib is the impossibility to perform one-to-many connections. These kind of connections are not considered in ARENALib or SIMANLib because neither Arena nor SIMAN permit them. However, the Parallel DEVS formalism allows this kind of connection so they have been taken into account.

This restriction appears because the way the port and the event communication mechanism is managed, using dynamic memory storage. As mentioned before, each receiver initializes its linked-queue to receive entities. A one-to-many connection cannot be performed because the sender can not store in just one integer variable the references to all the linked-queues created by the receivers. A solution

```
function con "Confluent Transtition Function"
  input st s;
  input Real e;
  input Integer q;
  input Integer port;
  output st sout;
  output st soutput;
algorithm
  soutput := s;
  sout := ext(int(s),e,q,port);
end con;

function int "Internal Transition Function"
  input st s;
  output st sout;
algorithm
  sout := s;
  sout.phase := 1; // passive
  sout.delta := Modelica.Constants.inf;
  sout.job := 0;
end int;

function ext "External Transition Function"
  input st s;
  input Real e;
  input Integer q;
  input Integer port;
  output st sout;
protected
  Integer numreceived;
  stdEvent x;
algorithm
  sout := s;
  numreceived := numEvents(q);
  if s.phase == 1 then
    for i in 1:numreceived loop
      x := getEvent(q);
      if i == 1 then
        sout.job := x.Value;
        Modelica.Utilities.Streams.
        print("* Event to process");
      else
        Modelica.Utilities.Streams.
        print("* Event balked");
      end if;
      sout.received := sout.received +1;
    end for;
    sout.phase := 2; // active
    sout.delta := s.dt; // processing_time
  else
    sout.delta := s.delta -e;
  end if;
end ext;

function out "Output Function"
  input st s;
  input Integer port;
  input Integer queue;
  output Boolean send;
protected
  stdEvent y;
algorithm
  if s.phase == 2 then
    send := true;
    y.Type := 1;
    y.Value := s.job;
    sendEvent(queue,y);
  else
    send := false;
  end if;
end out;

function ta "Time Advance Function"
  input st s;
  output Real delta;
algorithm
  delta := s.delta;
end ta;
```

Listing 2. Modelica code of the functions redeclared in the processor system.

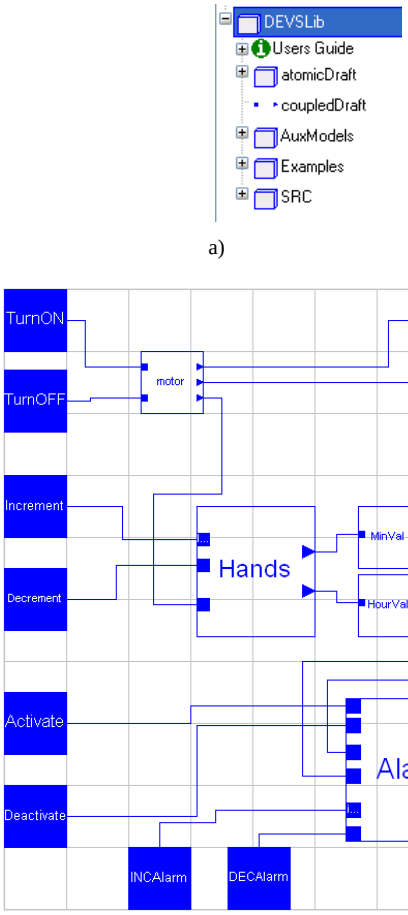


Figure 5. The DEVSLib Modelica library: a) architecture; and b) case of use (model of a pendulum clock).

```
record st "State of the model"
  Integer phase;    // 1 = passive, 2 = active
  Real delta; // internal transitions interval
  Real job;         // current processing job
  Real dt;          // default processing time
  Integer received; // num of jobs received
end st;

function initst "State Initialization Function"
  input Real dt;
  output st out;
algorithm
  out.phase := 1; // passive
  out.delta := Modelica.Constants.inf;
  out.job := 0;
  out.dt := dt;
  out.received := 0;
end initst;
```

Listing 3. Modelica code of the state and state initalization function of the processor system.

has been implemented in the DEVSLib library. This solution consists in an intermediate model that can be used to duplicate the events and send them to the receivers. Examples of this intermediate model are the *MinValue* and the *HourValue* models shown in Figure 5b.

By default, the information transmitted between models in DEVSLib, at event instants, is composed by two values:

the type of the event and a real value. The information communication mechanism using dynamic memory is relatively complex. It will not be easy for a user to change the structure of the information, type and value, transmitted in events. Anyway, it can be performed modifying the Modelica and C data structures that support the communication mechanism. In order to improve the mechanism for managing the information transmitted in events, additional information structures will be included to the DEVSLib library. For example, giving the possibility to transmit arrays or matrices instead of only real values.

4. Introducing Messages in Modelica

A conclusion of the performed work is that discrete-event system modeling with Modelica, using the process-oriented approach, is not an easy task. The components required for modeling these kind of systems and the solutions proposed for the problems are relatively complex. The developed libraries provide some functionalities for discrete-event system modeling with Modelica, using the process-oriented approach. Still, there are some problems without a solution, like the one-to-many connections in DEVSLib and the polymorphism of the information transmitted at event instants.

In this section the model communication using messages in Modelica is presented. The authors also propose a possible implementation of this mechanism that will be discussed in Section 5.

4.1 Motivation

The main difficulty observed in the presented work is the model communication mechanism. This is the way models are connected and communicate.

The connection of models in Modelica is represented by the `connect` equation. In a connection equation the value of the variables at the ends of the connection are either equaled, or summed and equaled to zero. A connection between discrete-event models does not establish any relation between variables of both models, but is used to communicate some information that has been generated in one model and is transmitted to another. Both connection concepts mean different things.

Event management is also different between Modelica and DEVS discrete-event systems. An event in Modelica involves a change in the value of a boolean condition that either makes the structure of the model to change, or performs a change in the discrete time variables or the state variables of the model. Events in DEVS discrete-event systems represent a change in the state of the system or its discrete time variables, and usually also involves the exchange of information between models. This is an instantaneous transmission/reception of an impulse of information between models at the time of an event. Event management in discrete-event systems involve additional things than in Modelica, because of this information communication.

In order to make the development of discrete-event systems more simple and easy, a new concept is proposed and introduced in Modelica. This concept is the messages communication mechanism. The messages mechanism provides the capacity for communicating impulses of information between models at event instants.

4.2 Messages and Mailboxes

The model communication mechanism using messages involves two parts: the message itself and the mailbox. The message represents the information either traveling from one model to another, or inside a model itself. The mailbox receives the incoming messages and stores them until they are read. The mailbox also represents the concept of a bag of events in the Parallel DEVS formalism.

The characteristics of the model communication using messages are the following:

- A message can be sent to any available mailbox. Available mailboxes are the ones that can be referenced from the model that sends the message, either accessing directly or using a connection.
- The mailbox warns the model when new incoming messages are received.
- Once received, the message can be read from the mailbox.
- The transmission of messages between models has to be performed instantly. Any message sent from one model will be immediately received by another model.
- Messages can be received simultaneously, either in the same or different mailboxes.
- The information transported by a message, the content, is independent from the message communication mechanism. It is a task of the user to define the structure of that information using the existing components of the Modelica language, so it can be managed by the models.
- Messages can be of different types. A mailbox can store any message independently of its type. The type of the message has also to be independent from the content of the message.
- Received messages have to be stored temporarily in the mailbox, until they are read.
- Message communication has to be performed in two stages: sending and reception. The sending involves the transmission of any message in the system at a given point in time, so all the messages sent are stored in the mailbox at the end. After the sending, all the messages are available for reception in each mailbox and can be read and managed as required. If a model sends several messages to the same mailbox, all the sent messages have to be stored in the mailbox before the first message can be read by the receiver.

4.3 Message Sending, Transmission, Detection and Treatment

A message can be sent from one model to any other model that contains a mailbox, even if no connection between models is available.

Mailboxes can also be shared between models. Sharing a mailbox represent that several models can access to the

message storage that it represents. Each model sharing the mailbox can access the messages stored, reading or extracting them from it. Read messages are kept in the mailbox until the are extracted, or fetched, from it.

An special case of mailboxes are the ones defined inside connectors. Two mailboxes, inside connectors, connected using a connect equation represent a bidirectional message communication pipe. They will act as input/output mailboxes instead of only receiving messages. A message sent to one end of the pipe will be transported to the opposite end, and viceversa. If more than two models are connected to the same pipe, a copy of the message will be transported to each receiver connected to the pipe. This provides a message broadcast functionality that also emulates the event transmission in DEVS, however in DEVS the communication is not bidirectional. The connect equation functionalities in Modelica have to be extended in order to support this mailbox behavior. An example of this behavior is shown in Figure 6.

The detection of a message is implicit in the action of sending it, since they are transferred instantly. Every time a model sends a message to a mailbox, the simulator knows that the message will be received by another model and will have to be treated properly.

The treatment of each message has to be defined by the user. The mailbox warns when a new message has arrived. The mailbox activates a listener function that can be used as a condition to detect any incoming message, used with statements like `when` or `if` in Modelica. This does not mean that the new message condition has to be effectively checked at each simulation step, because it is notified by the send message operation. Once a new message arrives to a mailbox, the arrived message or messages have to be read and treated.

5. Proposal of Implementation

This section contains a proposal of implementation in Modelica of the previously described message communication mechanism. This implementation is based on the definition of data structures that support the message and mailbox concepts, and the definition of the operations that can be performed with both data structures. Messages and mailboxes have to be defined as new predefined classes that have to be treated in a singular way, allowing objects of type message or mailbox. Due to the current Modelica language specification, the proposed implementation differs from the mechanism described above. The Modelica language will need to be extended in order to support the messages mechanism.

5.1 Data structures

There are two data structures needed to manage the messages mechanism. These are the definition of the message itself and the structure to support the mailbox that receives the defined messages.

The message structure contains two components: the type and the content. The type of a message can be represented with an integer value. It is used to separate the messages of the system in different classes. The content represents the information transported by the message. The content of a message is defined by the user and has to be independent from the message management mechanism. Thus, any mailbox can receive messages with any content and of any type. It is a task of the user to distinguish between the types of the messages and their contents. The content of the message is represented by a reference to an external data structure in C defined by the user. The user has to provide this data structure and the functions required to manage it using the reference in Modelica. Because of this definition, a message will be composed by two integer values: the type and the reference to the content.

The second structure required in the messages mechanism is the mailbox. A mailbox is a temporary storage for messages. If a message is sent to a mailbox, it is stored in the mailbox until the receiver reads it. The number of stored messages in a mailbox is not limited, so this structure has to be able to change its dimension depending on the number of stored messages. The implementation of a mailbox is very similar to the currently implemented linked-lists for storing delayed entities during processes.

5.2 Operations

The operations that can be performed with the previously described structures are defined below. Each operation is defined with its parameters and a short description of its behavior.

5.2.1 Mailbox Operations

- `newmailbox(mailbox)`. Initializes the mailbox.
- `checkmsg(mailbox)`. Warns about the arrival of a new message. It changes its value from false to true and immediately back to false at each message arrival event.
- `newmsg()`. Detects the arrival of a message to any of the mailboxes declared in the model. This helps to manage the simultaneous arrival of messages in different mailboxes.
- `nummsg(mailbox)`. Returns the number of waiting messages stored in the mailbox.
- `readmsg(mailbox,select)`. Reads a message from the mailbox. The select parameter represents a user-defined function used to select the desired message to be read from the mailbox.
- `getmsg(mailbox,select)`. Fetches a message from the mailbox, deleting it. The select parameter is used in the same way as in the `readmsg` function.
- `putmsg(mailbox,message)`. Sends the message to the mailbox.

5.2.2 Message Operations

- `newmsg(content,type)`. Creates a new message with the defined type and content.
- `gettype(message)`. Returns the type of the message.
- `settype(message,newtype)`. Updates the type of the message to the value of newtype.

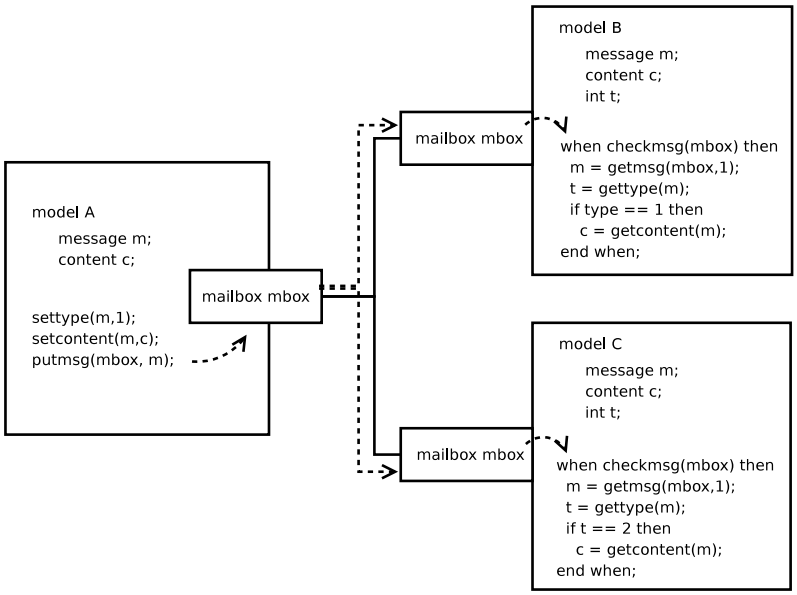


Figure 6. Model communication with messages using connectors.

- `getcontent(message)`. Reads the content of the message.
- `setcontent(message,newcontent)`. Inserts the newcontent into the message.

An example of a SIMAN single-queue system, with the Create, Queue, Seize, Delay, Release and Dispose blocks, modeled using the described messages mechanism is shown in Figure 7. Each block of the figure contains the pseudo-code that implements the basic actions for the entity management and communication. The select function, in the `readmsg` and `getmsg` functions, has been simplified and only represents the type of message to be read or extracted.

6. Conclusions

It has been observed that process-oriented modeling of discrete-event systems in Modelica is a difficult task. Several Modelica libraries have been developed to provide more discrete-event system modeling functionalities to Modelica, specially for modeling systems using the process-oriented approach. The implementation of these libraries present some problems and restrictions, and the solutions proposed and implemented are complex, hard to understand and difficult to maintain. In order to facilitate the development of discrete-event system models in Modelica, the message communication mechanism has been introduced and described. A possible implementation of this mechanism in Modelica has also been proposed.

Acknowledgments

This work has been supported by the Spanish CICYT, under DPI 2007-61068 grant, and by the IV PRICIT (Plan Regional de Ciencia y Tecnología de la Comunidad de Madrid 2005-2008), under S-0505/DPI/0391 grant.

References

[1] Dynasym AB. Dymola Dynamic Modeling Laboratory User's Manual. http://www.dymola.com/, 2006.

[2] Tamara Beltrame. Design and Development of a Dymola/Modelica Library for Discrete Event-Oriented Systems Using DEVS Methodology. Master's thesis, ETH Zürich, March 2006.

[3] Tamara Beltrame and Francois E. Cellier. Quantised State System Simulation in Dymola/Modelica using the DEVS Formalism. In *Proceedings of the* 5^{th} *International Modelica Conference*, pages 73–82, 2006.

[4] Francois E. Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[5] Mariana C. D'Abreu and Gabriel A. Wainer. M/CD++: Modeling Continuous Systems Using Modelica and DEVS. In *Proceedings of the* 13^{th} *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 229–236, 2005.

[6] http://www.euclides.dia.uned.es/.

[7] J.A. Ferreira and J.P. Estima de Oliveira. Modelling Hybrid Systems using Statecharts and Modelica. In *Proceedings of the* 7^{th} *IEEE International Conference on Emerging Technologies and Factory Automation*, pages 1063–1069, 1999.

[8] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Computer Society Pr, 2003.

[9] Håkan Lundvall and Peter Fritzson. Modelling concurrent activities and resource sharing in Modelica. In *Proceedings of the SIMS 2003 -* 44^{th} *Conference on Simulation and Modeling*, 2003.

[10] W. David Kelton, Randall P. Sadowski, and David T. Sturrock. *Simulation with Arena (*4^{th} *ed.)*. McGraw-Hill, Inc., New York, NY, USA, 2007.

[11] Julio Kriger. Trabajo práctico 1: Antiguo reloj des-

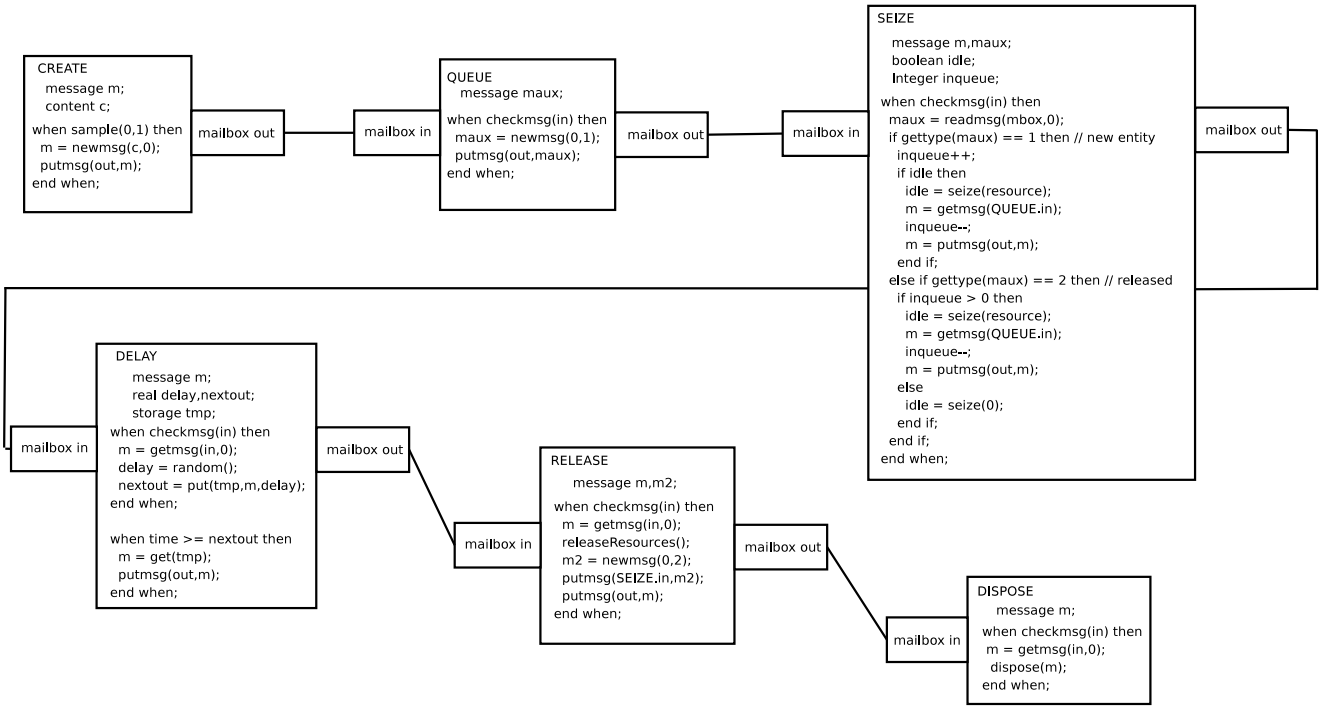


Figure 7. Example of a SIMAN single-queue system modeled using messages.

pertador. http://www.sce.carleton.ca/faculty/wainer/-wbgraf/samplesmain_1.htm.

[12] Averill M. Law. *Simulation Modelling and Analysis (4^{th} ed.)*. McGraw-Hill, 1221 Avenue of the Americas, New York, NY, 2007.

[13] Pierre L'Ecuyer. Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators. *Oper. Res.*, 47(1):159–164, 1999.

[14] Sven Erik Mattsson, Martin Otter, and Hilding Elmqvist. Modelica Hybrid Modeling and Efficient Simulation. In *Proceedings of the 38^{th} IEEE Conference on Decision and Control*, pages 3502–3507, 1999.

[15] Pieter J. Mosterman, Martin Otter, and Hilding Elmqvist. Modelling Petri Nets as Local Constraint Equations for Hybrid Systems using Modelica. In *Proceedings of the Summer Computer Simulation Conference*, pages 314–319, 1998.

[16] Martin Otter, Hilding Elmqvist, and Sven Erik Mattsson. Hybrid Modeling in Modelica Based on the Synchronous Data Flow Principle. In *CACSD'99*, pages 151–157, 1999.

[17] Martin Otter, Karl-Erik Årzén, and Isolde Dressler. StateGraph - A Modelica Library for Hierarchical State Machines. In *Proceedings of the 4^{th} International Modelica Conference*, pages 569–578, 2005.

[18] C. Dennis Pegden, Randall P. Sadowski, and Robert E. Shannon. *Introduction to Simulation Using SIMAN*. McGraw-Hill, Inc., New York, NY, USA, 1995.

[19] Manuel A. Pereira Remelhe. Combining Discrete Event Models and Modelica - General Thoughts and a Special Modeling Environment. In *Proceedings of the 2^{nd} International Modelica Conference*, pages 203–207, 2002.

[20] Victorino Sanz, Alfonso Urquia, and Sebastian Dormido. ARENALib: A Modelica Library for Discrete-Event System Simulation. In *Proceedings of the 5^{th} International Modelica Conference*, pages 539–548, 2006.

[21] Victorino Sanz, Alfonso Urquia, and Sebastian Dormido. DEVS Specification and Implementation of SIMAN Blocks Using Modelica Language. In *Proceedings of the Winter Simulation Conference 2007*, pages 2374–2374, 2007.

[22] Gabriel Wainer. CD++: A Toolkit to Develop DEVS Models. *Softw. Pract. Exper.*, 32(13):1261–1306, 2002.

[23] Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., Orlando, FL, USA, 2000.

[24] Bernard P. Zeigler and Hessam S. Sarjoughian. Introduction to DEVS Modeling & Simulation with JAVA: Developing Component-based Simulation Models. - http://www.acims.arizona.edu/PUBLICATIONS/.

EcosimPro and its EL Object-Oriented Modeling Language

Alberto Jorrín César de Prada Pedro Cobas

Department of Systems Engineering and Automatic Control, University of Valladolid, Spain,
{albejor,prada}@autom.uva.es.
Ecosimpro, Empresarios Agrupados, Spain, {Pedro Cobas} pce@ecosimpro.com

Abstract

This paper introduce the modeling and simulation tool EcosimPro and the possibility in the new version 4 of using object orientation in its modeling language called EL (Ecosimpro Language) which is the *official language of The European Space Agency* (ESA). Also shows the power that the use of classes gives to EcosimPro regarding other simulation environments.

1. Introduction

1.1 What is EcosimPro

EcosimPro is a simulation tool with a user-friendly environment developed by Empresarios Agrupados International for modeling simple and complex physical processes that can be expressed in terms of differential-algebraic equations or ordinary-differential equations and discrete events.

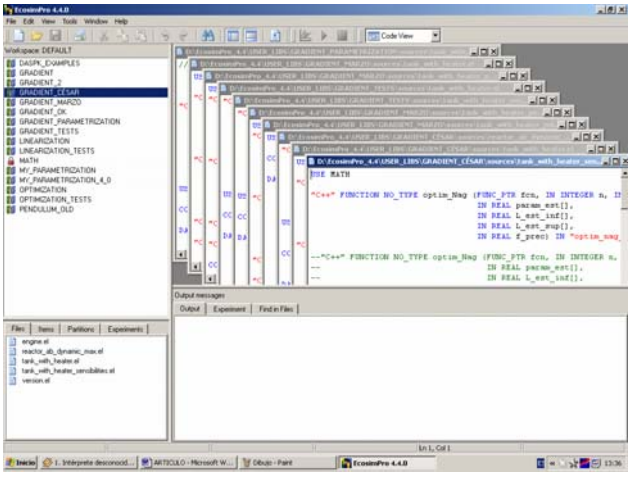


Figure 1. General view of the tool.

EcosimPro runs on the various Windows platforms and uses its own graphic environment for model design.

1.2 What is EL

The language used in EcosimPro simulation tool for modeling dynamic systems based on equations and discrete events is named EL, which is the *official language of The European Space Agency* (ESA).

EL has been developed for use in modeling combined continuous-discrete physical systems. It allows mathematical modeling of complex components represented by differential-algebraic equations. One of the design goals for EL was to make it clear and easy to use for engineers doing simulations of this kind.

Models in EL are represented in a natural and intuitive way. EL has a simple but powerful language to prepare experiments on the models created, calculate steady states, transients, perform parametric studies, etc. It can also generate reports, plots and other hard copies from within a classic sequential language, and has the ability to reuse C and FORTRAN functions and C++ classes.

1.3 Key concepts in EcosimPro

The fundamental concepts of EcosimPro are:

- **Component**: This represents a model of the system simulated by means of variables, differential-algebraic equations, topology and event-based behaviour. The component is the equivalent of the “class” concept in object-oriented programming.
- **Port connection type**: This defines a set of variables to be interchanged in connections and the behaviour and restrictions when there are connections between more than two ports. For instance, an electric connection type uses voltage and current as variables to be used in connections. The connection port avoids connecting individual variables; instead, sets of variables are managed together.
- **Partition**: To simulate a component, the user first has to define its associated mathematical model; this is called a partition. A component may have more than one partition. For example, if a component has several different boundary conditions, depending on the set of variables selected, each set of variables produces a different mathematical model, or partition. The next step is to generate experiments for each

partition. The partition defines the causality of the final model.

- **Experiment**: The experiments performed for each partition of the component are the different simulation cases. They may be trivial for calculating a steady state or very complex with many steady and transient states changing multiple variables in the model.
- **Library of components**: All the components are classified by disciplines into libraries.

1.4 EL and Object Orientation

EL is a language for automatically solving systems of differential-algebraic equations, as this is how it works internally. However, the complexity of these systems is hidden: all the user has to do is define high-level equations using a high level object-oriented language similar to some programming languages and expressing the equations as algebra does.

EL is object-oriented: components can inherit from one another and can be aggregated to create other more complex, modular components. These features allow the modeller to reuse tried and tested components to create other more complex ones, incrementally.

1.5 The Component in EL

The most important element in EL is the component. A component represents a model by means of variables, topology, equations and an event-based behaviour.

A component may be simple, for example an electrical resistor or capacitor (with a couple of equations); or it may be very complex, for example the pressurized cabin of a space vehicle (with dozens of physics equations and many events).

All components have one block which represents the continuous equations and another which handles all the discrete events. Before the model can be run, it has to be generated. Until then, the components are only theoretical entities waiting to be used by other components or generated into a model.

1.6 The Experiment in EL

Once a model has been generated, experiments can be run on it.

EL has an experiment language similar to the modeling language which is used to integrate the model, calculate steady states, optimize parameters, etc

1.7 EL Basis

EL's design is based on continuous modeling concepts developed in the 1970s, new ideas of the '80s and '90s, and modern object-oriented techniques applied successfully in other fields.

1.8 EL Uses

EL has been designed to be used in industry directly, ranging from simple systems to very complex systems with hundreds of variables and equations. EL has mainly been successfully used for aerospace applications to simulate complex systems in Environmental Control and Life Support Systems (ECLSS) and in power generation to model complex power plants.

1.9 Mathematical capabilities

EcosimPro has Symbolic handling of equations (e.g.: derivation, equations reduction, etc.) and robust solvers for non-linear equations (Newton-Raphson) and DAE systems (DASSL and Runge-Kutta). EcosimPro controls the mechanism to interact with these solvers.Also uses dense and sparse matrix formats depending on the size of the Jacobian matrix. This allows problems with thousands of state variables to be simulated.

EcosimPro has Math wizards for:

- Defining design problems
- Defining boundary conditions
- Solving algebraic loops
- Reducing high-index DAE problems

and clever mathematical algorithms based on graph theory to minimize the number of unknown variables and equations.

Also incorporate Powerful discrete events handler to detect when events occur and powerful root finder mechanism based on Zbrent and Illinois methods. It is completely transparent to the user so the exact moment of the crossover of discrete events can be determined.

2. Object Oriented Modeling

Object-oriented Modeling is a powerful and intuitive paradigm for building models which will outlive the inevitable changes that are part of the growth and ageing of any dynamic system. It is not a panacea, but at least it provides the modeller with powerful features to hide complexity (encapsulation), to enable reuse (inheritance and aggregation) and to create models which are independent and easy to maintain.

Modular development allows a system to be modelled bottom-up. Basic library components can be combined to create increasingly complex components by combining two methods:

- Extension by inheritance from existing components
- Instantiation and aggregation of existing components

These ideas are applied to create a component which represents a complete system. The intermediate components can also be simulated. This greatly reduces development and maintenance time.

2.1 Encapsulation

One of the aims of object-oriented Modeling is encapsulation of complexity. For example, an engineer could model a complex problem with a purpose-built model consisting of hundreds of variables and equations. Although the model works, it will probably be difficult to follow and understand, and will certainly be difficult to reuse for other similar problems.

EL makes the modeller's life easier with its powerful abstraction capability and encapsulation of data and behaviour. Its main elements are libraries and components.

The libraries encapsulate all the elements involved in a particular discipline, and are exposed for use by other libraries. The component is a fundamental item for Modeling to express a dynamic behaviour associated with certain data.

With a conventional object-oriented language such as C++, the public interface is the data and methods declared public.

In EL a component's public interface consists of its ports, construction parameters and data. These elements are unique and are visible from outside during Modeling, reinforcing encapsulation and favouring reuse of the component.

Components are connected by their ports. The port definition includes the variables for a connection and their behaviour and restrictions when there are connections between more than two ports. By defining the behaviour of the ports, the system is able to automatically insert several connection equations, so the user does not need to be concerned about them. To add a new component, the user only has to be concerned about its internal behaviour and not about the connection equations.

2.2 Inheritance

Inheritance is what gives EL its tremendous power for sharing interfaces and behaviour.

In all fields of simulation, when many components are developed it becomes apparent that they share much of the same behaviour. EL can bring together common data and equations in parent components, to be inherited by their child components. This allows the creation of libraries based on parent components with a linear rather than geometric order of complexity. A new component based on another parent will include all its data and behaviours.

EL provides multiple inheritance; i.e., a component can inherit data and behaviour from one or many components which have previously been designed and tested. This capability allows the creation of new components reusing parts of others which have already been created.

2.3 Aggregation

In EL, all Modeling items are components. As described above, a component can inherit from another (or others), and can contain multiple instances or copies of others internally. This concept enforces the capability of reuse because it allows compound components to be created based on others which have already been developed.

This paradigm is applied iteratively and has no limit. The complexity of a final component can be hidden by the aggregation of tried and tested internal instances of other components or classes.

2.4 Data Abstraction

EL provides typical numerical data like REAL and INTEGER as well as other convenient data types like BOOLEAN and STRING. It also offers enumerative types to define the user's own data (very useful in chemical Modeling).

In addition, EL has multidimensional arrays, 1D, 2D and 3D tables and pointers to functions.

3. Classes

3.1 Introduction

Classes in EL are the equivalent to classes in classic object-oriented programming languages such as C++ and Java, but their use is more restricted (and simple). In fact, they are like high-level wrappers for producing a C++ class but bearing in mind that *the final users are engineers and not programmers*.

There are times when the modeller wants to encapsulate data and behaviour in the same item, and later instantiate them and use them by means of certain methods.The main difference between a class and a component in EL is that a component is meant to include dynamic equations and discrete events that the simulation tool arranges and solves, whereas a class represents a set behaviour and only allows the publication of variables and methods.

Classes are normally used in EL to support the Modeling of complex systems where the use of functions is sometimes improved if all the functions referring to the

same utility are grouped together and share memory by means of common variables.

The general syntax to define classes is as follows:

```
class_def : CLASS IDENTIFIER ( IS_A
scoped_id_s)?
DESCRIPTION?
( DECLS var_object_decl_s )?
( OBJECTS class_instance_stm_s )?
( METHODS method_def_s )?
END CLASS
```

As all elements are optional, a very simple valid class would be:

```
CLASS math
END CLASS
```

An example of a slightly more complete class would be:

```
CLASS point2D “class reprenting a 2D
point”
DECLS
PRIVATE REAL x
PRIVATE REAL y
METHODS
METHOD NO_TYPE set2D(IN REAL valueX,
IN REAL valueY)
BODY
x = valueX
y = valueY
END METHOD
METHOD NO_TYPE get2D(OUT REAL valueX,
OUT REAL valueY)
BODY
valueX = x
valueY = y
END METHOD
END CLASS
```

This class represents the coordinates of a 2D point. It has a description, two private variables named “x” and “y” and two methods named “set2D()” and “get3D()”.

3.2 Inheritance

As seen above, a class can be inherited from one class (simple inheritance) or more (multiple inheritance) at the same time, in which case it will inherit all the associated variables and methods exactly as if they had been defined in the class itself.

To give an example of simple inheritance we can define a new class point3D as:

```
CLASS point3D IS_A point2D "a 3D
point"
DECLS
PRIVATE REAL z
METHODS
METHOD NO_TYPE set3D(IN REAL valueX,
IN REAL valueY,IN REAL valueZ)
“Method to init a 3D point”
BODY
set2D(valueX,valueY)
z = valueZ
END METHOD
METHOD NO_TYPE get3D(OUT REAL valueX,
OUT REAL valueY,OUT REAL valueZ)
“Method to obtain a 3D point”
BODY
get2D(valueX,valueY)
valueZ = z
END METHOD
END CLASS
```

This example shows how inheritance is used to inherit "point3D" class from "point2D" class and thus inherit their variables “x” and “y” and the methods “set2D()” and “get2D()”.To see an example of multiple inheritances, another class can be created.

```
CLASS statusClass “a status class”
DECLS
PRIVATE BOOLEAN status
METHODS
METHOD NO_TYPE setStatus(IN BOOLEAN
sta)
BODY
status= sta
END METHOD
METHOD BOOLEAN getStatus()
BODY
RETURN status
END METHOD
END CLASS
```

Then it can be inherited in the definition of point3D.

```
CLASS point3D IS_A point2D,
statusClass “a 3D point”
....
END CLASS
```

Here, the child class will inherit all variables and methods from parent classes; in other words, every point3D object will have access to all the public parts of classes “point2D”,“statusClass” and “point3D”.

Variables or methods cannot be inherited with the same name as this would produce an error in the compiler.

3.3 DECLS Block

With the DECLS block of a class, any kind of basic EL variable can be defined, whether this be a simple variable or a multidimensional array. For example, the following

declarations are valid:

```
CLASS testClass
DECLS
REAL x = 9.9
REAL z,y = 1.1
INTEGER v[3] = {1, 2, 4}
BOOLEAN stat
STRING str = "hello world"
END CLASS
```

Defined in it are variables such as REAL, INTEGER, BOOLEAN and STRING. Initial values are also assigned. The general syntax for defining variables in classes is as follows:

```
var : PRIVATE? CONST? data_type name_s
( '=' init_expression )? STRING_VALUE?
```

Examples of declarations are:

```
PRIVATE REAL x = 1

PRIVATE CONST REAL x = 1

REAL speed= 1 “speed of the aircraft
(m/s)”
```

3.4 OBJECTS Block

This section allows us to declare objects that are instances of classes. The objects are written in the OBJECTS block of classes, components, functions and experiments. The general syntax of the objects declaration is as follows:

```
PRIVATE? names STRING_VALUE?
```

Valid declarations are:

```
OBJECTS
point3D p1
PRIVATE statusClass object1, object2
Point2D points[3,4]
```

3.5 METHODS Block

Methods define the functional interface of a class. They are subroutines connected to a definition of a class. They are always declared within a class in the METHODS block and can only be invoked from instances of that class.

Like functions, a method can return a basic EL type and has a number of call arguments which are defined when the method is written.

The general syntax of a method is:

```
method_def : PRIVATE? METHOD data_type
IDENTIFIER
'(' EOL* func_arg_decl_s ')'
STRING_VALUE?
( DECLS var_decl_s )?
( OBJECTS class_instace_stm_s )?
( BODY seq_stm_s )?
( END METHOD )?
```

An example of its use is:

```
CLASS example
DECLS
PRIVATE REAL x
METHODS
-- method to increment x with value v
(returns nothing)
PRIVATE METHOD NO_TYPE incr(IN REAL v)
BODY
x= x + v -- increase the class
variable x
END METHOD
-- method to return the value of x
after increasing it with value v
METHOD REAL popValue(IN REAL v)
BODY
incr ( v )
RETURN x
END METHOD
END CLASS
```

3.6 Using Classes

Classes defined in EL can be used in functions, components, experiments and in other classes.

Instances in classes are always defined in the OBJECTS block of each statement, as described above.These objects are used the same way as in other object-oriented programming languages. All their variables of instance and public methods (eg, those which are not tagged PRIVATE in their definition) are directly accessible by using the point (.) operator. The general rule is:

```
OBJECTREF.METHOD(....)
OBJECTREF.VARIABLE
```

For example:

```
object1.setValue(5)
object1.speed= 4
object1.foo.setValue(4)
```

The following class uses objects of “example” class defined in the previous section:

```
CLASS useExample
METHODS
```

```
METHOD NO_TYPE use()
DECLS
REAL v
OBJECTS
example ex -- declare object named ex
BODY
ex.popValue(2)
RETURN v
END METHOD
END CLASS
```

3.7 Class Associated With a Partition

When generating a partition, the tool can automatically generate an internal class representing the mathematical model generated. This provides a number of advantages:

- Any partition can be encapsulated in a single class
- This class provides an interface for interacting with the partition. For instance, initialization of variables, steady and transient calculations, get values of variables,etc
- Simulations can be embedded in components, functions, experiments and classes, since they are programmed with the class interface
- Multiple experiments can be executed in the same run
- Child classes (inherited from the partition classes) can be created by adding new variables and methods. In fact, a child class could provide complex experiments embedded in a single method.

To automatically generate the class associated with the partition, the user must select the option:

"Generate an associated class for a partition"

located in GUI option Edit->Options->General. In this case, each time the modeller makes a partition, an internal class will be generated with the name:

"ComponentName_PartitionName"

Use the underscore (_) separator to create a joint name from the component and partition names, such as aircraft_transient.

Once the internal class has been created, the modeller can declare an object of that class from any OBJECTS block, such as:

```
OBJECTS
aircraft_transient air
```

3.7.1 Access to Variables During Simulation

The user can access any model variable of the partition in these special classes. Since the variables can be of different types, there are different methods that allow the user to access the type of variable, to see if the variable exists, etc. Here is an example of available methods:

```
INTEGER getNumberVars ()
REAL getValueReal (IN STRING name)
BOOLEAN setValueReal (IN STRING name,
IN REAL v)
...
```

3.7.2 Operations Allowed with Classes of Models

By using these types of classes, you can perform any calculation with the partitions as if you were writing an experiment. For example, you can perform steady state and transient calculations on the same model.

Let's look at an example of an experiment carried out on the component aircraftGear from the DEFAULT_LIB library. The experiment exp1 carries out the following transient study:

```
EXPERIMENT exp1 ON
aircraftGear.default
INIT
-- Dynamic variables
y3 = 0.
y3' = 0.
y2 = 0.
y2' = 0.
x = 0.
x' = 60.96
BODY
REPORT_TABLE("reportAll", " * ")
TIME = 0.
TSTOP = 10.
CINT = 0.05
INTEG()
END EXPERIMENT
```

This same experiment can be written using these special classes. Suppose we create a new component called useAircraft and that in its *initialization* (INIT block) we want it to perform a transient calculation of the partition aircraftGear_default just as we did in the experiment. The code would be:

```
COMPONENT useAircraft
OBJECTS
aircraftGear_default air
INIT
-- set initial values
air.setTraceProgramme(TRUE)
air.setValueReal("y3", 0)
air.setValueReal("y3'", 0)
air.setValueReal("y2", 0)
air.setValueReal("y2'", 0)
air.setValueReal("x", 0)
air.setValueReal("x'", 60.96)
-- integrates the model
air.REPORT_TABLE("rAir", " * ")
air.TIME = 0.
```

```
air.TSTOP = 10.
air.CINT = 0.05
air.INTEG()
END COMPONENT
```

When using this component, a transient study will be executed for the aircraftGear model at the beginning of the calculation. This way, we have managed to embed a calculation of a mathematical model into another component. This makes the language very powerful for embedding mathematical models inside others.

3.7.3 Activation of Flags

By default, performing calculations on any class associated with a partition will not produce any on-screen messages or log files.

There is a set of functions to activate and deactivate these flags which print simulation messages when running, return the actual status of flag for tracing the simulation, print simulation messages in the log file, check assertions when running, ...

3.7.4 Creation of Classes Based on Partition Classes

We can also create our own classes by inheriting them from the automatically generated class of partition. This way, a more user-friendly interface can be created for operating with a mathematical model.

For example, a new class can be created by inheriting it from the class aircraftGrear_default and writing the experiment written in the INIT block of the example in the previous section in a method of the class.

```
CLASS aircraftTransient IS_A
aircraftGear_default
METHODS
METHOD NO_TYPE run()
BODY
-- set initial values
setTraceProgramme(TRUE)
setValueReal("y3", 0)
setValueReal("y3'", 0)
setValueReal("y2", 0)
setValueReal("y2'", 0)
setValueReal("x", 0)
setValueReal("x'", 60.96)
-- integrates the model
REPORT_TABLE("rAir", " * ")
TIME = 0.
TSTOP = 10.
CINT = 0.05
INTEG()
END METHOD

END CLASS
```

In this example, it has been embedded the same experiment in a method of a new class called “aircraftTransient”. Thus, we can now re-write the class useAircraft as:

```
COMPONENT useAircraft
OBJECTS
aircraftTransient air
INIT
-- run the experiment
air.run()
END COMPONENT
```

This, as can be seen, is a great simplification, since it only calls the method “run()” which encapsulates the whole experiment.These classes have all the properties of a normal class and can define new variables and methods, be inherited by others, etc. This gives great flexibility for encapsulating experiments in methods of classes and reusing them later.

3.7.5 An ilustrative example: Initialization of models

The main objective of this problems is to allow to start a simulation from stationary conditions.To formulate this kind of problems in EL, classes are used.

For instance:

In a problem of an eletrical engine:

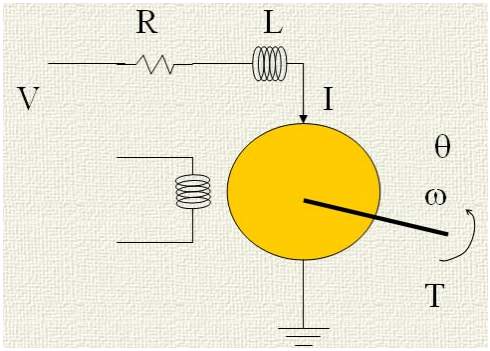


Figure 2. Electrical engine schematic.

whose model is given by:

$$J\frac{d\omega}{dt} = k_1 i - f\omega - T \qquad \text{Newton Law}$$

$$V = Ri + L\frac{di}{dt} + k_2\omega \qquad \text{Ohm Law}$$

ω angular velocity

J moment of inertia

V source voltage

I armatura current

T external torque

$k_2\omega$ f.c.e.m

Figure 3. Electrical engine dynamic equations.

The codification in EL is for instance:

```
COMPONENT engine

   DATA

      REAL R = 0.2      -- electric resistance (ohmios)
      REAL L = 0.01     -- electric inductance (H)
      REAL k1 = 0.006   -- armature constant (lbs-pie/A)
      REAL f            -- damping ratio of the mechanical system(lbs-pie/rad/seg)
      REAL J = 0.001    -- moment of inertia of the rotor (slug-pie2)
      REAL k2 = 0.055   -- speed constant (V.seg/rad)

   DECLS

      REAL V            -- source voltage (V)
      REAL omega        -- angular velocity
      REAL i            -- armatura current
      REAL T            -- motor torque applied to the shaft


   CONTINUOUS

      J * omega' = k1*i - f *omega - T
      L*i' = V - R*i - k2*omega


END COMPONENT
```

Figure 4. Electrical engine dynamic codification in EL.

The partition to work with that model for example could be:

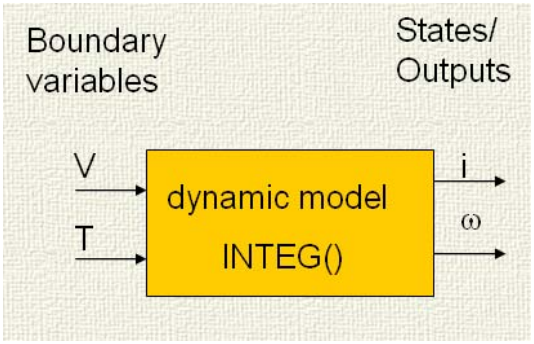


Figure 5. Partition of the system.

where the variables "V" and "T" are the boundary variables and "i" and "w" are the states and outputs of the process.

Intuitively, if the model were easy, to start with stationary conditions, only would be necessary write the equations in the INIT block by this way:

$$i(0) = \frac{V(0) - k_2\omega(0)}{R}$$

$$T(0) = k_1 i(0) - f\omega(0)$$

Figure 6. Initializations.

turning the derivatives into null and clearing the outputs up.

But this operation only could be done if the model equations are easy.

In general this problem could be solved using classes and creating a static partition with the component. In this case the static partition would be for example:

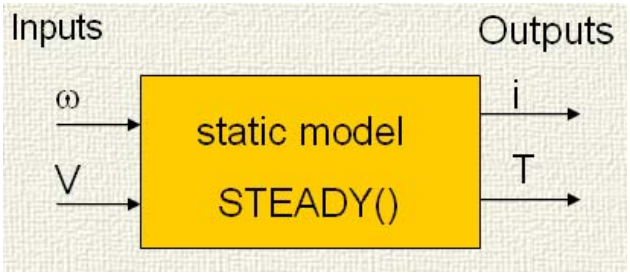


Figure 7. An example of a partition to initialize the dynamic model at stationary conditions.

where V and W are the boundary variables and inputs. With "w" and "v" values could be computed the i(0) value.

$$i(0) = \frac{V(0) - k_2\omega(0)}{R}$$

Figure 8. First calculation.

and with the i(0) value, could be computed the T(0) value.

$$T(0) = k_1 i(0) - f\omega(0)$$

Figure 9. Second calculation.

This partition could be used in the INIT zone of the dynamic component, so another component would have been created with the static equations.

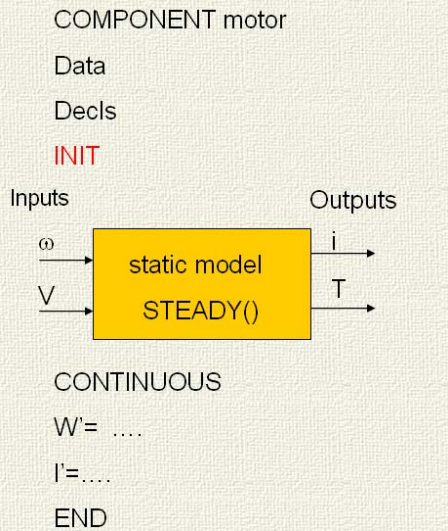


Figure 10. New component structure schematic.

So, the codification of the component using classes would be:

```
CLASS Stationary IS_A engine_static

METHODS
   METHOD NO_TYPE run()

      BODY
         STEADY()

   END METHOD

END CLASS


COMPONENT engine_test

   DATA

      REAL R = 0.2      -- electric resistance (ohmios)
      REAL L = 0.01     -- electric inductance (H)
      REAL k1 = 0.006   -- armature constant (lbs-pie/A)
      REAL f            -- damping ratio of the mechanical system(lbs-pie/rad/seg)
      REAL J = 0.001    -- moment of inertia of the rotor (slug-pie2)
      REAL k2 = 0.055   -- speed constant (V.seg/rad)

   DECLS

      REAL V            -- source voltage (V)
      REAL omega        -- angular velocity
      REAL i            -- armatura current
      REAL T            -- motor torque applied to the shaft

   OBJECTS

      Stationary stac

   INIT

      --i0 = (V0-k2*omega0)/R
      --T0 = k1*i0-f*omega0


      omega = 8.
      V = 5.

      stac.setValueReal("omega",omega)
      stac.setValueReal("V",V)

      stac.setTraceProgramme(TRUE)
      stac.STEADY()

      i = stac.getValueReal("i")
      T = stac.getValueReal("T")


   CONTINUOUS

      J * omega' = k1*i - f *omega - T
      L*i' = V - R*i - k2*omega


END COMPONENT
```

Figure 11. New Electrical engine dynamic codification in EL using classes to initialize .

Finally using that component and that class the simulation will start from stationary conditions.

4. Conclusions

The *object-oriented programming language* of *EcosimPro*, known as EL, is therefore one of the ***pioneer languages*** that has to deal with this new way of Modeling physical systems.

So finally the advantages of this type of methodology that offers the modeller over the others are going to be list:

- The modeller encapsulates data and behaviour into individual components (minimise global data)
- A component hides the complexity by making public only a certain part of the component
- The public interface comprises parameters, data and ports, while the local variables, discrete events and equations remain private
- The complexity grows in a linear rather than a geometrical way
- Reuse of tested components
- Inheritance simplifies the Modeling by sharing many common data and equations. Less code, more productivity and easier maintainability
- A component can contain equations that can be inserted at the time of simulation or not, depending on certain parameters passed to the component
- Use of virtual equations. Some equations change from parents to children. Users can decide to overwrite a parent equation with their own
- Equations format is declarative. Algorithms will symbolically transform the equations so that they fit in the best possible way to be solved numerically
- In general, it be can quoted the following considerations in the revolution of object-oriented Modeling:
- Modeling is non-causal. In other words, when a component is modelled, the causality of equations is not given. This will be solved during the final moment of simulation. Libraries of generic and reusable components can therefore be created in all situations
- Tried and tested components are constantly reused through simple or multiple inheritance and aggregation
- There is extensive use of hidden information and encapsulated data to deal with the complexity, solving each Modeling problem in its associated component and not taking other global data into account
- It is easier to make changes in the models because everything is divided into parts

Acknowledgements

A few words of thanks to the Empresarios Agrupados staff for giving me kindly all the information that I need.

References

[1] EcoEC: EcosimPro External Conections Version 4.4. EA International. 2008
[2] EcoEML: EcosimPro EL Modeling Language Version 4.4. EA International. 2008
[3] EcoIGS: EcosimPro Installation and Getting Started Version 4.4. EA International. 2008
[4] EcoMASG: EcosimPro Mathematical Algorithms and Simulation Guide Version 4.4. EA International. 2008
[5] EcoUM: EcosimPro User Manual Version 4.4. EA International. 2008
[6] EcoVU: EcosimPro Version 4.4 Upgrade. EA International. 2008
[7] www.ecosimpro.com

Activation Inheritance in Modelica

Ramine Nikoukhah
INRIA, BP 105, 78153 Le Chesnay, France
ramine.nikoukhah@inria.fr

Abstract

Modelica specifies two types of equations: the equations defined directly in the "equation" section, which are supposed to hold all the time, and the equations defined within a "when" statement. The latter are "activated" by explicit events at corresponding times. In making the analogy with Scicos, the equations of the first type are the counterpart of "always active" blocks whereas the second type equations are "event activated" blocks. A useful feature in Scicos is the mechanism of activation inheritance. In this paper, we examine the possible extension of the Modelica specification to introduce a similar mechanism in Modelica.

Keywords Modelica, Synchronous languages, Scicos, modeling and simulation

1. Introduction

Modelica (www.modelica.org) is a language for modeling physical systems. It has been originally developed for modeling systems obtained from the interconnection of components from different disciplines such as electrical circuits, hydraulic and thermodynamics systems, etc. These components are represented symbolically in the language providing the compiler the ability to perform symbolic manipulations on the resulting system of differential equations. But Modelica is not limited to modeling continuous-time systems; it can be used to construct hybrid systems, i.e., systems in which continuous-time and discrete-time components interact [1]. Modelica specification [2] defines the way these interactions should be interpreted and does so by inspiring from the formalism of synchronous languages. Modelica hybrid formalism can be interpreted or extended to include many features available already in Scicos. We have examined some of these issues in previous papers [4][5]. We consider here the implementation of the activation inheritance mechanism of Scicos in Modelica.

Scicos (www.scicos.org) is a modeling and simulation environment for hybrid systems. Scicos formalism is based on the extension of synchronous languages, in particular Signal [3], to the hybrid environment. The class of models that Scicos is designed for is almost the same as that of Modelica. So it is not a surprise that Modelica and Scicos have many similar features and confront similar problems. Just as in Modelica, Scicos is event driven, to be more precise, activation driven, i.e., the activation of equations are explicitly specified. This is in opposition to data-flow formalism where the activation is implicitly derived from the flow of data.

The data-flow mechanism has however many advantages in certain situations. That is why Scicos includes an activation inheritance mechanism providing a data-flow-like behavior without perturbing the overall formalism. The activation inheritance is treated at an early compilation phase.

In this paper, we examine the extension of the Modelica specification to implement an inheritance mechanism similar to the one available in Scicos. We work here only with flat Modelica models (usually obtained from the application of a front-end compiler).

This extension can have many applications. In the last section, we consider its possible use in the problem of code generation for control applications.

2. Basic Idea

There is a clear distinction between equations activated by inheritance and those that are declared “always active” in Scicos. This is not the case in Modelica where both types are placed in the `equation` section outside of `when` clauses. To distinguish the two types of activations, we propose a new `when` clause in this section. We start by considering only discrete dynamics.

Consider the following valid Modelica code:

```
  discrete Real z;
  Real u,y;
equation
  u=2*y;
  y=pre(z);
  when sample(0,1) then
    z=pre(z)+u;
  end when;
```

and compare it to

```
  discrete Real u,y,z;
```

2nd International Workshop on Equation-Based Object-Oriented Languages and Tools. July 8, 2008, Paphos, Cyprus.

EOOLT 2008 website:
http://www.eoolt.org/2008/

```
equation
  when sample(0,1) then
     u=2*y;
     y=pre(z);
    z=pre(z)+u;
  end when;
```

Both codes are correct and should yield identical simulation codes on any reasonable compiler/simulator. In our proposal, the two codes are considered semantically equivalent. The former being transformed to the latter in a first phase of compilation by the compiler.

To see where the activation inheritance comes into play in this example, note that the equations defining `u`, and `y` are not "explicitly activated"; we say then that these equations inherit their activations. Such an equation is activated only if at least one of the variables on which it depends may potentially change value. This of course is very similar to a data-flow behavior. In this particular example, the equation defining `u` depended on z, that is why it was moved to the `when` clause. The equation defining `y` depended on `u`, now in the `when` clause, so it was also moved inside the `when` clause.

This may seem like a minor issue however it allows us to present useful extensions as we shall see later.

3. Presence of Discrete States

To show that the activation inheritance mechanism cannot be assimilated with code optimization, consider the following code:

```
equation
  u=2*y;
  y=pre(z);
  j=pre(j)+y;
  when sample(0,1) then
    z=pre(z)+u;
  end when;
```

This code is not currently legal in Modelica whether `j` is defined discrete or not. We are not going to worry about declarations as discrete; from our point of view, this is redundant information that can only be useful for checking the model. Indeed, the placement of the variable in the equation section is enough to determine its discrete nature.

Now consider the effect of the transformation we had proposed on the previous example on this new one:

```
equation
  when sample(0,1) then
    u=2*y;
    y=pre(z);
    j=pre(j)+y;
    z=pre(z)+u;
  end when;
```

This is a valid Modelica code. As we propose that the pre and post versions be considered semantically equivalent, then the first version should be considered valid as well. Note that in this example we do not merely try to propose a code optimization strategy. Unlike memory-less equations where too much activation does not affect the simulation outcome, the activation instants of the equation `j=pre(j)+y;` must be uniquely specified in order to obtain an unambiguous model. Clearly without a rigorous definition of the concept of activation inheritance, the extension that we propose cannot work.

4. Multiple Inheritance

Let us now examine the situation where an equation depends on more that one variable updated in different `when` clauses:

```
 equation
   a=k+m;
when sample(0,1) then
   k=pre(k)+1;
end when;
when sample(.5,1) then
  m=pre(m)-1;
end when;
```

In this case, the variable `a` can potentially change value at both `sample(0,1)` and `sample(.5,1)`. So the transformation is carried out as follows:

```
equation
when sample(0,1) then
   k=pre(k)+1;
   a=k+m;
end when;
when sample(.5,1) then
  m=pre(m)-1;
  a=k+m;
end when;
```

This is an easy case because the two `when` clauses are primary (see [5]), i.e. asynchronous. The situation is a bit more complicated when this is not the case. Consider the following example:

```
 equation
    z=pre(z)+a+b;
  when sample(0,1) then
    a=pre(a)+1;
  end when;
  when a>3 then
    b=a;
  end when;
```

Clearly, we would not obtain a correct model if we placed the definition of `z` in both `when` clauses. In particular `z` would be incremented wrongly twice by `a+b` when `a` crosses 3. The fundamental reason is that in this case the two `when` clauses are not asynchronous. The activations updating `a` and `b` when `a` crosses 3 are in fact synchronous (same activation). In [5], we showed how the secondary `when` clauses are removed in the process of compilation. Adding the inheritance mechanism, in our case this yields the following code:

```
when sample(0,1) then
  a=pre(a)+1;
  if edge(a>3) then
    b=a;
```

```
  end if ;
  z=pre(z)+a+b;
end when;
```

The situation is of course more complex in the presence of multiple primary and secondary when clauses but the inheritance rules are clearly defined and the compiler can treat them in an unambiguous manner.

5. Always Active in Modelica

So far we have considered only cases where the model contained no continuous-time dynamics. In general, the main component of a Modelica model runs in continuous-time and the expressions within an `equation` section, but outside any `when` clause, have been associated systematically to the "always active" components in Scicos. It is for this reason that in our Modelica compiler, we replicate all these expressions inside the body of all `when` statements. For example in the following example:

```
equation
   a=f(k+3);
   b=sin(time);
  when sample(0,1) then
    k=pre(k)+1;
  end when;
  when sample(.5,1) then
    m=pre(m)-1;
  end when;
```

we obtain

```
equation
  when continuous then
   a=f(k+3);
   b=sin(time);
  end when;
  when sample(0,1) then
   k=pre(k)+1;
   a=f(k+3);
   b=sin(time);
  end when;
  when sample(.5,1) then
   a=f(k+3);
   b=sin(time);
   m=pre(m)-1;
  end when;
```

This represents of course the result of the brut force application of the systematic transformations; the final code can be much smaller after optimization. The `when continuous` clause corresponds essentially to the pure continuous-time dynamics, the part that would be left to the DAE solver, see [5] for more details.

This treatment is not really consistent with the inheritance mechanism we are proposing here. If we strictly apply the inheritance mechanism described previously, there wouldn't be any `continuous` clause and the expression `a=f(k+3)` would appear only in the first `when` clause, as it should, but `b=sin(time);` appears nowhere! Clearly there is a distinction between the two equations. The inheritance mechanism can work only if somehow we specify that `b=sin(time);` is always active. This can be done by introducing a new `when` clause:

```
 equation
    a=f(k+3);
when always_active then
  b=sin(time);
end when;
when sample(0,1) then
  k=pre(k)+1;
end when;
when sample(.5,1) then
  m=pre(m)-1;
end when;
```

This way we explicitly specify that `b=sin(time);` should be evaluated all the time, but the activation of `a=f(k+3);` is inherited. In particular for discrete variables, the compiler replicates this expression only in `when` statements in which "k" is computed (appears on the left hand side of an equation):

```
equation
when always_active then
   b=sin(time);
end when;
when sample(0,1) then
   k=pre(k)+1;
   a=f(k+3);
end when;
when sample(.5,1) then
  m=pre(m)-1;
end when;
```

The equations in the always active section are then copied in all `when` clauses. Of course if we had `a=f(k+m);` then the expression would be copied in both discrete `when` statements above (multiple inheritance).

The use of the `always_active` clause corresponds exactly to the block property "always active" in Scicos. To require such a clause however would break backward compatibility in Modelica. A possible solution would be to note that in almost all cases, always activation, which is associated with variables evolving continuously in time, originates from the use of the built-in variable `time` or the operator `der()`. It is then possible to implement a pre-compiler filter that scans over the flat Modelica model and place the equations in which `der()` and `time` appear inside an `always_active` clause. The rest of the continuous variables follow naturally by applying the inheritance rule.

6. Applications

There are three closely related issues that need to be carefully studied in Modelica: separate compilation of a part of a model, code generation for controller applications and general external functions allowing for internal states. The heart of all three problems is the ability to isolate any part of a diagram, and under certain conditions, generate a C code providing the overall behavior of this part with a canonical API.

The simplest situation is when this part of the diagram contains only memory-less event-less dynamics in which

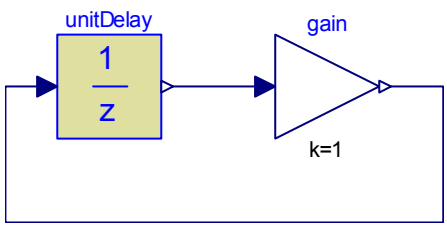
case the C code would be an external function as it already exists in Modelica.

The situation becomes more interesting when the part of the diagram we are considering contains discrete states. Consider for example a classical controller configuration where a continuous-time plant is controlled by a discrete-time, single frequency controller. By isolating the controller part, we end of with models driven synchronously. Based on the current interpretation of the Modelica specification by Dymola, these discrete blocks are each driven by a `sample` event generator; the use of an identical period and phase in the sample statements provides the synchronization. We have tried to show in [5] that this form of implicit synchronization detected only during simulation puts useless constraints on the model and the compiler, and it should be avoided. An alternative solution was proposed in [6] where the keyword `sample` was given a macro status allowing for proper synchronization via a clock calculus similar to that used in Simulink and SampleCLK blocks in Scicos.

Another technique to synchronize the discrete blocks would be the explicit use of events as signals. For that, we introduced a new type called Event in [4]. In this approach, every discrete block would have an event input port (called activation input port in Scicos) driving its activations. This way, a single `sample` event generator can be hooked up to all discrete blocks driving them synchronously.

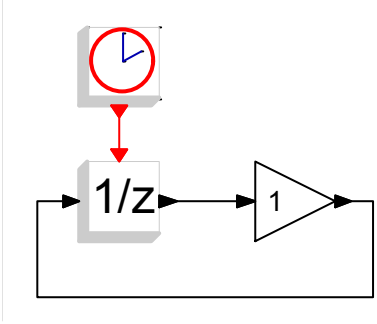
The inheritance mechanism provides an alternative solution to the synchronization problem. In most cases, there are already blocks in the discrete part that work almost on the inheritance mechanism. For example a simple Gain block used in this part is not driven by any event: the corresponding equations are placed in the equation section. We have seen that this is pretty much an inheritance mechanism relying on compiler optimization. With the inheritance mechanism we have introduced, models with states such as discrete-state space blocks can also be driven via inheritance. In fact it almost suffices to remove the `when, end when` statements from the existing models. In an actual implementation, it suffices to drive a first block (usually the analog to digital converter block defining the boundary between plant and controller) with the proper `sample` event generator. The rest of the blocks then follow through by the inheritance mechanism synchronously as they should. The advantage of the inheritance solution is that the discrete part need not even run on a fixed frequency clock.

Let us examine these ideas on actual examples. In the following Modelica diagram, we have a purely discrete single frequency system.



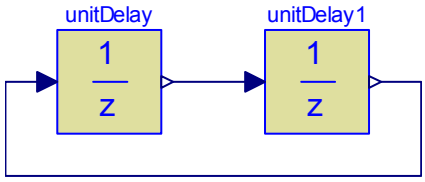
The Memory block is driven by a `sample` event generator (equations are all within a `when` clause) but the Gain block contributes an equation to the equation section, outside the `when` clause. The inheritance mechanism would move the Gain equation into the `when` clause yielding a purely discrete system (a system where all variables are discrete).

The Scicos counterpart of this Modelica model is

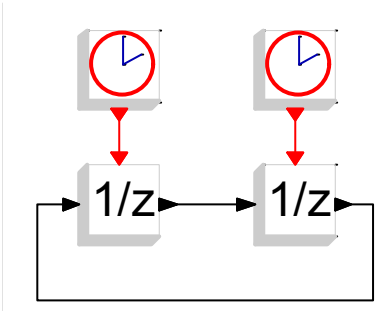


where the only difference is the explicit presence of an event clock (generator). Note that the event clock needs not have fixed frequency in this case.

Consider now the case where there are two discrete blocks in a diagram:

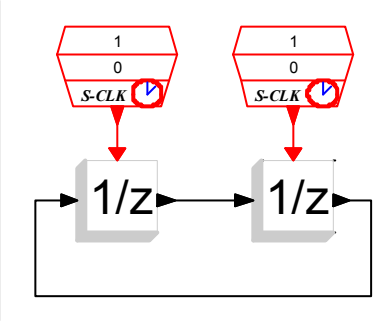


If each block has its own `sample` event generator, as it is the case today in the discrete Modelica library, the Scicos counterpart would be

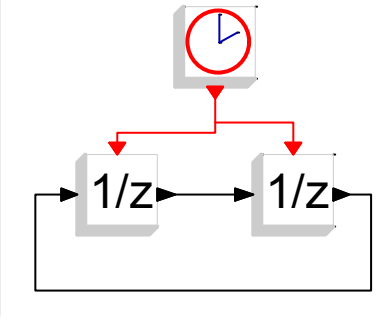


This diagram is not synchronous in Scicos even if the two event generators have identical periods. The simulation result for this diagram would not be predictable in Scicos

and we think it should not be in Modelica either unless a new definition is associated with the keyword sample as we have proposed in [6] where the keyword `sample` was given a macro status allowing for proper synchronization via a clock calculus similar to the one used in Simulink and SampleCLK blocks in Scicos. The corresponding Scicos diagram using SampleCLK blocks would be the following:



In this case, the Scicos compiler starts by performing the necessary computations on SampleCLK blocks' periods and offsets to find the unique slowest clock that by subsampling can replace the activations of these blocks. In this case the computation is trivial because the two blocks have identical periods; the equivalent diagram after this first phase is the following:



By adopting the proposed interpretation for the Modelica keyword `sample` in [6], Modelica compiler would function similarly.

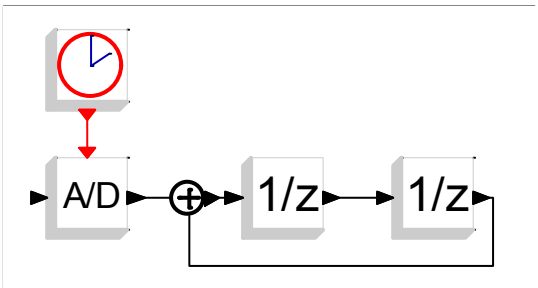
In Scicos, it is also possible to construct directly this latter diagram, i.e., to drive both memory blocks by the same event block. This is not possible to do in Modelica unless we adopt the Event type suggested in [4]. In that case we would be able to express the Memory block as follows:

```
model Memory
  input Event e1;
  output Real y;
  input Real u;
  discrete Real z;
  equation
  when e1 then
    z=u;
    y=pre(z);
  end when;
end Memory;
```

We would also need an event generator block, something resembling the following:

```
    Event e(start=t0) ;
    discrete Integer k(start=0) ;
  equation
    when pre(e)  then
      k=pre(k)+1 ;
      e=k*T+t0 ;
    end when ;
```

In Scicos, a Memory block, like any other block, can also be defined without an activation input port in which case it inherits its activation from its regular input. To see how this works, consider a slightly more complicated diagram that corresponds to a simple controller part of a Plant/controller diagram:



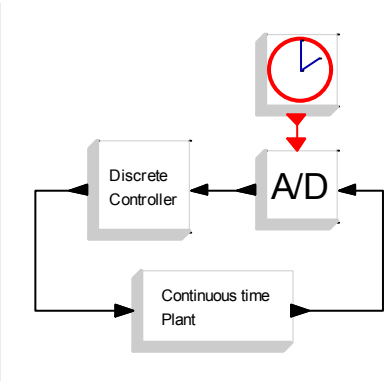
In this case the Memory blocks (and the sum block as well) are activated by inheritance. The Modelica counterpart of this Memory block would be:

```
model Memory
    output Real y;
    input Real u;
    discrete Real z;
  equation
     z=u;
     y=pre(z);
end Memory;
```

This means in particular that in a controller model, by defining all discrete blocks "inheriting", i.e., without any built-in `when sample` clauses, it suffices to drive only the interface between the plant and the controller with an event generator. This interface would be a model such as the following:

```
model AD
    discrete output Real y;
    input Real u;
    parameter Real T=1;
  equation
     when sample(0,T) then
        y=u;
     end when;
end AD;
```

It is the activation in the `when sample` clause of this block that is inherited by the controller part which would then run synchronously. The following is the general picture of this setup:



The continuous-time plant does not inherit any activation because it is already always active. This configuration is a classical representation used by control engineers. Note that no D/A block is necessary here because the output of discrete blocks hold their outputs constant until their next activation, nonetheless such a block can be used to clearly indicate the boundaries of the controller.

7. Conclusion

We have proposed an activation inheritance mechanism in Modelica that can be used to synchronize a set of discrete blocks, for example the controller part of a standard plant/controller configuration. This extension is one possible way to solve the synchronization issues in the discrete block library.

With this new extension, we have three different solutions to the synchronization problem: the use of macro `sample` clauses (similar to SampleCLK in Scicos) in different discrete blocks, the use of event input/outputs allowing the activation of multiple blocks via a single event generator, and activation inheritance. These solutions are complementary and can very well co-exist in Modelica; they do in Scicos.

The macro `sample` solution is limited to periodic systems. It is particularly useful for modeling synchronous multi-frequency systems. Having access to the `sample` parameters in each block allows also the computation of blocks parameters as a function of `sample` parameters. For example, the parameters of a discrete linear system block may be computed as a function of the sampling period using the exact discretization method.

The use of the Event type provides full control over the activation and synchronization of discrete blocks and is a key element needed for separate compilation in Modelica. It allows synchronous, asynchronous, single frequency, multi-frequency and sporadic activations.

Finally inheritance activation provides a modeling facility that avoids the need for explicitly defining the activation of each block. The use of inheriting blocks provides a viable alternative in the single frequency case to the other two solutions.

Acknowledgments

This work has been supported by the ANR project SIMPA2-C6E2.

References

[1] M. Otter, H. Elmqvist, S. E. Mattsson, “Hybrid Modeling in Modelica based on the Synchronous Data Flow Principle”, CACSD’99, Aug; 1999, Hawaii, USA.

[2] Modelica Association, Modelica® - A Unified Object-Oriented Language for Physical Systems Modeling, www.modelica.org/documents/ModelicaSpec30.

[3] A. Benveniste, P. Le Guernic, C. Jacquemot.,"Synchronous programming with events and relations : the Signal language and its semantics", Science of Computer Programming, 16, 1991, p. 103-149.

[4] R. Nikoukhah, “Extensions to Modelica for efficient code generation and separate compilation”, in Proc. EOOLT Workshop at ECOOP’07, Berlin, 2007.

[5] R. Nikoukhah, “Hybrid dynamics in Modelica: Should all events be considered synchronous”, in Proc. EOOLT Workshop at ECOOP’07, Berlin, 2007.

[6] R. Nikoukhah, S. Furic, “Synchronous and Asynchronous Events in Modelica: Proposal for an Improved Hybrid Model”, in Proc. Modelica Conference, Bielefeld, 2008.

[7] P. Fritzson - “Principles of Object-Oriented Modeling and Simulation with Modelica 2.1”, Wiley-IEEE Press, 2003.

[8] S. L. Campbell, Jean-Philippe Chancelier and Ramine Nikoukhah, “Modeling and Simulation in Scilab/Scicos”, Springer, 2005.

Selection of Variables in Initialization of Modelica Models

Masoud Najafi

INRIA-Rocquencourt, Domaine de Voluceau, BP 105, 78153, Le Chesnay, France
masoud.najafi@inria.fr

Abstract

In Scicos, a graphical user interface (GUI) has been developed for the initialization of Modelica models. The GUI allows the user to fix/relax variables and parameters of the model as well as change their initial/guess values. The output of the initialization GUI is a pure algebraic system of equations which is solved by a numerical solver. Once the algebraic equations solved, the initial values of the variables are used for the simulation of the Modelica model. When the number of variables of the model is relatively small, the user can identify the variables that can be fixed and can provide the guess values of the variables. But, this task is not straightforward as the number of variables increases. In this paper, we present the way the incidence matrix associated with the equations of the system can be exploited to help the user to select variables to be fixed and to set guess values of the variables during the initialization phase.

Keywords Modelica, initialization, coupling algorithm, numerical solver, Scicos

1. Introduction

Scicos[1] is a free and open source simulation software used for modeling and simulation of hybrid dynamical systems [3, 4]. Scicos is a toolbox of Scilab[2] which is also free and open-source and used for scientific computing. For many applications, the Scilab/Scicos environment provides an open-source alternative to Matlab/Simulink. Scicos includes a graphical editor for constructing models by interconnecting blocks, representing predefined or user defined functions, a compiler, a simulator, and code generation facilities. A Scicos diagram is composed of blocks and connection links. A block corresponds to an operation and by interconnecting blocks through links, we can construct a model, or an algorithm. The Scicos blocks represent elementary systems that can be used as building blocks. They can have several inputs and outputs, continuous-time states, discrete-time states, zero-crossing functions, etc. New custom blocks can be constructed by the user in C and Scilab languages. In order to get an idea of what a simple Scicos hybrid models looks like, a model of a control system has been implemented in Scicos and shown in Figure 1.

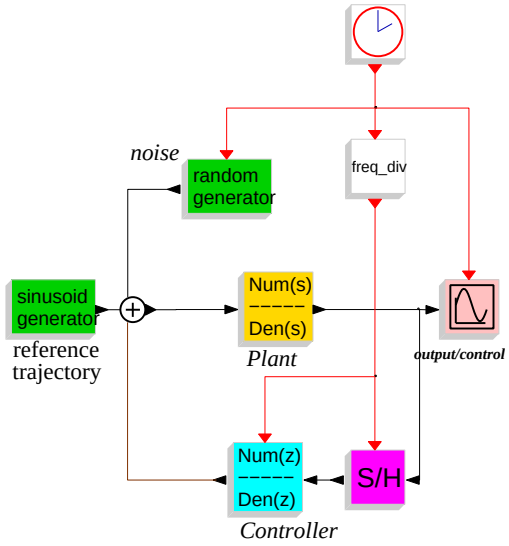


Figure 1. Model of a control system in Scicos

Besides causal or standard blocks, Scicos supports a subset of the Modelica[3] language [7]. The diagram in Figure 2 shows the way a simple DC-DC Buck converter has been modeled in Scicos. The electrical components are modeled with Modelica while the blocks that are used to control the On/Off switch are modeled in standard Scicos.

The Modelica compiler used in Scicos has been developed in the SIMPA[4] project. Recently the ANR[5]/RNTL SIMPA2 project has been launched to develop a more complete Modelica compiler. The main objectives of this project are to extend the Modelica compiler resulted from the SIMPA project to fully support inheritance and hybrid systems, to give the possibility to solve inverse problems

[1] www.scicos.org

[2] www.scilab.org

[3] www.modelica.org

[4] Simulation pour le Procédé et l'Automatique

[5] French National Research Agency

2nd International Workshop on Equation-Based Object-Oriented Languages and Tools. July 8, 2008, Paphos, Cyprus.

EOOLT 2008 website:
http://www.eoolt.org/2008/

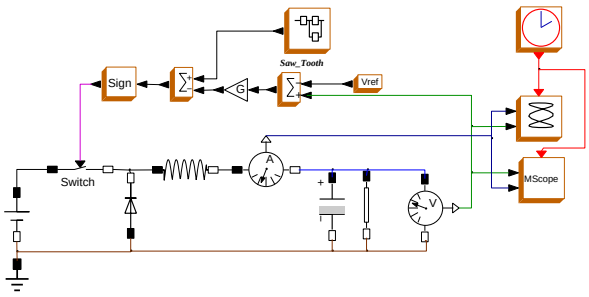


Figure 2. Model of a DC-DC Buck converter in Scicos using Modelica components

by model inversion for static and dynamic systems, and to improve initialization of Modelica models.

An important difficulty when simulating a large Modelica model is the initialization of the model. In fact, a model can be simulated only if it is initialized correctly. The reason lies in the fact that a DAE (Differential-Algebraic Equation) resulting from a Modelica program can be simulated only if the initial values of all the variables as well as their derivatives are known and are consistent.

A DAE associated with a Modelica model can be expressed as

$$0 = F(x', x, y, p) \tag{1}$$

where x, x', y, p are the vector of differential variables of size N_d, derivative of differential variables of size N_d, algebraic variables of size N_a, and model parameters of size N_p, respectively. $F(.)$ is a nonlinear vector function of size $(N_d + N_a)$. Since, the Modelica compiler of Scicos supports index-1 DAEs [1, 2], in this paper we limit ourselves to this class of DAEs.

In Scicos, in order to facilitate the model initialization, the initialization phase and the simulation phase have been separated and two different codes are generated for each phase: The initialization code (an algebraic equation) and the simulation code (a DAE).

In the Initialization phase, x' is considered as an algebraic variable (*i.e.*, dx) and then an algebraic equation is solved. Modelica parameters p are considered as constants unless they are relaxed by the user. There are $(N_d + N_a)$ equations and $(2N_d + N_a + N_p)$ variables and parameters that can be considered as unknowns. In order to have a square problem solvable by the numerical solver, (N_p+N_d) variables/parameters must be fixed. The values of x and p are often fixed and given by the user and the values of dx and y are computed. But the user is free to fix or relax any of variables and parameters. For example, in order to initialize a model at the equilibrium state, dx is fixed and set to zero whereas x is relaxed to be computed. Another example is parameter sizing where the value of a parameter is computed as a function of a fixed variable. In this case, the parameter p is relaxed and the variable x is fixed.

In the simulation phase, the values obtained for x, dx, y, p are used for starting the simulation. During the simulation, the value of p (model parameters) does not change.

In Modelica, the `start` keyword can be used to set the start values of the variables. The start values of derivatives of the variables can be given within the `initial equation` section. For small programs, this method can easily be used but as the program size grows, it becomes difficult to set start values and change the `fixed` attribute of variables or parameters directly in the Modelica program; initialization via modifying the Modelica model is specially difficult for models with multiple levels of inheritance; the visualization and fixing and relaxing of the variables and the parameters are not easy. Furthermore, the user often needs to have a model with several initialization scenarios. For each scenario a copy of the model should be saved.

In Scicos, a GUI has been developed to help the user to initialize the Modelica models. In this GUI, the user can easily change the attributes of the variables and the parameters such as `initial/guess value`, `max`, `min`, `nominal`, etc. Furthermore, it is possible to indicate whether a variable, the derivative of a variable, and a parameter must be fixed or relaxed in the initialization phase.

In the following sections, the initialization methodology for Modelica models and the initialization GUI features will be presented.

2. Initialization and Simulation of Modelica Models

The flowchart in Figure 3 shows how initialization and simulation of Modelica models are carried out in Scicos. The first step in both tasks is removing inheritances. This provides access to all variables and generates a flat model. The flat model is used to generate the initialization and the simulation codes. Note that the initialization data used for starting the simulation is passed to the simulation part by means of an XML file containing all initial values.

In Scicos, three external applications are used in initialization and simulation: `Translator`, `XML2Modelica`, and `ModelicaC`.

`Translator` is used for three purposes:

- Modelica Front-end compiler for the simulation: when called with appropriate options, `Translator` generates a flat Modelica program. For that, `Translator` verifies the syntax and semantics of the Modelica program, applies inheritance rules, generates equations for `connect` expressions, expands `for` loops, handles predefined functions and operators, performs the implicit type conversion, etc. The generated flat model contains all the variables, the derivatives of differential variables, and the parameters defined with attribute `fixed=false`. Constants and parameters with the attribute `fixed=true` are replaced by their numerical values.
- Modelica Front-end for initialization: when called with appropriate options, `Translator` generates a flat Modelica program containing the variables and the parameters defined with attribute `fixed=false`. The derivatives of the variables are replaced by algebraic

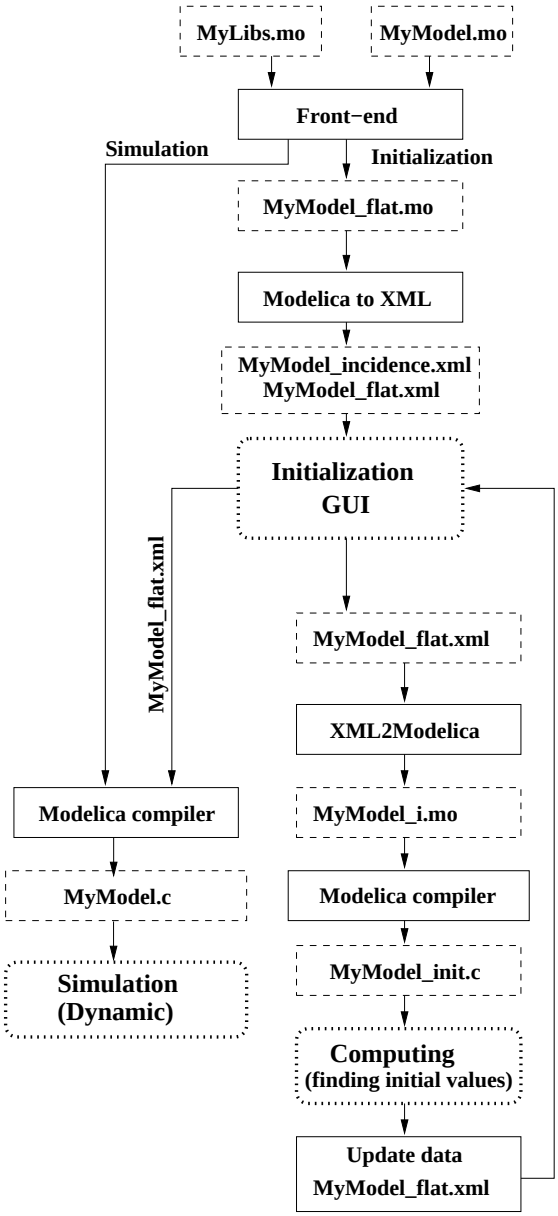


Figure 3. Initialization flowchart in Scicos

variables. Furthermore, the flat code contains the equations defined in the `initial equation` section in the Modelica programs. Constants and parameters with the attribute `fixed=true` are replaced by their numerical values.

- XML generator: when called with -xml option, Translator generates an XML file from a flat Modelica model. The generated XML file contains all the information in the flat model.

Once the XML file generated, the user can change variable and parameter attributes in the XML file with the help of the GUI. The modified XML file have to be reconverted into a Modelica program to be compiled and initialized. This is done by `XML2Modelica`.

`ModelicaC`, which is a compiler for the subset of the Modelica language, compiles a flat Modelica model and generates a C program for the Scicos target. The main features of the compiler are the simplification of the Modelica models and the generation of the C program ready for simulation. It supports zero-crossing and discontinuity handling and provides the analytical Jacobian of the model. It does not support DAEs with index greater than one. Another important feature of the Modelica compiler is the possibility of setting the maximum number of simplification carried out during the code generation phase. Thus, the compiler can be called to generate a C code with no simplification or a C code with as much simplification as possible. This is an important feature for the debugging of the model.

A new feature of `ModelicaC` is generating the incidence matrix. When a C code is generated, the corresponding incidence matrix is generated in an XML file. The incidence matrix is used by the initialization GUI to help the user.

As shown in Figure 3, once the user requests the initialization of the Modelica model, the Modelica front-end generates a flat Modelica model as well as its corresponding XML file. The XML file is then used in the initialization GUI. In the GUI, the user can change the variable and parameter attributes defined in the XML file. The modified XML file is then translated back to a Modelica program. The Modelica program is compiled with the Modelica compiler and a C program is generated. The C program is used by the Scicos simulator to compute the value of unknowns. Once the initialization finished, whether succeeded or failed, the XML file is updated with the most recent results. The GUI automatically reloads and displays the results. The user can then decide whether the simulation can be started or not.

In order to simulate the Modelica model, similar to the model initialization, a flat model is generated. Then, the Modelica compiler simplify the model and generates the simulation code. The generated code is simulated by a numerical solver. The initial values, needed to start the simulation, are read directly from the XML file. The end result of the simulation can also be saved in an another XML file to be used as a starting point for another simulation.

3. Initialization GUI

In Scicos, a GUI can be used for the initialization of the Modelica models. Figure 4 illustrates a screen shot of the GUI corresponding to the Modelica parts of the Scicos diagram of Figure 2. In this GUI, the Modelica model is displayed in the hierarchical from, as shown in Figure 4. Main branches of the tree represent components in the Modelica model. Subbranches are connectors, partial models, etc. If the user clicks on a branch, the variables and parameters defined in that branch are displayed and the user can modify their attributes. In the following subsections, some main features of the GUI will be presented.

3.1 Variable/Parameter Attributes

Any variable/parameter has several attributes which are either imported directly from the Modelica model such as

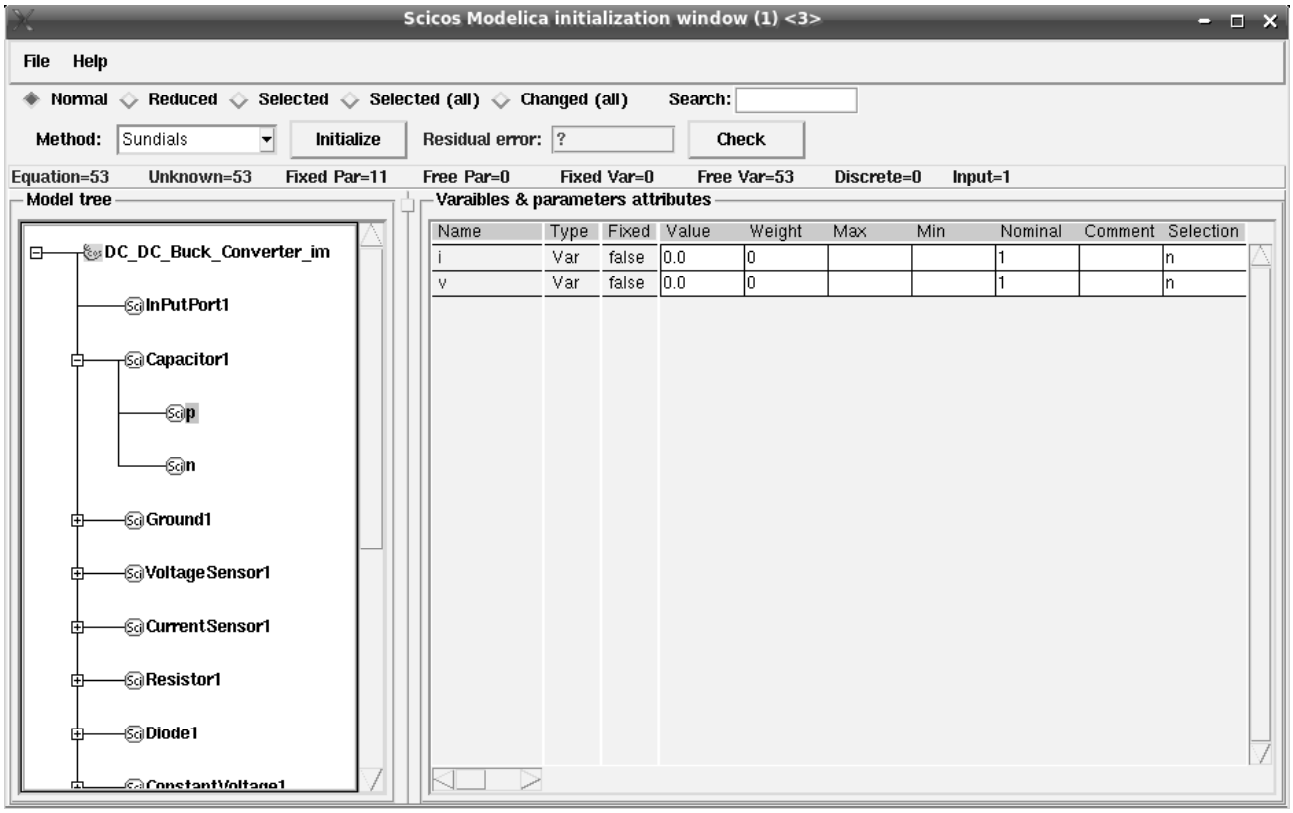


Figure 4. Screen shot of the initialization GUI in Scicos for the electrical circuit of Figure 2

`name`, `type`, `fixed` etc. or defined and used by the GUI *i.e.*, `id` and `selection`.

- `name` is the name of the variable/parameter used in the Modelica program. The user cannot change this attribute in the GUI.
- `id` is an identification of the variable/parameter in the flat Modelica program. The user cannot change this attribute in the GUI.
- `type` indicates whether the original type has been `parameter` or `variable` in the Modelica program. The user cannot change this attribute in the GUI.
- `fixed` represents the value of the `'fixed'` attribute of the variable/parameter in the Modelica program. The user cannot change this attribute in the GUI.
- `weight` is the confidence factor. In the current version of Scicos, it takes either values 0 or 1. `weight=0` corresponds to the `fixed=false` in Modelica whereas `weight=1` corresponds to `fixed=true`. The default value of `weight` for the parameters and differential variables is one, whereas for the algebraic variables and the derivatives of differential variables (converted to variables) is zero.
- `value` is the value of the variable/parameter. If the `weight=1`, the given `value` is considered as the final value and it does not change in the initialization. If `weight=0`, the given `value` is considered as a guess value. If the user does not provide any value, it is automatically set to zero. The user can modify this value in the GUI.
- `selection` is used to mark the variables and parameters. This information will be used by the GUI for selective display of variables/parameters and to influence the Modelica compiler in the model simplification phase.

Note that if the user sets the `weight` attribute of a variable to one, it will be considered as a constant and in the initialization phase it will be replaced by its numerical value. On the other hand, if the user sets the `weight` attribute of a parameter to zero, the parameter will be considered as an unknown and its value will be computed in the initialization phase. This is in particular useful when the user tries to find a parameter value as a function of a variable in the Modelica model.

3.2 Display Modes

Accessing to variables and parameters of the model becomes easier, if different display modes of the GUI are used:

- **Normal** mode is the default display mode. Clicking on each branch of the model tree, the user can visualize/modify the variables/parameters defined in that part of the Modelica model.
- **Reduced** mode is used to display the variables of the simplified model. When the user pushes the initialization button, the flat Modelica model is compiled and a simplified model is generated. In this display mode,

only the remaining variables are displayed. This display mode is in particular useful when the numerical solver cannot converge and the user should help the solver either by influencing the compiler to eliminate the undesirable variables or by giving more accurate guess values.

- **Selected** mode is used to display only the marked variables and parameters of the active branch. A variable or parameter can be marked by putting `'y'` in its selection field in the GUI. By default, all parameters, all differential variables and all algebraic variables whose start values are given are marked. Marking is useful in particular when a branch has many variables/parameters whereas the user is interested in a few ones. In this display mode, unmarked variables/parameters are not shown.
- **Selected (all)** mode is used to display all marked variables and parameters of the Modelica model.
- **Changed** mode is used to display the variables and the parameters whose `weight` attributes have been changed, such as the relaxed parameters.

3.3 Initialization Methods

Once the user modified the attributes of the variables and the parameters, the initialization process can be started by clicking on the "Initialize" button. The initialization consists of calling a numerical solver to solve the final algebraic equation. There are several algebraic solvers available in Scicos such as `Sundials` and `Fsolve` [8, 9, 10].

Once the solver finished the initialization, the obtained results, either successful or not, are put back into the XML file and new values are displayed in the GUI. If the result is not satisfactory, the user can either select another initialization method or help the solver by giving initial values more accurately. This try and error can be continued until satisfactory initialization results are obtained. Then, the simulation can be started.

4. Problems in Variable Fixing and Variable Selection

The initialization of DAE (1) can be formulated as the following algebraic problem

$$0 = F(dx_0, x_0, y_0, p_0) \tag{2}$$

where x_0, dx_0, and y_0 are solutions or the initial values of differential variables, derivative of differential variables, algebraic variables, and parameter values, respectively. The degree of freedom of the equation (2) is $N_d + N_p$, therefore the user should fix $N_d + N_p$ variables or parameters and let the solver find the values of the remaining $N_d + N_a$ unknowns.

Fixing the variables/parameters and giving the start values of the relaxed variables/parameters are essential in the initialization of models. But they are not easy and straightforward for large models. In the next subsections the way these problems are handled in Scicos will be explained.

4.1 Fixing the Variables

Consider the following equation set, composed of two equations and three unknowns.

$$F : \begin{cases} 0 = f(x) \\ 0 = g(x, y, z) \end{cases}$$

Since the degree of freedom is one, the user should provide and fix the value of a variable. But, it is clear that x cannot be fixed, because its value is imposed by the first equation. In this case, the GUI should prevent the user from fixing x.

Consider the next set of equations composed of three equations and five unknowns.

$$F : \begin{cases} 0 = f(x, u) \\ 0 = g(x, z) \\ 0 = h(x, y, z, v) \end{cases} \tag{3}$$

Although the degree of freedom is two, the user cannot fix (u, z), (x, z), or (x, u) at the same time. In general, it is not easy to identify the set of variables that can be fixed. This is in particular important when the number of equations increases. In this case, if the user tries to fix an inadmissible variable, the GUI should raise an error message and prevent the user from fixing the variable.

This problem can be solved using the incidence matrix of the Modelica model. For example, this is the incidence matrix of (3):

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Fixing u and z means removing u and z from the equations which results in the following equation set and the incidence matrix.

$$F : \begin{cases} 0 = f(x, u_0) \\ 0 = g(x, z_0) \\ 0 = h(x, y, z_0, v) \end{cases} \quad \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

Although, there are three unknowns and three equations, the incidence matrix is not structurally full rank. This means that u and z cannot be fixed at the same time.

Computing the structurally rank of the incidence matrix is a straightforward way to determine if the user is allowed to fix variables or parameters of the model. Since the incidence matrix is very often large and sparse in practical models, we should use special methods for sparse matrices. In the GUI, a `maximum matching` method (also called a maximum transversal method) is used to compute the structural rank of the incidence matrix. The maximum matching method is a permutation of the matrix so that its k^{th} diagonal is zero-free and $|k|$ is uniquely minimized. With this method, the structural rank of the matrix is the number of non-zero elements of the matrix diagonal [6].

When the user tries to fix a variable or a parameter, the initialization GUI computes the new structural rank of the incidence matrix. If the fixing operation lowers the rank, an error message will raised and the modification will be inhibited.

4.2 Selection of Variables to Be Eliminated

Another recurrent problem in solving algebraic equations is the convergence failure of the solver. Newton methods are convergent if the initial guess values of unknowns are not too far from the solution. So, the user should provide reasonable initial guess values. If the problem size is small and the user knows the nominal values of the unknowns, the user can provide the guess values. But in large models, it is nearly impossible to give all guess values. In medium size Modelica models, we usually end up with models with many variables whose start values are not specified by the user. In this case, their initial guess values are automatically set to zero which is not often a good choice. Furthermore, many variables of a model are redundant and the user does not know for which ones the initial guess should be given. This often happens with variables linked by the `connect` operator in Modelica. Suppose that two Modelica components are connected via a connector, *e.g.*,

```
connect (Block1.x, Block2.y);
```

During the model simplification, the compiler may eliminate either `Block1.x` or `Block1.y`. Even if the user knows the guess values of both, it is not reasonable to ask the user to provide them. Since the user has no influence on the compiler's variable selection, this may cause a problem in solving the initialization equation. Consider, *e.g.*, the following situation.

$$F : \begin{cases} 0 &= \dfrac{x-3}{(x-3)^2+1} - 0.1 \\ 0 &= x - y \end{cases}$$

Here, if the user sets the initial guess of y to 10 and leaves the guess value of x unspecified *i.e.*, $x = 0$, although $y = 10$ is close to the solution, the Newton's method will likely fail. The reason is that the solver ignores the initial value of y and uses that of x. In fact, there is no way to tell the solver the guess value which is "more" correct than the others.

The solution is to formally simplify the equations by eliminating the variables whose guess-values are not given, by replacing them with the variables having given guess-values. For that, in the initialization GUI, variables with known guess-values are marked and the Modelica compiler is told to eliminate the unmarked variables. The user, of course, can modify the list of these marked variables.

The compiler tries to eliminate the variables as much as possible, but a problem may arise when the compiler fails to eliminate all of unmarked variables. Since, the simulator sets their guess-value to zero, the original problem still persists. In this case, the user should be asked to provide the guess-value of the remaining variables. But, usually the user has no idea about the nominal values of the remaining variables or even does not know the physical interpretation of them. As an example, consider the following set of equations for which no guess-values are given.

$$F : \begin{cases} 0 &= f(x) \\ 0 &= x - y \end{cases}$$

Suppose that the compiler eliminates y, but the user does not know the start value of x while y has a physical interpretation and its nominal value can be given. In this case, the initialization GUI should propose to the user all variables that can replace x, *i.e.*, y.

Proposing alternative variables for formal simplification is done in the initialization GUI. In the next sections, it will be shown the way these problems can be handled by the use of the incidence matrix of the model. This is done using the maximum flow algorithms.

5. Maximum Flow Problem

The maximum flow problem is to find the maximum feasible flow through a single-source, single-sink flow network [5]. The maximum flow problem can be seen as a special case of more complex network flow problems. A directed graph or digraph G is an ordered pair $G := (V, A)$ with

- V is the set of vertices or nodes,
- A is the set of ordered pairs of vertices, called directed edges or arcs.

An edge $e = (u, v)$ is considered to be directed from u to v; v is called the head and u is called the tail of the edge; v is said to be a direct successor of u, and u is said to be a direct predecessor of v. The edge (v, u) is called the inverted edge of (u, v).

Given a directed graph $G(V, E)$, where each edge u, v has a capacity $c(u, v)$, the maximal flow f from the source s to the sink t should be found. There are many ways of solving this problem, such as linear programming, Ford-Fulkerson algorithm, Dinitz blocking flow algorithm, etc [12, 11].

5.1 Ford-Fulkerson Algorithm

The *Ford-Fulkerson algorithm* computes the maximum flow in a flow network. The name "Ford-Fulkerson" is often also used for the Edmonds-Karp algorithm, which is a specialization of Ford-Fulkerson. The idea behind the algorithm is very simple: as long as there is a path from the source to the sink, with available capacity on all edges in the path, we send flow along one of these paths. Then we find another path, and so on. A path with available capacity is called an augmenting path.

Algorithm: Consider a graph $G(V, E)$, with capacity $c(u, v)$ and flow $f(u, v) = 0$ for the edge from u to v. We want to find the maximum flow from the source s to the sink t. After every step in the algorithm the following is maintained:

- $f(u, v) \leq c(u, v)$. The flow from u to v does not exceed the capacity.
- $f(u, v) = -f(v, u)$. Maintain the net flow between u and v. If in reality a units are going from u to v, and b units from v to u, maintain $f(u, v) = a - b$ and $f(v, u) = b - a$.
- $\sum_v f(u, v) = 0 \Longleftrightarrow f_{in}(u) = f_{out}(u)$ for all nodes u, except s and t. The amount of flow into a node equals the flow out of the node.

This means that the flow through the network is a legal flow after each round of the algorithm. We define the residual network $G_f(V, E_f)$ to be the network with capacity $c_f(u, v) = c(u, v) - f(u, v)$ and no flow. Notice that it is not certain that $E = E_f$, as sending flow on u, v might close u, v (it is saturated), but open a new edge v, u in the residual network.

1. $f(u, v) \leftarrow 0$ for all edges (u, v)
2. While there is a path p from s to t in G_f, such that $c_f(u, v) > 0$ for all edges $(u, v) \in p$:
   (a) Find $c_f(p) = \min\{c_f(u, v)|(u, v) \in p\}$
   (b) For each edge $(u, v) \in p$
      i. $f(u, v) \leftarrow f(u, v) + c_f(p)$
      ii. $f(v, u) \leftarrow f(v, u) - c_f(p)$

The path p can be found with, *e.g.*, a *breadth-first* search or a *depth-first* search in $G_f(V, E_f)$. The former which is called the Edmonds-Karp algorithm has been implemented in Scicos.

By adding the flow augmenting path to the flow already established in the graph, the maximum flow will be reached when no more flow augmenting paths can be found in the graph. When the capacities are integers, the runtime of Ford-Fulkerson is bounded by $O(E * f_{max})$, where E is the number of edges in the graph and f_{max} is the maximum flow in the graph. This is because each augmenting path can be found in $O(E)$ time and increases the flow by an integer amount which is at least 1. The Edmonds-Karp algorithm that has a guaranteed termination and a runtime independent of the maximum flow value runs in $O(VE^2)$ time.

5.2 Problem of Proposition of Alternative Variables

In order to handle this problem, we build the bipartite graph shown in Figure 5. The left-hand side vertices indicate unknowns, and each vertex at the right-hand side indicates an equation. The edges are bidirectional and their capacity is infinite.

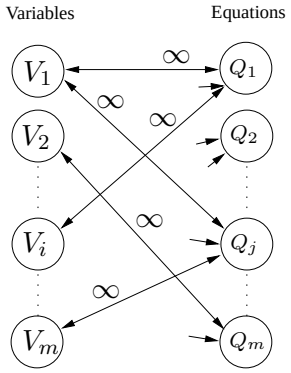


Figure 5. Bipartite graph of variables and equations

Note that, at this stage of initialization, the number of unknowns and the number of equations are identical and the incidence matrix is full rank.

For the problem of proposing alternative variables that can be initialized instead of a variable V_i, based on the bipartite graph in Figure 5, we build another directed graph as shown in Figure 6. In this graph, a source vertex and a target (sink) vertex have been added to the graph. The edge connecting the source vertex to V_i has infinite capacity. All m edges connecting the target vertex to the variable vertices have the capacity 1 (except the edge connected to the vertex V_i). The edges are mono-directional.

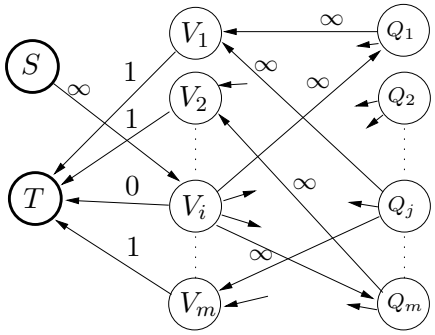


Figure 6. Directed graph for the problem of proposing all alternative variables for V_i

Now, the problem of finding all alternative variables for V_i is transformed into that of finding of all feasible paths from the source to the target. All predecessors of the target are possible alternative variables that can be used instead of V_i. In the initialization GUI, when the user double-clicks on a variable, its alternative variables are displayed. This is a useful help during the initialization.

6. Initialization Iterations

The role of the GUI and the marking in the initialization loop (see the flowchart in the Figure 3) can be summarized in the following algorithm.

1. The GUI automatically marks the model parameters, the differential variables and the algebraic variables whose guess value are given.
2. In the GUI, the user can
   - visualize/modify the `fixed` attribute of the variables and the parameters.
   - change the guess values of variables and parameters (final values if they are fixed).
   - modify whether a variable or a parameter is marked or not.
3. Initialization is invoked.
   - If necessary, the model is compiled. The Modelica compiler tries to reduce the number of unknowns by performing several stages of substituting and elimination. In this phase the marked variables are more likely to be eliminated by the compiler.
   - A numerical solver is used to find the solution of the reduced model.
   - The obtained solution values are send back to the GUI to be displayed.

4. If the obtained results are satisfactory, goto **step 7**.
5. The user can readjust the guess values of the remaining unknowns. If there are still unmarked unknowns in the reduced model, either the user can provide more accurate guess values for them or can click on the variables to see their alternatives variables. The alternative variables should be marked to be remained in the reduced model.
6. Goto step 2
7. Start the simulation

7. Example

The model of a thermo-hydraulic system is shown in Figure 6. In this model, there are a pressure source, two pressure sinks, three pipes (pressure losses), a constant volume chamber, and two flow-meter sensors linked to a Scicos scope.

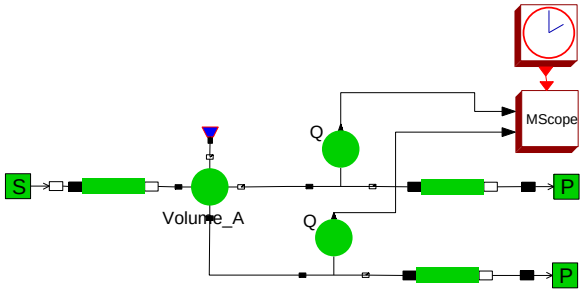


Figure 7. A thermo-hydraulic system

As shown in Figure 8, the initial non-simplified model is composed of 132 equations, 131 relaxed variables and 1 relaxed parameter (*i.e.*, 132 unknowns). The number of fixed parameters and variables are 36 and 1, respectively.

When the model is simplified, the model size is reduced to only 11 unknowns. In Figure 9, where the display mode is `Reduced`, the remaining variables as well as their solution values are shown.

8. Conclusion

In the Modelica models, initialization is an important stage of the simulation. At the initialization, variables and parameters can be fixed or relaxed and their start values can be changed by the user. In this paper, we presented a special GUI to facilitate the task of selecting fixed and relaxed variables.

Acknowledgments

The author would like to thank Sébastien Furic (LMS.Imagine Co.) for a number of helpful comments.
This work is supported by the ANR/SIMPA2-C6E2 project.

References

[1] K. E. Brenan, S. L. Campbell, and L. R. Petzold. Numerical solution of initial-value problems in differential-algebraic equations. *SIAM pubs., Philadelphia*, 1996.

[2] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold. Consistent initial condition calculation for differential-algebraic systems. *SIAM Journal on Scientific Computing*, 19(5):1495–1512, 1998.

[3] S. L. Campbell, J-Ph. Chancelier, and R. Nikoukhah. *Modeling and simulation Scilab/Scicos*. Springer Verlag, 2005.

[4] J. P. Chancelier, F. Delebecque, C. Gomez, M. Goursat, R. Nikoukhah, and S. Steer. *An introduction to Scilab*. Springer Verlag, Le Chesnay, France, 2002.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd edition, 2001.

[6] Timothy A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[7] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, 2004.

[8] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Softwar 31(3)*, pages 363–396, 2005.

[9] A.C. Hindmarsh. The pvode and ida algorithms. *LLNL technical report UCRL-ID-141558*, 2000.

[10] M. Najafi and R. Nikoukhah. Initialization of modelica models in scicos. *Conference Modelica 2008, Bielefeld, Germany.*, 2008.

[11] Ronald L. Rivest and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill, Inc., New York, NY, USA, 1990.

[12] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 114–122, New York, NY, USA, 1981. ACM.

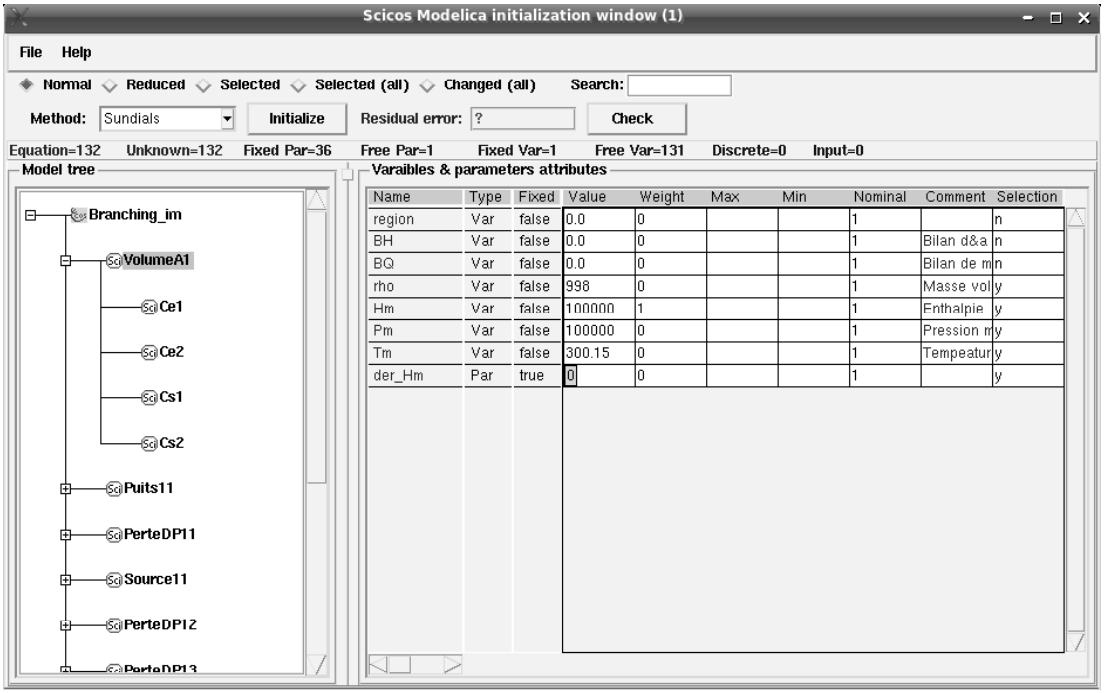


Figure 8. The initialization GUI for the model in Figure 6 (the display mode is `normal` and the variables and the parameters of the block `Volume` are shown)

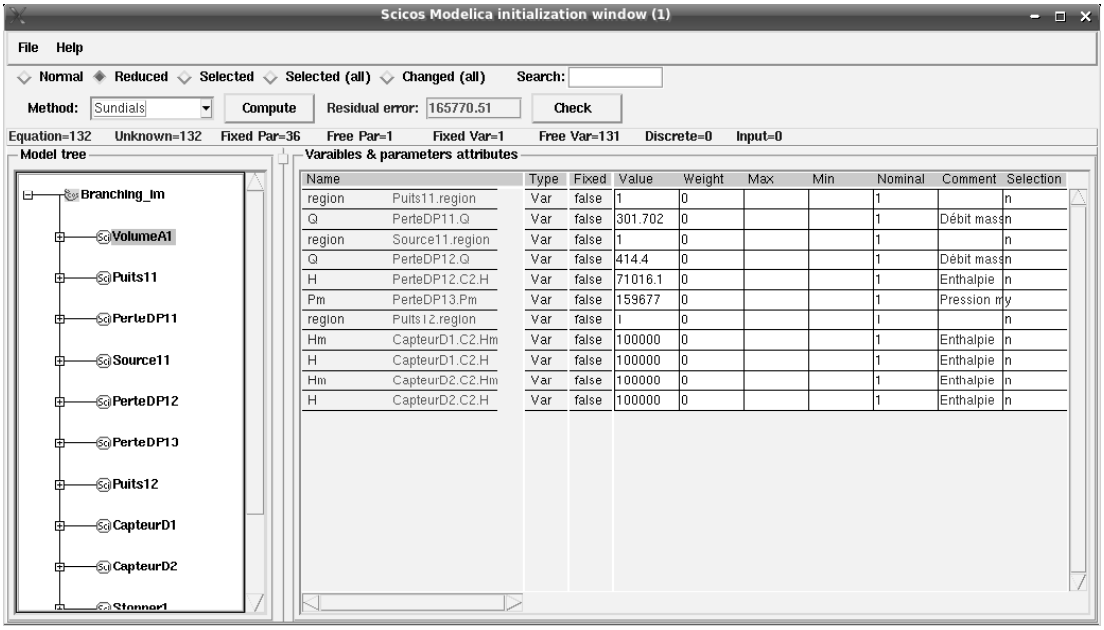


Figure 9. The remaining variables as well as their initial values after the model simplification. The display mode is `Reduced`.

Supporting Model-Based Diagnostics with Equation-Based Object Oriented Languages

Peter Bunus[1] Karin Lunde[2]

[1]Department of Computer and Information Science, Linköping University, Sweden,
petbu@ida.liu.se
[2]University of Applied Sciences Ulm, Germany,
k.lunde@hs-ulm.de

Abstract

The paper focuses on the application of equation-based object oriented languages to creating models for model-based diagnosis. We discuss characteristics and language constructs essential for diagnostic purposes. In particular, we describe the main features of the declarative modeling language Rodelica, which is based on the well-known language Modelica but enhances it with additional features related to diagnosis. Rodelica is used in a commercial model-based diagnosis tool to build and exploit complex diagnostic models of industrial size. Developed models can be used in an interactive diagnostic process as well as for the generation of more compact forms of diagnostic knowledge like diagnostic rules or decision trees which are popular for on-board diagnostics or trouble-shooting in the service bay. A case study concludes the paper, illustrating those applications and emphasizing their implications for the language itself.

Keywords: model-based diagnostics, Modelica, Rodelica, constraint propagation, interval arithmetic, failure mode, decision tree.

1. Introduction

In today's global economy, time to market decreases due to the global competitive pressure and due to the increased customer's demand for new products with new and improved functionality. A company's market share depends largely on its capability to satisfy the ever increasing customer requirements with respect to functionality and reliability. The shortened development times and the increasing complexity of the products - as indicated by the significantly increasing electrical and electronic content - may lead to difficulties if not handled appropriately. Despite careful development and construction, some of these components may eventually fail. To avoid unnecessary damage, environmental or financial, there is a need to locate and diagnose these faults as fast as possible. This can be done with a diagnostic system, which should alert the user if there is a fault in the system and, if possible, indicate the reason behind it.

The importance of the ability to perform high quality and high reliability diagnostics on a system lies in:

- Lowering the repair and maintenance time of the real system which results in lower maintenance costs and increased customer satisfaction.
- Lowering the number of sound components that are replaced erroneously during maintenance and repair.
- Lowering the downtime and non-operational time of critical systems.

In the last decade, model-based technology in diagnosis matured so far that it was transferred from academic research into real applications. It provides an alternative to more traditional techniques based on experience, such as rule-based reasoning systems or case-based reasoning (CBR) systems. Early model-based diagnosis tools include MDS (Mauss, et al. 2000 [11]), RAZ'R (Sachenbacher, et al. 2000 [15]) and RODON (Lunde, et al. 2006 [9]).

In early diagnosis systems, the knowledge about interrelations between observations (symptoms), possible failure causes, and repair actions were encoded in simple rules like this: "*If the back rest of a passenger seat system (business class) cannot be moved to the upright position although the forward activation button is pushed then the hydraulic controller may be defective or the forward activation button might be disconnected*". Rule-based systems are fairly simplistic. As it can be seen from the previous example, they provide a set of assumptions and a set of rules that specify how to act on the set of assertions. The advantage of the rule-based systems is that they are very efficient with respect to memory and computing time. For this reason, they are still widely used in practice, especially for on-board diagnostics when the lack of computational power is an issue. However, in modern systems on-board rules are automatically generated by model-based tools, which are able to produce more complete and systematic rule sets compared to the traditional hand-written ones.

In a case-based reasoning system, system expertise, fault detection and fault isolation is embodied in a library of past cases rather than in classical rules. When presented with a new target problem, a case-based reasoning

2nd International Workshop on Equation-Based Object-Oriented Languages and Tools. July 8, 2008, Paphos, Cyprus.

EOOLT 2008 website:
http://www.eoolt.org/2008/

system will first attempt to retrieve from the case data base a similar case that is relevant to solve the target problem. For a diagnostic problem, a case usually consists of a symptom, and a repair solution. The foundations of cased-based reasoning systems are described by Pal and Shiu 2004 [13], and an overview of industrial cases is given by Althof, et al. 1995 [1].

In comparison, the model-based approach tries to use the knowledge incorporated in a model to derive the symptom-failure associations as well as appropriate repair actions automatically. The existence of a modeling language that can be used for capturing model knowledge for diagnostics purposes is central for model-based-diagnosis. Early model-based diagnosis systems used traditional general purpose programming languages for specifying models. However, in some aspects, general purpose programming languages are inadequate to the task of formalizing significant domains of engineering practice. They lack the expressiveness and power of abstraction required by engineers. In this paper, we propose a declarative equation-based language called Rodelica. It is derived from the standardized modeling language Modelica (Modelica Association 2007 [2]) and differs mainly in the behavior representation, by using constraints rather than differential equations. This deviation is due to the requirements of model-based diagnosis.

The rest of the paper is organized as follows: In Section 2, we give a brief description of the principles of model-based diagnosis, by presenting some classical textbook examples to illustrate and explain the main concepts. In Section 3, we introduce the Rodelica language, and we highlight the main language constructs that facilitate various failure analyses. The extended case study presented in Section 4 illustrates the potential and the benefits of the Rodelica modeling language. The model presented in the paper was built using RODON, a commercial model-based reasoning (MBR) system. The Rodelica language is an essential part of RODON's integrated modeling environment. We show how, from the developed model, we are able to automatically derive decision trees for troubleshooting of the system in the workshop, and decision rules for on-board diagnosis. We will also illustrate an interactive model-based diagnostics process in which additional measurements are proposed in case that the initial diagnosis does not result in a single candidate as a root case. Finally, Section 5 presents our conclusions.

2. Principles of Model-Based Diagnosis

The basic principle of model-based diagnosis consists in comparing the actual behavior of a system, as it is observed, with the predicted behavior of the system given by a corresponding model. A discrepancy between the observed behavior of the real system and the behavior predicted by the model is a clear indication that a failure is present in the system. Diagnosis is a two stage process: in the first stage, the error should be detected and located in the model, and in the second stage, an explanation for that error needs to be provided. Diagnoses are usually performed by analyzing the deviations between the nominal (fault free) behavior of the system and the measured or observed behavior of the malfunctioning system.

In Figure 1, a model of a real system (an airplane passenger seat system) is depicted at the lower left corner. It might contain, for example, the behavior of the mechanical components incorporated in the seats, or the behavior of the in-flight entertainment system, or both. Note that, like all models, the model is only an abstraction of the real system (depicted at the upper left corner) and can be incomplete. The granularity of the model and the amount of information and behavior that is captured into it will directly influence the method employed by the reasoning engine as well as the precision of the diagnostic process.

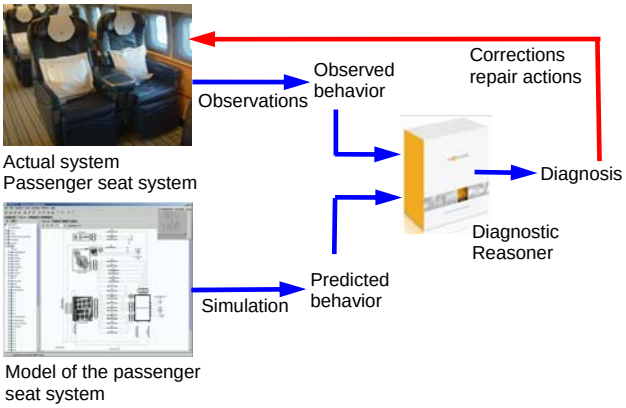


Figure 1. Basic principle of model-based diagnosis

As a general rule, the models are built to enable the identification of failed least repairable units (LRUs). Once a model of the real system is built, simulation or prediction can be performed on the model. The predicted behavior, which is the result of the simulation, can then be compared with the observed behavior of the real system. This comparison is usually done by a reasoning engine (in our case RODON) that is able to detect discrepancies and also to generate and propose corrective actions that need to be performed on the real system to repair the identified fault. We should note that the process of diagnosis (incorporated in the diagnostic reasoner) is separated from the knowledge about the system under diagnosis (the model). This ensures that the model can be reused for other purposes as well, such as optimization and reliability analysis.

The main ideas of model-based reasoning can be illustrated by the following simple multiplier-adder circuit taken from Kleer and Kurien 2003 [7].

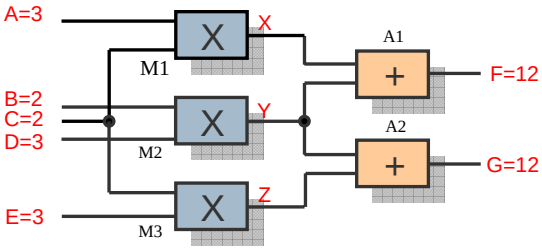


Figure 2. Multiplier-Adder circuit

The inputs to the system are `A`, `B`, `C`, `D` and `E`, while the outputs of the system are `F` and `G`. `X`, `Y` and `Z` are inter-

mediary probing points in the system. If requested, the user can perform an observation and measure the value in these points. If the system is supplied with the following input set: A=3, B=2, C=2, D=3, E=3, then the expected output should be F=12 and G=12. The output is calculated by using a deduction-based reasoning, first computing the intermediate results X=6, Y=6, and Z=6 at the output of the multiplier gates, which then are becoming inputs to the adder gates. For computing the results, a simple inference engine like the ones used by most simulation environments is enough to perform the calculations. Note that the behavior of the components is formulated in terms of relations between model variables which are called constraints. In general, constraints are not directed (unlike assignments), and may be of various nature, e.g. equations, inequalities, or logical relations.

A *conflict* is any discrepancy or difference between the prediction made by the inference engine and the observed behavior. Now let us suppose that by observing the real system one notices that the output value of F is 10, which is different from the expected calculated value of 12. In our case, the observation F=10 leads to a conflict that will be the triggering point for the diagnostic engine to indicate that something is wrong. It means that one of the components in the system does not work correctly, in other words: there is a contradiction between the assumptions and the observed behavior. In the next step, the diagnostic reasoner should find an explanation for the contradictory observation.

A possible explanation of this behavior might be that adder A1 or the multiplier M1 is defective which might cause the defective output F=10 (as depicted in Figure 3).

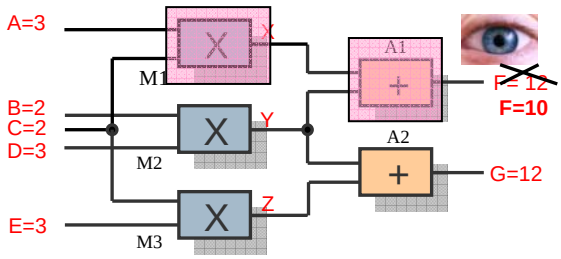


Figure 3. A defective adder A1 or multiplier M1 might explain the wrongly computed value F=10

A wrongly computed output by the adder A1 might be due to the fact that the adder itself is defective, in which case we can add the adder to the possible list of candidates, or due to the wrong inputs received by the adder. It should be noted that the inputs of adder A1 are the outputs of multiplier M1 and M2. The output of M2 is also input to the second adder A2 that computes the value of G. Since the value of G was correctly computed, we can exclude M2 from the suspected candidates. A defective M2 would certainly influence the computed value of G. Therefore, with the current information about the system we can conclude that either M1 or A1 is defective.

So far, we have considered only single faults in our multiplier-adder circuit. Considering multiple simultaneous faults will give us a new set of candidates, as depicted in Figure 4.

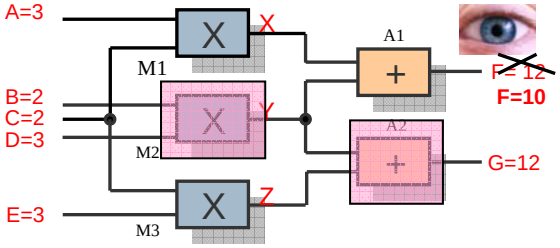


Figure 4. Multiple fault: Both M2 and A2 are defective

Let us analyze the set of multiple fault candidates shown in Figure 4 and see if it is consistent with the observation. The multiplier M2 is defective, which means that it will produce a wrong output, which will be input to the adder A1. With a wrong input, the adder A1 will produce a wrong output, which might explain that we observe F = 10. Moreover, the output of M2 is also input to the adder A2, which is also defective. It might happen that the defect inside A2 compensates the defect of the multiplier M2, and by chance, it produces a good value for the output G. With the given information, this is what we can conclude about the system. This reasoning method is called abductive reasoning. It starts from a set of accepted facts and infers their most likely explanations in the form of hypotheses. If at a later time, new evidence emerges which disagrees with a hypothesis, this hypothesis will be proven false and must be discarded. In our example, the hypothesis "M2 and A2 are faulty simultaneously" might be disproved by an additional measurement of the value Y = 6. But without additional information about the system, it is a valid hypothesis that explains the abnormal observation. By using the same type of reasoning, more hypotheses can be advanced, e.g. “M2 and M3 are simultaneously faulty”.

3. Rodelica, a Modeling Language for Diagnosis

For modeling diagnosis-related problems, like the example presented in the previous section, we describe the Rodelica language which is strongly related to the equation-based object-oriented modeling language Modelica. Rodelica was first proposed by Lunde 2000 [8]. Afterwards, it was implemented, and it is now the modeling language of the commercial model-based reasoning tool RODON. Since 2001, Rodelica was exploited in a large number of industrial-size projects.

The class concept of Rodelica is identical to the class concept of Modelica. However, we added set-valued data types as well as a new behavior description with semantics which differ from the equation-based behavior description in Modelica. Those deviations are motivated by the requirements of applications in model-based diagnosis. We describe the most important diagnosis-related semantic features of the Rodelica language in the following subsections.

3.1 The need for over-constrained systems

Modeling and simulation environments associated to traditional equation-based modeling languages like Modelica (Modelica Association 2007 [2]) , gProms proposed by Barton and Pantelides 1993 [3], Ascend proposed by Piela, et al. 1991 [14] or VHDL-AMS (Christen and Bakalar 1999 [6]) use numerical solvers for performing simulations. They rely on the fact that the system of differential equations extracted from the model is structurally and numerically nonsingular. The structural singularity checks whether the system of equations is well-posed or not, but cannot guarantee anything regarding existence or uniqueness of the solution. For this reason, the structural consistency checking is considered as a preprocessing phase to the more powerful notion of numerical singularity. The need to ensure that a system of equations extracted from the model is structurally nonsingular imposes some additional restrictions on the semantics of traditional equation-based object-oriented languages. For example, a necessary but not sufficient condition for ensuring the structural nonsingularity is that the number of equations must be equal to the number of variables. The second necessary condition is that the sparsity matrix associated to the structural Jacobian with respect to the higher order derivatives in the system can be permuted in such a way that it has a non-zero free diagonal. This restriction, imposed by the existence of a numerical solver for computing the solution of the equations, makes it impossible to formulate over- or under-constrained models. Models formulated in traditional equation-based languages need to be well-constrained. Several methods have been proposed to check the structural nonsingularity of the underlying system of equations associated to a model built using a traditional equation-based language. For the Modelica language, Bunus and Fritzson 2004 [5] proposed a graph theoretical approach for checking the structural nonsingularity and for debugging over- or under-constrained systems. Broman, et al. 2006 [4] proposed a concept called structural constraint delta to determine over- and under-constrained systems of equations in models by using static type checking and a type inference algorithm. Recently, additional restrictions have been added into the Modelica language in order to ensure that each model is "*locally balanced*", which means that the number of unknowns and equations must match on every hierarchical level. The rationale behind these restrictions introduced in Modelica 3.0 is presented by Olsson, et al. 2008 [12].

As we have seen in Section 2, a model-based diagnosis reasoning algorithm is triggered by conflicts. In Section 2 Figure 3, we present a situation in which the observed value `F` was equal to 10 compared to the computed value `F=12`. A model-based diagnosis system should have the possibility to specify and enter symptoms consisting of observations. Adding an observation to the model will automatically add an extra constraint (equation) making the model over-constrained. Model-based diagnosis systems like RODON use constraint solvers for performing diagnosis tasks that do not require the model to be well-constrained. It should be also noted that the main task of a model-based diagnosis system is to compute a diagnosis and not to perform a simulation.

The inference engine (constraint solver) will make use of the constraint network that is automatically extracted from the model. A constraint network is a set of variables and relations between them, described by constraints. Together, they define the admissible values for all variables. The constraint network is the equivalent to the flattened form of equations extracted by the Modelica compiler. Recall that there are no structural singularity conditions imposed on the extracted constraint network. The inference engine is able to operate with both insufficient information and redundant information. Inference strategies transform the constraint network into equivalent networks which describe the set of solutions in a more explicit way. Often they do not solve the problem directly, but they reduce the search space by problem reformulation. Transformations include reduction of variable domains and addition, removal, or modification of constraints. The reasoning process is explained in detail in Lunde 2006 [10].

3.2 The need for failure modes

The multiplier-adder circuit described in Section 2 has only used the correct behavior description of its constitutive components. The "*correct behavior*" models are usually easy to acquire; this kind of information should be available at the product design phase. As we have seen in the previous example, one could compute a list of candidates whose abnormal behavior might explain the faulty observed behavior. However, this list can be significantly reduced in size if additional information is available, in the form of models describing the most probable modes of faulty behavior. A diagnostic engine can use these additional behavior modes to check whether the assumption of an abnormal behavior mode explains the observed system behavior. The following example will illustrate why the specification of the faulty behavior (fault modes) is very helpful in order to achieve physically sensible diagnostic results.

Let us consider the simple electrical circuit from Figure 5 consisting of three electrical bulbs (`B1`, `B2`, `B3`) connected in parallel to a battery (`BAT`). The wire connection between the battery `BAT` and bulb `B1` is marked `w1` in Figure 5. The connection between `B1` and `B2` is marked `w2`, while a wire connection named `w3` connects bulb `B2` and bulb `B3`.

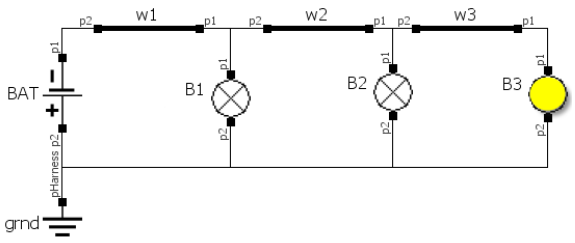


Figure 5. Simple electrical circuit consisting of three electrical bulbs connected in parallel, and illustrating the use of failure modes in diagnosis

Now let us consider the following symptom: the circuit is correctly powered by the battery, and we observe that only bulb B3 emits light, while B1 and B2 are off (are not emitting any light). If we admit the possibility of multiple faults in our circuit, we can conclude that a simultaneous failure in both B1 and B2 explains the behavior that we just observed. An alternative explanation consists in a simultaneous defect of wire w1 and wire w2. As long as we do not know anything about the nature of the defective behavior, it is thinkable that the defects in w1 and w2 will cause bulb B1 and B2 to be switched off, while they compensate each other causing B3 to behave normally, as observed. Actually, this is physical nonsense, since we know that an electrical wire can be either disconnected or have a short to ground, and there is no possible physical situation in which the failures in two different wires connected in series will compensate each other in that way and make bulb B3 emit light. Based on a similar formal reasoning, a diagnostic reasoning engine might erroneously consider the failure in the battery BAT and wire w2 as a potential candidate pair that might explain the observed behavior. To avoid those physically impossible candidates, additional information about how a component is most likely to fail should be integrated into any model intended for diagnosis purposes.

In Rodelica, it is possible to define several failure modes associated with each component. Let us consider a simple electrical component that has two pins:

```
model TwoPin
  Pin p1;
  Pin p2;
  FailureMode fm(max =1);
behavior
   // current balance that defines the
   // nominal behavior
   p1.i + p2.i = 0;
   // constraints for the failure mode
   // "disconnected"
   if (fm == 1){
      p1.i = 0;
   }
end TwoPin;
```

Compared to a Modelica representation of a TwoPin component, it can be noticed that the Rodelica TwoPin component, besides the nominal behavior, defines the behavior of the component when it is disconnected. The disconnected failure mode will have the effect that the current in pin1 will be zero (p1.i = 0). The alternative behavior of the component is specified with the help of a type variable FailureMode that acts like a switch between the two operation modes of the component. In our case, the failure mode behavior is enclosed between the brackets of the if(fm==1) statement.

A Resistor component that extends the TwoPin component can be defined as follows:

```
model Resistor extends TwoPin(fm (max = 2) );
   public     Resistance rNom(final min = 0);
   protected  Resistance rAct(final min = 0);
behavior
   // Basic constraints for nominal case:
   if (fm == 0){
      // Actual resistance is nominal resistance
      rAct = rNom;
      // Ohms law for voltage drop between pins
      p1.u - p2.u = rAct * p1.i;
    }

   // Extensions for failure mode "disconnected":
   if (fm == 1){
      // Infinite resistance between pin 1 and 2:
         rAct = INF_PLUS;
   }

   // Extensions for failure mode "short circuit
   // between pin 1 and 2":
   if (fm == 2) {
       // Same potential at pins 1 and 2:
       p1.u = p2.u;
       // No resistance between pin 1 and 2:
       rAct = 0.0;
    }
end Resistor;
```

It should be noticed that a Resistor component will inherit all constraints, and thus all failure modes, from the TwoPin component. By extending TwoPin, the resistor class has the possibility to add new constraints, thus extending the inherited failure modes or even adding new failure modes. By default, nominal behavior is assigned to the failure mode fm = 0. In the example, it is extended by specifying that the actual resistance will take the value of the nominal resistance, and by specifying Ohm's law for the voltage drop between pins. Note that constraints which are not enclosed in any if-statement (like Kirchhoff's law in the TwoPin class) are valid in all behavior modes. It should be also noticed that the Resistor has an extra failure mode that captures the situation when there is a short circuit between p1 and p2. In this case, p1 and p2 will have the same potential (p1.u = p2.u), and due to the short circuit the resistance of the Resistor will be equal to zero (rAct = 0). The short-circuit current is not specified within the resistor class.

Note that the number of constraints in each of the if-cases is not necessarily identical. This distinguishes Rodelica's if-statement from the if-statement in Modelica, where each branch is required to contain exactly the same number of equations, thus ensuring nonsingularity of the resulting system of equations. However, allowing different numbers of constraints is very useful in diagnosis. For some components, it may be appropriate to specify a generic failure mode which summarizes all kinds of faulty behavior which is too complex to describe in detail. For instance, imagine an electrical connector block with N pins. By the laws of combinatorics there is a large number of ways how those pins can be shorted, and it is unfeasible to provide the equations for each of those potential failure modes. An elegant way to avoid this complexity is the additional definition of a universal failure mode containing no constraint at all, which then may serve to explain any unexpected behavior which is not specified explicitly.

3.3 Interval data types

Another characteristic of the Rodelica language is the use of set-valued data types for defining model variables. This

is motivated mainly by the constraint solver, which does not always compute a single solution but rather constricts the values of all model values in an iterative way as far as possible without loosing any solution. In particular, most continuous model variables have the data type `Interval`. This is especially useful when working with data that is subject to measurement errors or uncertainties. For instance, a leak in a pipe with an uncertain size can be modeled by assigning the diameter a range of reasonable values, thus avoiding a potentially infinite number of failure modes.

Consequently, the inference engine uses interval arithmetics to propagate the values of the variables through the constraint network. The basic arithmetic operations for two intervals [a, b] and [c, d] are given below:

```
[a,b] + [c,d] = [a + c, b + d]
[a,b] - [c,d] = [a - d, b - c]
[a,b] × [c,d] = [min (ac, ad, bc, bd),
                 max (ac, ad, bc, bd)]
[a,b] / [c,d] = [min (a/c, a/d, b/c, b/d),
                 max (a/c, a/d, b/c, b/d)]
```

As an example the following simple Rodelica model

```
model testIntervalAddition
   Interval x = [1,6];
   Interval y = [3,7];
   Interval z;
behavior
   z = x + y;
end testIntervalAddition;
```

will restrict the possible values of the variable `z` in the interval [4 13].

Using interval variable types will have certain unexpected effects. For instance, consider the piecewise defined function given below in Figure 6:

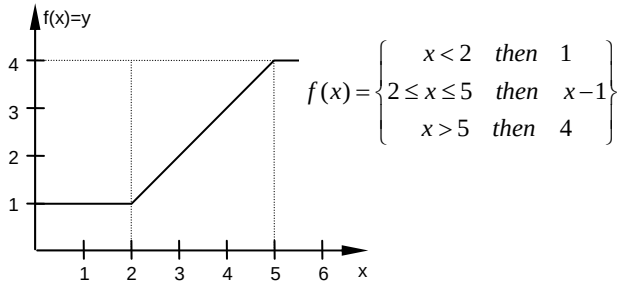


Figure 6. Piecewise defined example function

In Modelica, this function can be easily represented by the following model:

```
model PiecewiseRealFunction
   Real x;
   Real y;
equation
   y = if     (x < 2) then 1;
       elseif (x >= 2 & x <= 5) then x - 1;
       else   (x > 5) then 4;
end PiecewiseRealFunction;
```

Now, let us consider that the variables `x` and `y` are of type `Interval` and the initial value of `x` is the interval [0 10]. In this case, the conditions `x<2`, `x>=2 & x<=5` and `x>5` are undecidable – for some values in x, they are true, but for others they are false. As a consequence, none of these constraints can be evaluated and `y` remains undetermined, although it is actually clear that the resulting value range should be `y=[1 4]`. Modeling with conditional constraints in a conventional style when interval type variables are involved in the condition can lead to a loss of information. For this reason, the "`or`" clause was introduced as an alternative to the `if`-statement. In Rodelica, the piecewise real function from the example above would be more appropriately formulated as follows:

```
model PiecewiseRealFunction
   Interval x(min = 0, max = 10);
   Interval y;
behavior
   or { (x < 2; y = 1;}
        {x = [2 5]; y = x - 1;}
        {x > 5; y = 4;}
      }
end PiecewiseRealFunction;
```

An `or`-clause is treated as a single very complex constraint, whose evaluation is a two-step process: firstly, the branches of the `or` clause are evaluated separately; in a second step, the final result for each variable is calculated as the set union of the value ranges from all branches. In case that a conflict in one of the branches is detected (which means that there is no solution for at least one variable involved) the branch is excluded from the merging. As an example, assume that the variable `x` can take values between 1.5 and 4 (`x=[1.5 4]`) and y can take any real value (`y=[- +]`). By propagating the interval [1.5 4] for `x`, the three `or` branches of the piecewise real function previously defined will result in the following:

```
x = [1.5 2]; y = 1;
x = [2 4]; y = [2 4] - 1 = [1 3];
x = {}; y = 4;
```

The third branch will be excluded because it results in a conflict, which is easily detectable by the empty set assigned to x. It is the result of the set intersection of the initial value of x (`x = [1.5 4]`) and the solution of the constraint `x > 5` which is `x = [5 +]`. The unification of the solutions from the other two branches results in the overall solution `x = [1.5 4]` and `y = [1 3]`.

4. Industrial Example

Let us consider as an example a front-light power window system of a Volvo V70 car. The model was built and analyzed by means of RODON, and it is formulated using the Rodelica language. The model of the front light system contains an Electronic Control Unit (ECU), the front lights and the associated electrical harness consisting of electrical wires, fuses and connector blocks. The ECU is able to set and detect diagnostic fault codes. The activation of a diagnostic fault code is an indication that something is wrong in the system. A small section of the front light system is depicted in Figure 7.

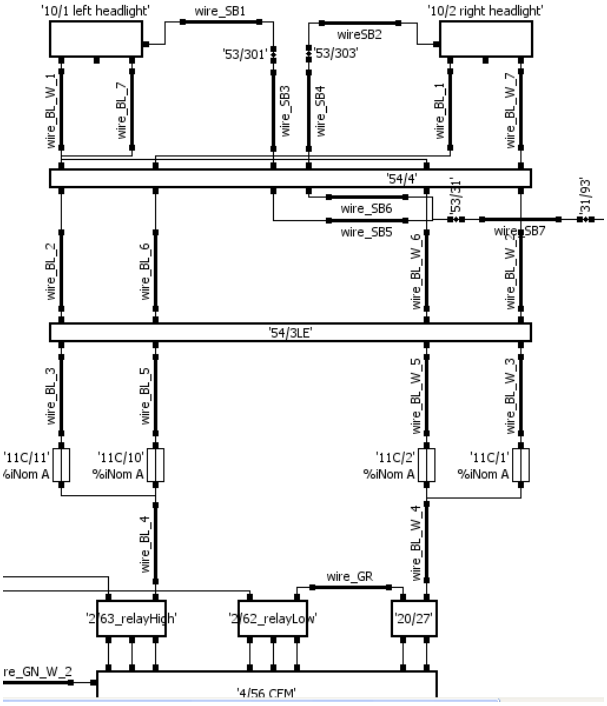


Figure 7. The front-light subsystem model of a Volvo V70 car developed in RODON.

The front light system is powered by the car battery which also powers the power-window system of the car. The power window subsystem contains an ECU of its own as well as two power window motors with hall sensors, fuses, and connector blocks and wires. The ECU detects and sets diagnostic trouble codes for power failure and Hall sensor failure. The Hall sensors are used to precisely locate the position of the window.

A small part of the power window model is shown in Figure 8.

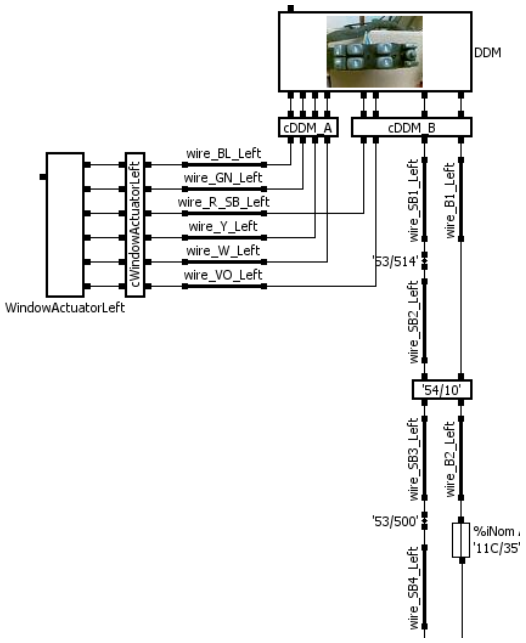


Figure 8. A part of the power-window subsystem model of a Volvo V70 car developed in RODON

The implementation details of the front-light and power-window subsystem models depicted in Figure 7 and Figure 8 are not relevant for the discussion in the paper. Let us just mention that for each component, the nominal behavior was modeled and augmented with the relevant failure modes, and that variables whose values can be measured in the real system have been marked as observable in the model.

Once the model has been created, RODON supports several diagnostic methods:

- Model-Based Diagnosis (MBD), including interactive MBD which means that additional measurements can be provided by the user to narrow down the number of diagnostic candidates.
- The automatic generation of decision trees (or diagnostic trouble-shooting trees), which can serve as a model documentation or to assist the mechanic in a workshop in a guided diagnosis.
- The automatic generation of diagnostic rules for on-board diagnostics.

In the following, two of these approaches are illustrated using the model described above.

4.1 Interactive Model-Based Diagnosis

Model-based diagnosis is the most powerful, but also the most resource-consuming diagnostic approach. By using both nominal and faulty behavior, as specified by the Rodelica model, it is able to detect single or multiple faults, or to propose additional measurements in an interactive way. Available observations and measurements can be fed to the model in several ways: there are a file interface, a GUI, and a CAN-bus interface for direct communication with the car. The main principles of MBD were described in Section 2.

As an example, we consider the situation in which the user pushes the power window button with the intent to slide down the window pane, but the pane does not move. Obviously, there is a failure in the system that immobilizes the window pane. After entering this symptom into the tool, we can start the model-based diagnostic process to find an explanation for the observed behavior, and to isolate the component that caused that particular behavior. In the first step, the diagnostic engine will compute a list of candidates (hypotheses) that explain the observed behavior:

```
PowerWindowSystem.'11C/35' disconnected,
PowerWindowSystem.'54/10' disconnected,
PowerWindowSytem.DDM.powerWindowSwitchFrontLeft.
                  switchWindowFront disconnected,
PowerWindowSystem.WindowActuatorLeft.
                  WindowUpDown disconnected,
PowerWindowSystem.cDDM_B disconnected,
PowerWindowSystem.cWindowActuatorLeft
                  disconnected,
PowerWindowSystem.wire_B1_Left disconnected,
PowerWindowSystem.wire_B2_Left disconnected,
PowerWindowSystem.wire_R_SB_Left disconnected,
PowerWindowSystem.wire_SB1_Left disconnected,
PowerWindowSystem.wire_SB2_Left disconnected,
PowerWindowSystem.wire_SB3_Left disconnected,
PowerWindowSystem.wire_SB4_Left disconnected,
```

```
PowerWindowSystem.wire_VO_Left disconnected
```

So far, 14 candidates have been identified where each corresponds to a single fault which can fully explain the symptom. The list is ordered by the associated confidence values. These confidence values are part of the model and can be imagined as “rough order of magnitude” reliability figures. Components with a lower confidence value are listed first because they are less reliable than others. In the absence of confidence values, the tool will sort the candidates by secondary criteria, for instance lexically. In the graphical user interface, the candidates are highlighted using color shades ranging from red to blue, with red representing lower confidence value and blue representing less probable candidates. The highlighting of candidates in the GUI is depicted in Figure 9.

Dealing with such a big number of candidates is not very efficient in a workshop environment where the mechanic needs to isolate the failure in a very short period of time. There is a need to narrow further down the number of candidates. This can be done by providing extra information to the tool in the form of measurements.

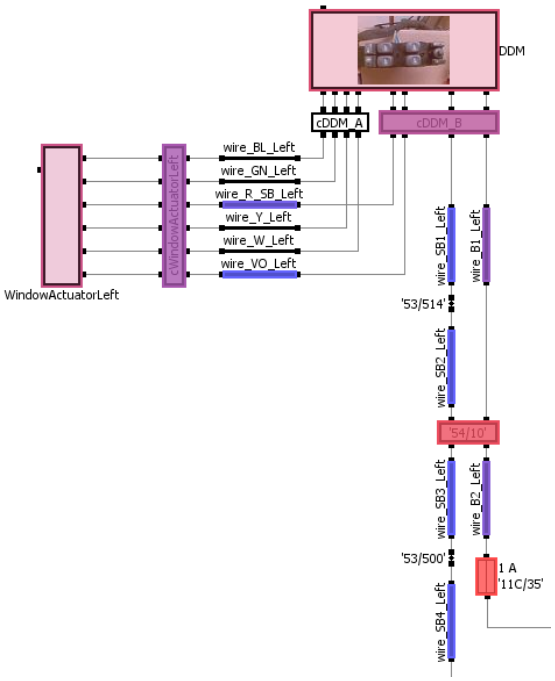


Figure 9. Diagnostics candidates are highlighted in the model browser

The inference engine can profit from this new information to validate the previously computed candidates, and possibly retract those that do not match the measured values. In Figure 10, in the upper part of the window, the list of candidates is presented, whereas the lower part shows a list of potentially useful measurements or observations to be performed on the system. The latter are ordered by the estimated impact they will have in reducing the number of candidates. The first measurement in the list has the biggest potential to reduce the number of candidates.

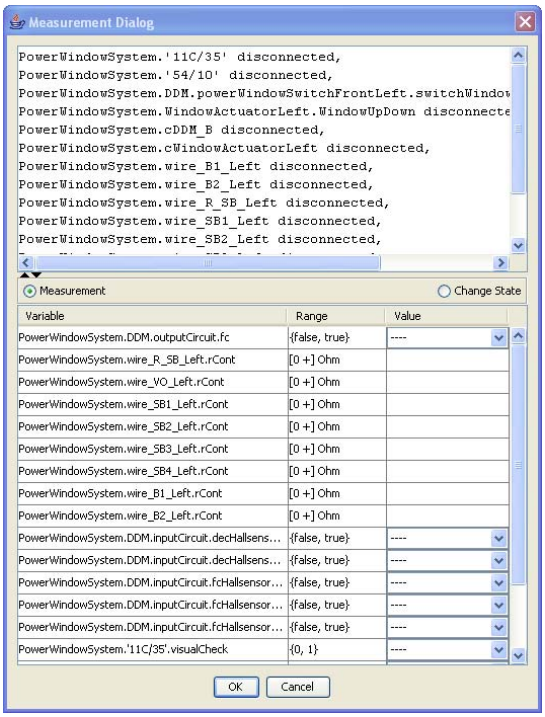


Figure 10. Lists of diagnostic candidates (hypotheses) and of proposed measurements

However, the mechanic is free to choose any measurement from the list. In practice, there might be other selection criteria which are unknown to the tool. For instance, checking a fault code activation on the dash board is less expensive than a voltage measurement on a connector block, which involves dismounting the door to have access to the electrical harness. In the present context, we choose the first proposal in the list. It is a fault code reading which can be automatically read from the car by the off-board diagnosis device. If the car dashboard is activated, the fault code may be read off the dashboard, too. The status of the fault code is given to the tool by means of the measurement GUI. It leads to the assignment of the corresponding value (true or false) to the variable `outputCircuit.fc` which is part of the subsystem `PowerWindowSystem.DDM`.. We assume that the fault code is active. This additional information is used by the reasoning engine to exclude some of the candidates from the list. In the described situation, there are only 4 candidates left:

```
PowerWindowSystem.WindowActuatorLeft.
                  WindowUpDown disconnected,
PowerWindowSystem.cWindowActuatorLeft
                  disconnected,
PowerWindowSystem.wire_R_SB_Left disconnected,
PowerWindowSystem.wire_VO_Left disconnected,
```

The diagnostic process can be continued by entering further measurements from the proposed list until the final diagnosis is produced (only one single-fault candidate is left). We call this process, in which the user is requested to provide additional measurements to progressively refine the diagnosis, *interactive model-based diagnosis* (IMBD).

4.2 Generation of Decision Trees

In environments where fewer resources are available, a more compact form of diagnostic knowledge representation is desirable. RODON is able to derive several forms of compiled diagnostic knowledge from the object-oriented Rodelica model, automatically, by means of a systematic simulation of all essential system states. To this end, the modeler has to specify which single faults and which operational states of the system are relevant for the analysis. The Cartesian product of all those operational states with the set of fault states (plus the state *System ok*) defines a so-called state space. An automatic simulation control module can then be used to simulate each state in the state space, systematically, and to write the results into a data base, which we call *state data base* (SDB). The SDB can be used for risk analyses, like failure-modes and effects analysis (FMEA), and it provides the necessary information for generating decision trees and diagnostic rules.

Decision trees are used to determine which system state explains a symptom, with minimal effort and costs. The root node of a decision tree is the symptom. Leaf nodes are result nodes describing a fault state, e.g. "*w1 is disconnected*". The intermediate nodes are decision nodes which help to discriminate the system state. Decisions may involve a measurement or a visual check to be done by the mechanic. To perform a diagnosis for a selected symptom, the decision tree is traversed starting from the root node, finally arriving at the leaf node with the correct diagnosis. The path through the tree to the diagnosis depends on the answers given at each passed decision node.

The generation process is configurable in a very high degree. In particular, actions required at the decision nodes may be more or less expensive. Consequently, the decision nodes in the generated tree are ordered with respect to a cost measure defined by the modeler. For instance, if fault code checks are declared to be cheap in comparison to actual measurements, then the mechanic will be asked by the resulting trees to check all helpful fault codes before encountering a decision node containing a measurement.

Figure 11 displays a decision tree whose root symptom is an active fault code at one of the input circuits of the Hall sensors in the power window subsystem. If this fault code is activated, the user (the mechanic) is instructed to check another fault code PDN-0020. In case that this fault code is active as well, the user is further asked to perform a visual check on `cPDM_A`, which is a switch in the power window system. Otherwise, the user is instructed to make a continuity test (Ohmic measurement) on one of the wires. Similarly to the interactive model-based diagnosis, the user is asked to perform a certain measurement or to make an observation. Based on the result of the user action, a certain branch of the failure tree is followed until the component that caused the failure is isolated.

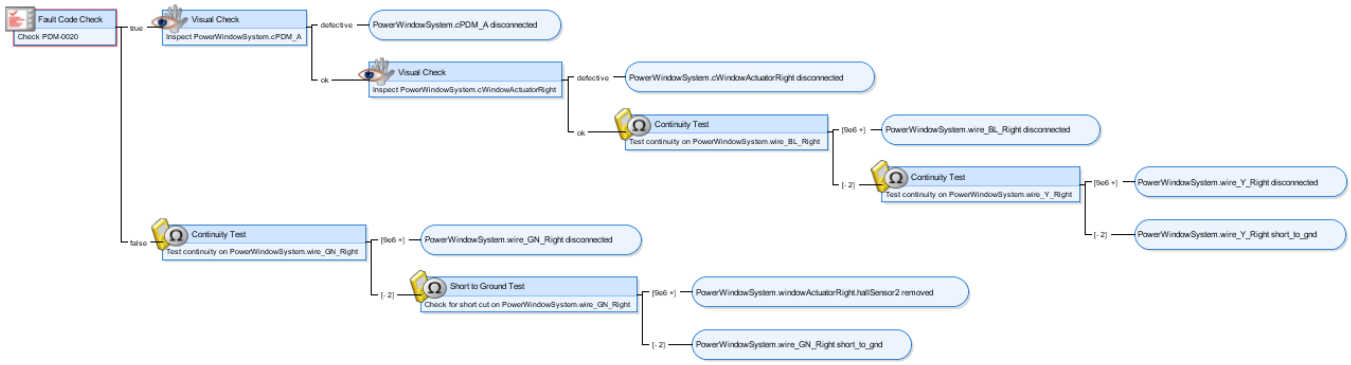


Figure 11. Generated Fault Tree

Traditionally, the generation of decision trees is done manually by the system experts, which is an extremely time consuming and error-prone task. Model-based generation of decision trees provides a systematic and safer way to analyze the combinations of all relevant operational states and component failures that can occur in a system, thus serving as a valuable tool in the authoring of troubleshooting documentation.

5. Summary and Conclusions

In this paper, we have presented Rodelica, an equation-based object oriented language derived from Modelica and adapted to model-based diagnosis purposes. Some of the characteristics of the language that makes it suitable for diagnosis are:

- Numerical representation of values in general as value sets (intervals, sets of discrete or Boolean values); qualitative representation is possible as well. This allows to cope with sensor and manufacturing tolerances as well as with insufficient information in case of faulty behavior. Both situations are common in diagnostic applications.
- Relations between variables are formulated as constraints. Supported constraint types include equations, but also inequalities, conditional constraints (if and or clauses), Boolean relations (formulas or truth tables), and spline interpolation.
- The number of constraints in a model is not restricted by the number of variables or by any notion of regularity. A model may be under- or over-determined. Underdetermined models lead to large value ranges

as the result of the inference process, over-determined models lead to conflicts which can be used as a starting point for diagnosis.

- In particular, it is possible to define failure modes for each class, in addition to modeling the nominal behavior. Any number of failure modes can be defined per class or component.
- The solver provides the appropriate computational methods based on constraint propagation and interval set arithmetic.

The benefits of using the Rodelica language have been illustrated in many industrial-size projects. Like the Modelica language, which is considered to be the "de facto" standard for modeling and simulation of hybrid systems, we believe that Rodelica can be proposed to constitute the standard for the exchange of diagnostic models.

Acknowledgements

We would like thank to the development team of RODON at Sörman Information AB Sweden for valuable discussions and the feedback received for this paper.

References

[1] Klaus-Dieter Althof, Eric Auriol, Ralph Barleta and Michel Manago (1995). *A Review of Industrial Case-Based Reasoning Tools.* AI Intelligence, Oxford, UK

[2] Modelica Association (2007). *Modelica - a Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.0.* September 2007

[3] Paul Inigo Barton and C.C. Pantelides (1993). *Gproms - a Combined Discrete/Continuous Modelling Environment for Chemical Processing Systems.* The Society for Computer Simulation, Simulation Series, vol: 25, issue: 3. pg 25-34, 1993

[4] David Broman, Kaj Nyström and Peter Fritzson (2006). *Determining over- and under-Constrained Systems of Equations Using Structural Constraint Delta*. In Proceedings of Fifth International Conference on Generative Programming and Component Engineering (GPCE'06), Portland, Oregon, USA, 2006

[5] Peter Bunus and Peter Fritzson (2004). *Automated Static Analysis of Equation-Based Components.* Simulation: Transactions of the Society for Modeling and Simulation International. Special Issue on Component Based Modeling and Simulation., vol: 80, issue: 8. pg 2004

[6] E. Christen and K. Bakalar (1999). *Vhdl-Ams-a Hardware Description Language for Analog and Mixed-Signal Applications.* IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, vol: 46, issue: 10. pg 1263-1272, 1999

[7] Johan de Kleer and James Kurien (2003). *Fundamentals of Model-Based Diagnosis*. In Proceedings of IFAC SafeProcess, Washington USA, June 2003

[8] Karin Lunde (2000). *Object-Oriented Modeling in Model-Based Diagnosis*. In Proceedings of Modelica Workshop, Lund, Sweden, Oct 23-24 2000

[9] Karin Lunde, Rüdiger Lunde and Burkhard Münker (2006). *Model-Based Failure Analysis with Rodon*. In Proceedings of ECAI 2006 - 17th European Conference on Artificial Intelligence Riva del Garda, Italy, August 29 -- September 1 2006

[10] Rüdiger Lunde (2006). *Towards Model-Based Engineering: A Constraint-Based Approach.* Shaker, Aachen

[11] Jakob Mauss, Volker May and Mugur Tatar (2000). *Towards Model-Based Engineering: Failure Analysis with Mds*. In Proceedings of ECAI-2000 Workshop on Knowledge-Based Systems for Model-Based Engineering, Berlin, Germany, 2000

[12] Hans Olsson, Martin Otter, Sven Erik Mattsson and Hilding Elmqvist (2008). *Balanced Models in Modelica 3.0 for Increased Model Quality*. In Proceedings of 6th International Modelica Conference, University of Applied Sciences, Bielefeld, Germany, March 3rd-4th 2008

[13] Sankar K. Pal and Simon C. K. Shiu (2004). *Foundation of Soft Case Based Reasoning*. John Wiley & Sons, Inc., Hoboken, New Jersey.

[14] P.C. Piela, T.G. Epperly, K.M. Westerberg and A.W. Westerberg (1991). *Ascend: An Object-Oriented Computer Environment for Modeling and Analysis: The Modeling Language*. Computers and Chemical Engineering, vol: 15, issue: 1. pg 53-72, 1991

[15] Martin Sachenbacher, Peter Struss and Claes M. Carlén (2000). *A Prototype for Model-Based on Board Diagnosis of Automotive Systems*. AI Communications, vol: 13, issue: 2. pg 83 - 97, 2000

Towards an Object-oriented Implementation of VON MISES' Motor Calculus Using Modelica

Tobias Zaiczek Olaf Enge-Rosenblatt

Fraunhofer Institute for Integrated Circuits, Design Automation Division, Dresden, Germany,
{Tobias.Zaiczek,Olaf.Enge}@eas.iis.fraunhofer.de

Abstract

This paper deals with a first implementation of the so-called motor calculus within Modelica. The motor calculus can be used to describe the dynamical behaviour of spatial multibody systems in an efficient way. This method represents an alternative approach to modelling of multibody systems. In the paper, some fundamentals of motor calculus are summarized. Furthermore, a simple implementation of motor algebra by special additional Modelica code within some components of the Modelica Multibody Standard Library is presented. This approach fully corresponds with the paradigm of object-oriented modelling. However, the present realisation is not equation-based in its full sense because of the missing possibility of operator overloading (at least in the available Modelica simulator environment). Instead of this, some functions are used carrying out the necessary calculations. Using this implementation, some examples are given to prove the applicability and correctness of the implemented approach.

Keywords Motor calculus, Screw theory, Rigid multibody system, Modelica

1. Introduction

The notion of motor, composed of the words moment and rotor, was coined by CLIFFORD in 1873 in his algebra of biquaternions [4]. But Clifford did apply his concept neither to the modelling of motion of a single rigid body nor to the modelling of spatial multibody systems. The approach of motor calculus to 3D mechanics was suggested by VON MISES in 1924 [11, 12]. In the first part [11], VON MISES introduces the dual motor product. He indicates the role of the dual motor product as a measure of the instantaneous change of a motor associated to a rigid body by the action of a second motor. In the second part [12], VON MISES applied the motor calculus in the derivation of a general form of the equations of motion of a rigid body. Due to this work, translations and rotations, velocities and angular velocities, forces and torques, etc. can be described by motor calculus (or motor algebra). Hence, this approach is well suited to investigate the behaviour of spatial multibody systems.

One of the authors studied motor calculus in his Diploma thesis [22] initiated and supervised by Prof. K. Reinschke from the Technical University Dresden (one of the former institutes of R. VON MISES). Recent publications dealing with this subject can rarely be found (exept e.g. for [8, 18]). In the context of the modelling language for heterogenous systems Modelica (see e.g. [5, 13, 19]), the motor calculus has not been taken into account up to now.

Within the Modelica community, spatial multibody systems are usually modelled using the Modelica Multibody Standard Library (see [14] or [15]). Meanwhile, many researchers apply this library to model different kinds of – partially very complex – multibody systems [2, 9, 10, 16, 20]. This library has proven to be a well suited resource to modelling such systems. However, applying the motor calculus, the equations of motion for a rigid body become more concise and clearer, e.g.

$$\dot{\mathfrak{p}} = \mathfrak{f}$$

($\mathfrak{p}$ – momentum motor, $\mathfrak{f}$ – force motor). Despite the formal equivalence to Newton's Second Law for a point mass, this equation fully describes the three-dimensional mechanics of a rigid body.

The motivation to follow up the motor calculus in the Modelica context is to investigate the possible simplification of handling spatial mechanical systems. A test realisation within the Modelica Multibody Standard Library has been carried out by implementing special additional Modelica code within some components of this library. These modifications take advantage of the built-in feature of inheritance. Hence, it is possible to compare both approaches e. g. with respect to numerical correctness.

In the following section, some fundamentals of motor calculus are shortly sketched. Some of the most important mathematical operations are defined. The test implementation is presented in section 3. It fully corresponds with the paradigm of object-oriented modelling (see e.g. [3]). In modelling and simulation, one usually distinguishes between equations and assignments. In this context, the test

2nd International Workshop on Equation-Based Object-Oriented Languages and Tools. July 8, 2008, Paphos, Cyprus.

implementation is not completely equation-based because some special mathematical operations had to be realised by functions. Some examples in section 4 show the principal applicability of the motor calculus approach.

2. Fundamentals of Motor Calculus

A *motor*

$$\mathfrak{h} = \begin{pmatrix} \boldsymbol{g} \\ \boldsymbol{h}_o \end{pmatrix}$$

is an ordered pair of vectors, $\boldsymbol{h}_o$ and $\boldsymbol{g}$, that define a vector field

$$\boldsymbol{h}(\boldsymbol{r}) = \boldsymbol{h}_o + \boldsymbol{g} \times \boldsymbol{r} \tag{1}$$

in the three-dimensional Euclidean space. In this definition, $\boldsymbol{r}$ is the position vector of any point in space, while the vectors $\boldsymbol{h}$ and $\boldsymbol{g}$ are called the *moment* and the *resultant vector* of the motor, respectively. Accordingly, $\boldsymbol{h}_o$ stands for the *moment* of the motor at the origin O of the reference coordinate system.

For every motor, an infinite number of points exists, for which the moment of the motor $\boldsymbol{h}$ is parallel to the resultant vector $\boldsymbol{g}$. All these points exhibit the same moment $\boldsymbol{h}_n$ and lie on a straight line $\mathcal{N}$ given by

$$\boldsymbol{r}_n(\lambda) = \frac{\boldsymbol{g} \times \boldsymbol{h}_o}{|\boldsymbol{g}|^2} + \lambda \boldsymbol{g} \,, \qquad \lambda \in \mathbb{R}.$$

Geometrical Interpretation A very strong goal of the motor calculus is the fact that motors and all operations with motors (that will be defined later on) can be interpreted as geometrical objects or constructions. Hence, all motors can be seen as abstract objects that do not depend on the choice of a reference frame. R. VON MISES emphasises this fact by giving the definition of motors in terms of geometrical objects describing them. Here, just an interpretation of the foregoing definition is given.

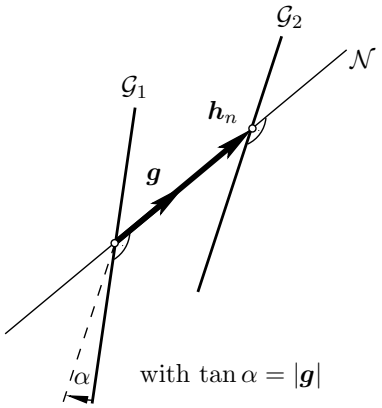


Figure 1. Geometrical interpretation of motors

For every pair of straight lines ($\mathcal{G}_1$ and $\mathcal{G}_2$) defined in Euclidean space, there exists a straight line $\mathcal{N}$ connecting them and being orthogonal to both of them (see Fig. 1). For a pair of non-parallel lines, $\mathcal{N}$ is uniquely defined. Otherwise, there exists an infinite number of such connecting lines that are parallel to each other. Now, every ordered pair of straight lines ($\mathcal{G}_1$, $\mathcal{G}_2$) can be mapped to a motor (see Fig. 1). In this case, $\mathcal{N}$ is denoted as *motor axis*, according to VON MISES. The oriented segment of the axis $\mathcal{N}$ between the intersection with $\mathcal{G}_1$ and the intersection with $\mathcal{G}_2$ can be interpreted as the moment $\boldsymbol{h}_n$ of the motor on its axis. The smaller one of both angles included by the lines $\mathcal{G}_1$ and $\mathcal{G}_2$ is understood as a meassure for the orientation and is simultaneously interpreted as the length of the resultant vector $\boldsymbol{g}$. The tangent of this angle α is equal to the length of the resultant vector, while the direction of the resultant vector is defined in such a manner that $\mathcal{G}_1$ can be transferred into $\mathcal{G}_2$ by a mathematically positive screw motion across the resultant vector. The mapping from an ordered pair of straight lines to a motor is not a one-to-one mapping because all ordered pairs of straight lines that can be transferred into each other by a screw motion across $\mathcal{N}$ define the same motor.

2.1 Motor Calculus

In the following, some computational rules of motor calculus are recalled.

Let $\mathfrak{h}$, $\mathfrak{h}_1$, and $\mathfrak{h}_2$ be three motors given by

$$\mathfrak{h} = \begin{pmatrix} \boldsymbol{g} \\ \boldsymbol{h}_o \end{pmatrix}, \qquad \mathfrak{h}_1 = \begin{pmatrix} \boldsymbol{g}_1 \\ \boldsymbol{h}_{o1} \end{pmatrix}, \qquad \mathfrak{h}_2 = \begin{pmatrix} \boldsymbol{g}_2 \\ \boldsymbol{h}_{o2} \end{pmatrix}.$$

Then, according to VON MISES, the following mathematical operations can be defined:

2.1.1 Addition

The addition of motors is performed component-wise according to

$$\mathfrak{h}_1 + \mathfrak{h}_2 = \begin{pmatrix} \boldsymbol{g}_1 + \boldsymbol{g}_2 \\ \boldsymbol{h}_{o1} + \boldsymbol{h}_{o2} \end{pmatrix}.$$

The neutral element of the addition is the zero motor

$$\mathfrak{o} = \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{0} \end{pmatrix}.$$

2.1.2 Multiplication

For the multiplication of motors the following three cases can be distinguished:

Multiplication with a scalar The scalar multiplication is defined component-wise

$$\alpha\mathfrak{h} = \begin{pmatrix} \alpha\boldsymbol{g} \\ \alpha\boldsymbol{h}_o \end{pmatrix}, \qquad \alpha \in \mathbb{R}.$$

Inner product The result of the inner product of two motors is a scalar. Thus, the product corresponds to the scalar product of the vector calculus. The definition is

$$(\mathfrak{h}_1, \mathfrak{h}_2) = (\boldsymbol{g}_1, \boldsymbol{h}_{o2}) + (\boldsymbol{g}_2, \boldsymbol{h}_{o1}), \tag{2}$$

while $(\boldsymbol{g}, \boldsymbol{h})$ is the scalar product of two vectors. Using matrix notation, the equation

$$(\mathfrak{h}_1, \mathfrak{h}_2) = \begin{pmatrix} \boldsymbol{g}_1^{\mathrm{T}} & \boldsymbol{h}_{o1}^{\mathrm{T}} \end{pmatrix} \boldsymbol{\Gamma} \begin{pmatrix} \boldsymbol{g}_2 \\ \boldsymbol{h}_{o2} \end{pmatrix}$$

holds, where $\boldsymbol{\Gamma}$ is a well chosen matrix according to

$$\boldsymbol{\Gamma} = \begin{pmatrix} \boldsymbol{0} & \boldsymbol{I_3} \\ \boldsymbol{I_3} & \boldsymbol{0} \end{pmatrix}$$

and $\boldsymbol{I_3}$ denotes the (3×3) identity matrix.

Outer product The outer product of two motors results in another motor, which is composed as follows:

$$\mathfrak{h}_1 \times \mathfrak{h}_2 = \begin{pmatrix} \boldsymbol{g}_1 \times \boldsymbol{g}_2 \\ \boldsymbol{g}_1 \times \boldsymbol{h}_{o2} + \boldsymbol{h}_{o1} \times \boldsymbol{g}_2 \end{pmatrix}. \tag{3}$$

The outer product is also referred to as motorial product or as dual motor product [6]. In terms of vectors and vector dyads, the product can be written as

$$\mathfrak{h}_1 \times \mathfrak{h}_2 = \begin{pmatrix} \boldsymbol{0} & \boldsymbol{G}_1 \\ \boldsymbol{G}_1 & \boldsymbol{H}_{o1} \end{pmatrix} \boldsymbol{\Gamma} \begin{pmatrix} \boldsymbol{g}_2 \\ \boldsymbol{h}_{o2} \end{pmatrix},$$

where $\boldsymbol{G}_1$ and $\boldsymbol{H}_{o1}$ are the cross product matrices[1] of the vectors $\boldsymbol{g}_1$ and $\boldsymbol{h}_{o1}$, respectively.

2.1.3 Motor dyads

In analogy to the vector calculus, VON MISES declared dyads for the motor calculus by linear vector functions mapping motors to motors. Referred to a concrete coordinate system, such a dyad can be represented as a (6×6) matrix.

The mapping can be described in the following manner:

$$\begin{aligned} \mathfrak{T} \circ \mathfrak{h}_1 &= \begin{pmatrix} \boldsymbol{T}_{11} & \boldsymbol{T}_{12} \\ \boldsymbol{T}_{21} & \boldsymbol{T}_{22} \end{pmatrix} \boldsymbol{\Gamma} \begin{pmatrix} \boldsymbol{g}_1 \\ \boldsymbol{h}_{o1} \end{pmatrix} \\ &= \begin{pmatrix} \boldsymbol{T}_{11}\boldsymbol{h}_{o1} + \boldsymbol{T}_{12}\boldsymbol{g}_1 \\ \boldsymbol{T}_{21}\boldsymbol{h}_{o1} + \boldsymbol{T}_{22}\boldsymbol{g}_1 \end{pmatrix}. \end{aligned} \tag{4}$$

For multiple applications of different linear vector functions, it is useful to introduce the product of two motor dyads as

$$\mathfrak{T}_1 \circ \mathfrak{T}_2 = \boldsymbol{T_1 \Gamma T_2}. \tag{5}$$

The neutral element of the dyadic multiplication in motor calculus is the identity motor dyad $\mathfrak{G}$ that can be represented in every frame as

$$\mathfrak{G} = \begin{pmatrix} \boldsymbol{0} & \boldsymbol{I_3} \\ \boldsymbol{I_3} & \boldsymbol{0} \end{pmatrix}.$$

Now, all calculation rules for the motor calculus can be derived readily, some of which are presented here for any arbitrarily chosen motors $\mathfrak{h}_1$, $\mathfrak{h}_2$, $\mathfrak{h}_3$ and $\alpha \in \mathbb{R}$:

$$\begin{aligned} (\mathfrak{h}_1, \mathfrak{h}_2) &= (\mathfrak{h}_2, \mathfrak{h}_1) \\ \mathfrak{h}_1 \times \mathfrak{h}_2 &= -\mathfrak{h}_2 \times \mathfrak{h}_1 \\ (\mathfrak{h}_1, (\mathfrak{h}_2 + \mathfrak{h}_3)) &= (\mathfrak{h}_1, \mathfrak{h}_2) + (\mathfrak{h}_1, \mathfrak{h}_3) \\ \mathfrak{h}_1 \times (\mathfrak{h}_2 + \mathfrak{h}_3) &= \mathfrak{h}_1 \times \mathfrak{h}_2 + \mathfrak{h}_1 \times \mathfrak{h}_3 \\ (\alpha\mathfrak{h}_1, \mathfrak{h}_2) &= \alpha\,(\mathfrak{h}_1, \mathfrak{h}_2) \\ \alpha\mathfrak{h}_1 \times \mathfrak{h}_2 &= \alpha(\mathfrak{h}_1 \times \mathfrak{h}_2) \\ (\mathfrak{h}_1, (\mathfrak{h}_2 \times \mathfrak{h}_3)) &= (\mathfrak{h}_2, (\mathfrak{h}_3 \times \mathfrak{h}_1)) = (\mathfrak{h}_3, (\mathfrak{h}_1 \times \mathfrak{h}_2)) \end{aligned}$$

[1] The cross product matrix $\boldsymbol{A}$ for a vector $\boldsymbol{a} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}$ is given by $\boldsymbol{A} = \begin{pmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{pmatrix}$.

Remark: Due to the definition of addition and scalar multiplication, motors span a vector space over the field of real numbers. Moreover, by the introduction of the outer product, motors form a *Lie-Algebra*[2] since all the following conditions are fulfilled:

1. The bilinearity of the motorial product is given, i. e. for all real α and β, the motors $\mathfrak{h}_1$, $\mathfrak{h}_2$, and $\mathfrak{h}_3$ satisfy the equations

   $$(\alpha\mathfrak{h}_1 + \beta\mathfrak{h}_2) \times \mathfrak{h}_3 = \alpha\mathfrak{h}_1 \times \mathfrak{h}_3 + \beta\mathfrak{h}_2 \times \mathfrak{h}_3$$

   and

   $$\mathfrak{h}_1 \times (\alpha\mathfrak{h}_2 + \beta\mathfrak{h}_3) = \alpha\mathfrak{h}_1 \times \mathfrak{h}_2 + \beta\mathfrak{h}_1 \times \mathfrak{h}_3.$$

2. The motorial multiplication is skew commutative, i. e.

   $$\mathfrak{h}_1 \times \mathfrak{h}_2 = -\mathfrak{h}_2 \times \mathfrak{h}_1.$$

3. The Jacobian Identity holds, i. e. for arbitrarily chosen motors $\mathfrak{h}_1$, $\mathfrak{h}_2$, and $\mathfrak{h}_3$, the equation

   $$\mathfrak{h}_1 \times (\mathfrak{h}_2 \times \mathfrak{h}_3) + \mathfrak{h}_2 \times (\mathfrak{h}_3 \times \mathfrak{h}_1) + \mathfrak{h}_3 \times (\mathfrak{h}_1 \times \mathfrak{h}_2) = 0$$

   is true.

2.1.4 Coordinate transformations

For concrete calculations with motors, it is necessary to introduce a coordinate system, also called frame, in which the components of the motor are given. Considering two different frames $\mathcal{F}_1$ and $\mathcal{F}_2$, it may be of interest how to transform the components of a motor $\mathfrak{h}$ given in frame $\mathcal{F}_1$ into the components referred to frame $\mathcal{F}_2$ and vice versa. So let vector $\boldsymbol{r}_{12}$ denote the position vector of the origin of $\mathcal{F}_2$ declared in frame $\mathcal{F}_1$. Furthermore, let the rotation from frame $\mathcal{F}_1$ to frame $\mathcal{F}_2$ be given by the direction cosine matrix $\boldsymbol{A}$. Then, the transformation is performed by the equation

$$[\mathfrak{h}]_{\mathcal{F}_2} = \left[\begin{pmatrix} \boldsymbol{0} & \boldsymbol{A} \\ \boldsymbol{A} & -\boldsymbol{A}\boldsymbol{R_{12}} \end{pmatrix} \circ \mathfrak{h} \right]_{\mathcal{F}_1}.$$

Here, the matrix $\boldsymbol{R}_{12}$ is the cross product matrix of the vector $\boldsymbol{r}_{12}$

2.1.5 Differentiation with respect to real-valued parameters

Consider a motor $\mathfrak{h}$ that depends on a real parameter t (e. g. the time). Then, the first derivative of this motor with respect to t can be computed component-wise:

$$\frac{d\mathfrak{h}}{dt} = \begin{pmatrix} \frac{d\boldsymbol{g}}{dt} \\ \frac{d\boldsymbol{h}_o}{dt} \end{pmatrix}.$$

[2] Named after the mathematician SOPHUS LIE (*1842, †1899).

2.1.6 Differentiation in moving frames

The temporal change of a motor seen from two different frames will, in general, lead to differing results if one frame, say $\mathcal{F}_1$, moves relatively to the other frame, say $\mathcal{F}_0$. The relative motion of the origin of frame $\mathcal{F}_1$ measured in frame $\mathcal{F}_0$ shall be given by the velocity vector $\underline{\boldsymbol{v}}_o$, while the angular velocity vector of frame $\mathcal{F}_1$ with respect to frame $\mathcal{F}_0$ is denoted by $\underline{\boldsymbol{\omega}}$. Then, the equation

$$\dot{\mathfrak{h}} = \mathring{\mathfrak{h}} + \begin{pmatrix} \underline{\boldsymbol{\omega}} \\ \underline{\boldsymbol{v}}_o \end{pmatrix} \times \mathfrak{h} \tag{6}$$

holds for the derivation with respect to time observed in frame $\mathcal{F}_0$. In Equ. (6), $\mathring{\mathfrak{h}}$ denotes the derivation w. r. t. time of the motor $\mathfrak{h}$ observed in frame $\mathcal{F}_1$.

2.2 Applications of Motor Calculus

The most important application of motor calculus is the description and analysis of the static and dynamic behaviour of rigid bodies subject to external forces and torques. Following the ideas of VON MISES, the next paragraphs will give an overview, how to describe the rigid body movements in the three-dimensional space in a very effective way using the motor calculus.

Before that, some definitions have to be explained that are essential for the succeeding subsections. To describe the motion of a rigid body in three-dimensional space, one chooses a reference point O of the body. The motion of point O can be expressed w. r. t. a reference frame $\mathcal{F}$ by the position vector $\underline{\boldsymbol{r}}_o$ (see Fig. 2). The origin of frame $\mathcal{F}$ is denoted by $\underline{O}$.

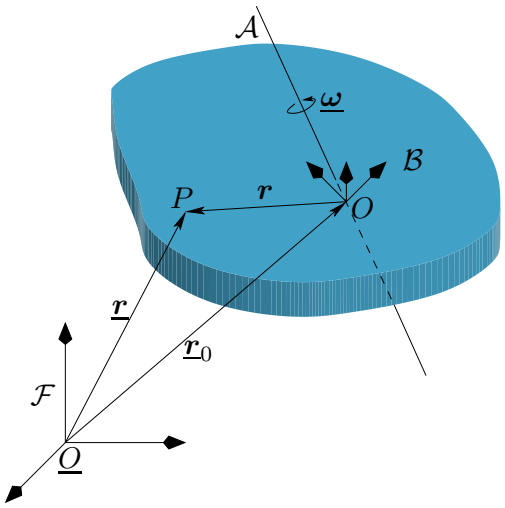


Figure 2. Definition of vectors at the rigid body

For the description of all other points of the rigid body, it is suitable to introduce a body fixed frame, called body frame $\mathcal{B}$, with the origin located in the reference point O. To distinguish the position vectors of both frames, the position vectors of the inertial frame are underlined. The position of an arbitrarily chosen point P of the body is therefore given by

$$\underline{\boldsymbol{r}} = \underline{\boldsymbol{r}}_o + \boldsymbol{r} \ .$$

The motion of the rigid body is fully described by the velocity of the reference point $\underline{\boldsymbol{v}}_o = \underline{\dot{\boldsymbol{r}}}_o$ and the angular velocity $\underline{\boldsymbol{\omega}}$ the body frame $\mathcal{B}$ is rotating w. r. t. $\mathcal{I}$.

2.2.1 Definition of physically motivated motors

The introduction of motor calculus is justified by the comfortable applicability to mechanical rigid body issues in three-dimensional space. As already described before, some physical quantities for the description of rigid body movements can be composed to motors. Hence, the motion laws of rigid body mechanics can be written in a very compact and clear form. This will be shown in the subsequent paragraphs.

We introduce some motors, that are able to describe the motion sequence of a rigid body as well as the acting torques and forces in a physically meaningful manner.

The first motor is called the *force motor* $\mathfrak{f}$ combining the resulting force $\boldsymbol{f}$ and torque $\boldsymbol{d}_o$ (referred to the reference point O) acting on the rigid body, i. e.

$$\mathfrak{f} = \begin{pmatrix} \boldsymbol{f} \\ \boldsymbol{d}_o \end{pmatrix} .$$

For any rigid body, every single force $\boldsymbol{f}_i$ and torque $\boldsymbol{d}_j$ can be assigned to a force motor according to

$$\mathfrak{f}_{f,i} = \begin{pmatrix} \boldsymbol{f}_i \\ \boldsymbol{r}_i \times \boldsymbol{f}_i \end{pmatrix}$$

and

$$\mathfrak{f}_{d,j} = \begin{pmatrix} 0 \\ \boldsymbol{d}_j \end{pmatrix} ,$$

respectively. The resulting force motor can then be simply calculated as the sum of all single force motors

$$\mathfrak{f} = \sum_{(i)} \mathfrak{f}_{f,i} + \sum_{(j)} \mathfrak{f}_{d,j} \ .$$

Please note that the overall torque $\boldsymbol{d}_o$ as well as the representation of the motor depend upon the chosen reference point O. Hence, the torque referred to any other point with the position vector $\boldsymbol{r}$ is calculated by

$$\boldsymbol{d}(\boldsymbol{r}) = \boldsymbol{d}_o + \boldsymbol{f} \times \boldsymbol{r} \ .$$

That is exactly the relationship stated in Equ. (1). This characteristic can be interpreted as a force screw (see e.g. [1]), since there always exists an instantaneous line on which the force and the torque vectors act parallel.

A second motor, the so-called *velocity motor*, is able to describe the whole motion of a rigid body. It consists of the velocity vector $\underline{\boldsymbol{v}}_o$ of the chosen reference point O and the angular velocity vector $\underline{\boldsymbol{\omega}}$ representing the rotation of the body w. r. t. an inertial frame:

$$\mathfrak{v} = \begin{pmatrix} \underline{\boldsymbol{\omega}} \\ \underline{\boldsymbol{v}}_o \end{pmatrix} .$$

This motor is able to describe the velocity $\underline{\boldsymbol{v}}$ of any point $\boldsymbol{r}$ of the rigid body by the equation

$$\underline{\boldsymbol{v}}(\boldsymbol{r}) = \underline{\boldsymbol{v}}_o + \underline{\boldsymbol{\omega}} \times \boldsymbol{r} \ .$$

Two other important vectors in the description of dynamic mechanical systems are the momentum vector $\boldsymbol{p}$ and the angular momentum vector $\boldsymbol{l}_o$. Both are combined in the *momentum motor* $\mathfrak{p}$ with

$$\mathfrak{p} = \begin{pmatrix} \boldsymbol{p} \\ \boldsymbol{l}_o \end{pmatrix}.$$

Similar to the force motor, the representation of momentum motor depends upon the chosen reference point. Between the angular momentum $\boldsymbol{l}_o$ referred to O and the angular momentum vector $\boldsymbol{l}(\boldsymbol{r})$ referred to any other point at position $\boldsymbol{r}$, the relationship

$$\boldsymbol{l}(\boldsymbol{r}) = \boldsymbol{l}_o + \boldsymbol{p} \times \boldsymbol{r}\ .$$

holds. This statement can be proven by using the definition of the vectors $\boldsymbol{p}$ and $\boldsymbol{l}_o$ according to

$$\begin{aligned} \boldsymbol{p} &= \int \underline{\dot{\boldsymbol{r}}}\, dm\ = \underline{\dot{\boldsymbol{r}}}_o \int dm + \underline{\boldsymbol{\omega}} \times \int \boldsymbol{r}\, dm \\ &= m\underline{\dot{\boldsymbol{r}}}_o - m\boldsymbol{r}_s \times \underline{\boldsymbol{\omega}} = m\underline{\boldsymbol{v}}_o - m\boldsymbol{r}_s \times \underline{\boldsymbol{\omega}} \end{aligned}$$

and

$$\begin{aligned} \boldsymbol{l}_o &= \int \boldsymbol{r} \times \underline{\dot{\boldsymbol{r}}}\, dm \\ &= \int \boldsymbol{r}\, dm\ \times \underline{\dot{\boldsymbol{r}}}_o + \int \boldsymbol{r} \times (\underline{\boldsymbol{\omega}} \times \boldsymbol{r})\, dm \\ &= m\boldsymbol{r}_s \times \underline{\boldsymbol{v}}_o + \boldsymbol{\Theta}_o \underline{\boldsymbol{\omega}}\ , \end{aligned}$$

where m denotes the mass of the body and $\boldsymbol{\Theta}_o$ the inertia tensor w. r. t. the reference point O. The vector $\boldsymbol{r}_s$ is the position vector of the centre of mass referred to the body frame given by $\boldsymbol{r}_s = \frac{\int \boldsymbol{r}\, dm}{\int dm}$.

2.2.2 Some fundamental laws of mechanics in terms of motor calculus

With the definitions above, a relationship between the velocity motor $\mathfrak{v}$ and the momentum motor $\mathfrak{p}$ can be derived by introducing the inertia dyad $\mathfrak{M}$ for the motor calculus:

$$\mathfrak{p} = \underbrace{\begin{pmatrix} m\boldsymbol{I} & -m\boldsymbol{R_s} \\ m\boldsymbol{R_s} & \boldsymbol{\Theta}_o \end{pmatrix}}_{\mathfrak{M}} \circ \mathfrak{v}\ . \tag{7}$$

The new symbol $\boldsymbol{R_s}$ describes the cross product dyad of the vector $\boldsymbol{r}_s$.

Referred to a concrete frame in u, v, and w, the dyad can be written as a (6×6) matrix of the following form

$$\mathfrak{M} = \begin{pmatrix} m & 0 & 0 & 0 & mw_s & -mv_s \\ 0 & m & 0 & -mw_s & 0 & mu_s \\ 0 & 0 & m & mv_s & -mu_s & 0 \\ 0 & -mw_s & mv_s & \Theta_{uu} & \Theta_{uv} & \Theta_{uw} \\ mw_s & 0 & -mu_s & \Theta_{vu} & \Theta_{vv} & \Theta_{vw} \\ -mv_s & mu_s & 0 & \Theta_{wu} & \Theta_{wv} & \Theta_{ww} \end{pmatrix},$$

where u_s, v_s, and w_s are the coordinates of centre of mass. Choosing the body frame parallel to the body's principal axes of inertia and selecting the centre of mass as the reference point, $\mathfrak{M}$ becomes a diagonal matrix.

With the help of the foregoing motor relations, the main mechanical laws can be rewritten in terms of motors.

The first law describes the change of momentum and angular momentum in the presence of external forces and torques in a very efficient and short way, namely

$$\boxed{\dot{\mathfrak{p}} = \mathfrak{f}}\ .$$

Here, $\dot{\mathfrak{p}}$ denotes the time derivative of the momentum motor $\mathfrak{p}$ observed in an *inertially fixed* reference frame.

The unique simplicity and shortness of this equation is doubtless a goal of this calculus, even more considering that it formally takes exactly the form of Newton's Second Law for mass points. Unfortunately, this formula is not very practical, since the derivation has to be done w. r. t. the inertial frame. However, the momentum motor is much easier to determine in a body fixed frame, because the inertia dyad $\mathfrak{M}$ is therein constant. So, a much more applicable form for concrete calculations can be derived using (6) to express the time derivation w. r. t. the body frame

$$\boxed{\mathring{\mathfrak{p}} + \mathfrak{v} \times \mathfrak{p} = \mathfrak{f}}\ , \tag{8}$$

where $\mathfrak{p}$, $\mathfrak{v}$, and $\mathfrak{f}$ are referred to the origin of the body frame.

Replacement of the momentum motor using Equ. (7) yields the following relationship

$$\boxed{\mathfrak{M} \circ \mathring{\mathfrak{v}} + \mathfrak{v} \times (\mathfrak{M} \circ \mathfrak{v}) = \mathfrak{f}}$$

if all components are given in the body frame.

The kinetic energy of a rigid body can be expressed by means of motor calculus as follows:

$$\boxed{T = \frac{1}{2}(\mathfrak{v}, \mathfrak{p}) \qquad \text{with} \qquad \mathfrak{p} = \mathfrak{M} \circ \mathfrak{v}}\ .$$

Again, this expression agrees formally with the equation of the kinetic energy of a mass point, if therein the mass is substituted by the inertia dyad $\mathfrak{M}$ and the vectors are substituted by their corresponding motors.

Similarly, the equation for the power performed by the applied forces and torques is given by

$$\boxed{P = (\mathfrak{f}, \mathfrak{v})}$$

so that the energy law for a rigid body results in

$$\boxed{\frac{dT}{dt} = \frac{1}{2}\frac{d}{dt}(\mathfrak{v}, \mathfrak{M} \circ \mathfrak{v}) = (\mathfrak{f}, \mathfrak{v})}\ .$$

3. Object-oriented Implementation

The test implementation presented here is based on the Modelica Multibody Standard Library. Hence, it fully corresponds with the paradigm of object-oriented modelling. Due to some limitations of the Modelica language, compromises had to be made during implementation of the motor calculus. Because of the necessarily used functions, the realisation is not a completely equation-based formulation.

3.1 Motor Library

The first step of the implementation towards a description of rigid body motion by means of motor calculus is the realisation of a general motor class. From the view of data structure, motors are nothing more than a combination of six scalars. According to the definition of motors provided above, the first idea of arranging these scalars within the motor class was to group them into two vectors of type `Real`. The first vector would represent the resultant vector and the second vector would be the moment vector of the motor at the reference point:

```
record Motor "Motor"
    Real[3] res "resultant vector";
    Real[3] mom "moment vector at O";
end Motor;
```

Unfortunately, this approach, similar to the implementation of the complex numbers in [7], prohibits the use of basic mathematical operators on the newly defined data types. A solution would be the overloading of these operators as it is possible in C++ [17]. However, Modelica does still not support this feature. Thus, an alternative implementation has been chosen, where all six scalars are stored within one vector:

```
type Motor = Real[6]
    "Motor: [Resultant;Moment at r0]";
```

The reason for the chosen implementation was the ability to keep at least the operators "+" and "−" as well as the multiplication with scalars for the motor calculus in its original sense. Within the context of inheritance, no real specialization concerning the physical units of the quantities can be made. Hence, the child classes of velocity motor, force motor, and momentum motor have also a quite simple definition, namely:

```
type VelocityMotor = Motor "Velocity motor";
type ForceMotor    = Motor "Force motor";
type MomentumMotor = Motor "Momentum motor";
```

All the other calculation rules introduced in section 2.1 had to be implemented using Modelica functions. The first function has been written to perform the inner product between two motors according to Equ. (2):

```
function dot "Inner product of motor calculus"
    input Motor m1 "First motor";
    input Motor m2 "Second motor";
    output Real r3 "Resulting scalar";
algorithm
    r3 := m1[1:3]*m2[4:6] + m1[4:6]*m2[1:3];
end dot;
```

Similarly, the outer product has been implemented as stated in Equ. (3):

```
function 'x' "Outer product of motor calculus"
    input Motor m1 "First motor";
    input Motor m2 "Second motor";
    output Motor m3 "Resulting motor";
algorithm
    m3 := vector([ cross(m1[1:3],m2[1:3]);
                   cross(m1[1:3],m2[4:6])
                  +cross(m1[4:6],m2[1:3])]);
end 'x';
```

A function that returns the moment of the motor for any position vector $\boldsymbol{r}$ has also been realised to simplify the motor handling:

```
function mom "Moment of the motor referred to
         position vector r"
    input Motor m "Motor";
    input Modelica.SIunits.Position[3] r
          "Position vector";
    output Real[3] mom "Moment of the motor";
algorithm
    mom := m[4:6] + cross(m[1:3],r);
end mom;
```

The foregoing reasons for the simple implementation of the motor class apply for the implementation of the motor dyads, too. Hence, a motor dyad given w. r. t. a given frame can be expressed as a (6×6) matrix:

```
type MotorDyad = Real[6,6] "Motor Dyad";
```

To apply a motor dyad to a motor, another function has been created. Referring to Equ. (4), the function has been defined by:

```
function times "Application of a Motor Dyad on a
     Motor"
    input MotorDyad m1
        "Motor dyad to be applied";
    input Motor     m2 "Input motor";
    output Motor    m3 "Output motor";
algorithm
   m3 := m1[:,1:3]*m2[4:6] + m1[:,4:6]*m2[1:3];
end times;
```

Finally, there exist two functions that are able to transform the components of a motor from one frame to another and vice versa (refer to section 2.1.4):

```
function coordChange1 "Transforms motor from
    frame a to frame b"
    import F = Modelica.Mechanics.MultiBody.
        Frames;
    input Modelica.SIunits.Position[3] r_0
         "Vector pointing from origin of frame a
              to origin of frame b, resolved in
             frame a";
    input F.Orientation R "Orientation object of
         frame b resolved in frame a";
    input Motor m1 "Motor resolved in frame a";
    output Motor m2 "Motor resolved in frame b";
algorithm
    m2 := vector([R.T*m1[1:3];R.T*mom(m1,r_0)]);
end coordChange1;

function coordChange2 "Transforms motor from
    frame b to frame a"
    import F = Modelica.Mechanics.MultiBody.
        Frames;
    input Modelica.SIunits.Position[3] r_0
         "Vector pointing from origin of frame a
              to origin of frame b, resolved in
             frame a";
    input F.Orientation R "Orientation object of
         frame b resolved in frame a";
    input Motor m1 "Motor resolved in frame b";
    output Motor m2 "Motor resolved in frame a";
algorithm
    m2 := vector([transpose(R.T)*m1[1:3];
          transpose(R.T)*m1[4:6]
          + cross(r_0,transpose(R.T)*m1[1:3])]);
end coordChange2;
```

3.2 Multibody Implementation

After implementing the most important operations of the motor calculus, we were able to take advantage of the efficient description of the rigid body motion. Therefore, as a first step, the existing implementation of a rigid body object from the Modelica Multibody Standard Library was adapted to the motor algebra. To simplify the implementation, all interfaces and all existing variables were kept. Only some small changes had to be made within the so-called `Body` class.

The first changes were the declaration of the following physically motivated `Motor` and `MotorDyad` objects:

```
// Motor Dyads
// ----------
Real[3,3]   I0     "Inertia dyad wrt. B";
MotorDyad   I_mot "Motorial inertia dyad wrt. B";

// Motors
// ------
VelocityMotor vel_B "Velocity motor wrt. B";
MomentumMotor mom   "Momentum motor wrt. B";
ForceMotor    f_g   "Gravity force motor wrt B";
ForceMotor    f_a   "Cut force motor wrt. B";
```

Afterwards, all declared motors and motor dyads had to be defined using the following statements:

```
// force motors
// ------------
f_g   = vector([ m*frame_a.R.T*g_0;
                 cross(r_CM, m*frame_a.R.T*g_0)]);
f_a   = vector([frame_a.f; frame_a.t]);

// velocity motor
// --------------
vel_B = vector([ frame_a.R.w;
                 frame_a.R.T*der(frame_a.r_0)]);

// inertia matrices
// ----------------
I0    = I + m*( diagonal(r_CM*r_CM*ones(3))
                - [r_CM]*transpose([r_CM]));
I_mot = [ diagonal({m, m, m}), -skew(m*r_CM);
          skew(m*r_CM)       , I0];

// momentum motor
// --------------
mom   = vector(times(I_mot, vel_B));
```

Finally, the equations of motion originally implemented according to

```
frame_a.f = m*(Frames.resolve2(frame_a.R,
                               a_0 - g_0)
               + cross(z_a, r_CM)
               + cross(w_a, cross(w_a, r_CM)));
frame_a.t = I*z_a + cross(w_a, I*w_a)
                  + cross(r_CM, frame_a.f);
```

have been replaced by the very clear and short Equ. (8):

```
f_a = der(mom) + 'x'(vel_B,mom) - f_g;
```

Because of the object-oriented structure of the Modelica Standard Library, the changes had to be implemented only once. All subclasses of the `Body` class, like `BodyShape`, `BodyBox`, or `BodyCylinder` inherit the changes automatically.

4. Examples and Verification

4.1 Movable Double Pendulum

As a first example, the movable double pendulum (Fig. 3) was chosen to show the correctness of the implemented body classes based on motor calculus. The pendulum consists of a trolley with the mass M_0 and two rigid bodies with masses M_1 and M_2. The trolley is able to move horizontally. The first body is suspended on the trolley by a revolute joint. The second body is suspended on the first body via a revolute joint, too. Both axes of rotation are parallel to the z-axis which lies perpendicular to the xy-plane (see Fig. 3). The moments of inertia of both bodies around the axis of rotation w. r. t. their particular centre of mass are given by J_1 and J_2.

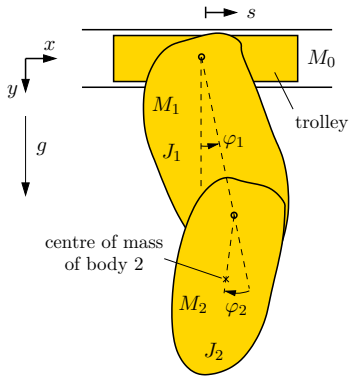


Figure 3. Sketch of double pendulum

The pendulum moves from an initial deflection of $\varphi_1(0) = 90\,\text{deg}$ and $\varphi_2(0) = 0\,\text{deg}$ due to the earth's gravity field. A viscous friction, acting in every joint, damps the motion of the pendulum. As a reference, the same pendulum system has been implemented using the Modelica Standard Library. A sketch of the structure is shown in the lower part of Fig. 5. The upper part of this figure shows the pendulum using the modified `Body` objects adapted to the motor calculus.

Fig. 4 shows the trajectory for the position s of the trolley. Figs. 6 and 7 depict the time histories of the revolute joint angles φ_1 and φ_2. In every diagram, the trajectory of both systems, the double pendulum using the motor calculus and the double pendulum using the Modelica Standard Library, were plotted together.

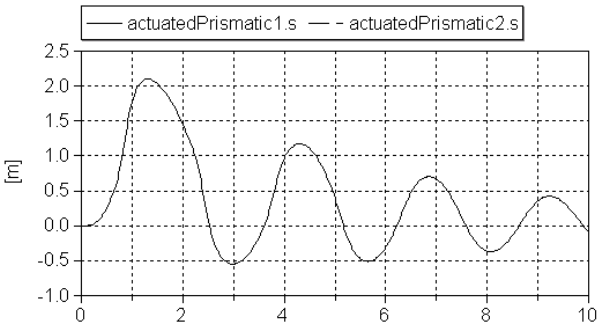


Figure 4. Trajectory of the trolley position s

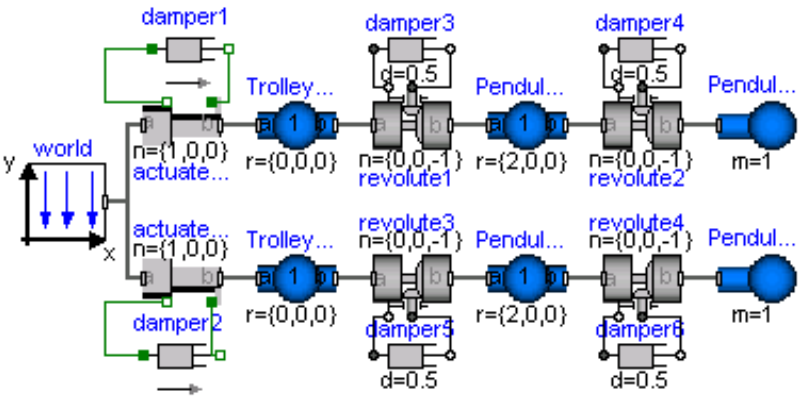


Figure 5. Implementation of double pendulum

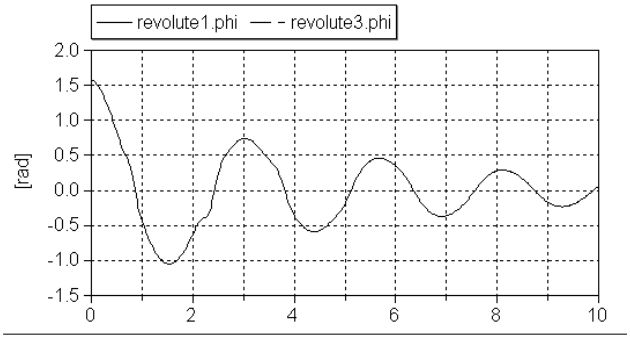


Figure 6. Trajectory of the first pendulum angle φ_1

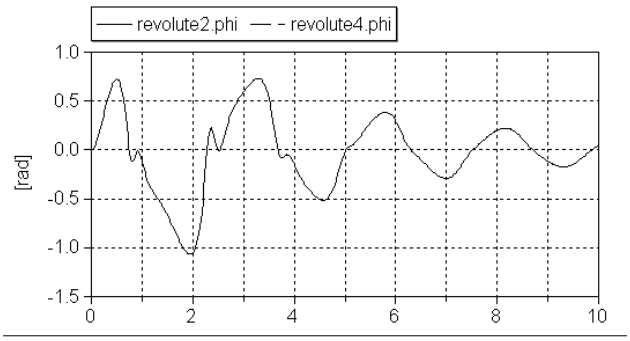


Figure 7. Trajectory of the second pendulum angle φ_2

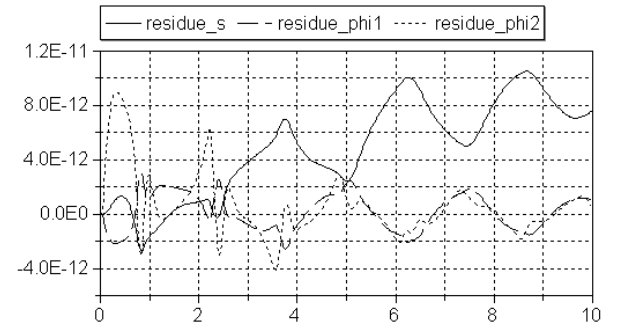


Figure 8. Deviation of the most interesting coordinates between the motor calculus and the Modelica Standard Library implementation

Fig. 8 presents the deviation for all corresponding variables (`residue_s`, `residue_phi1/2`).

Apparently, the deviation of the position stays smaller than $1.2 \cdot 10^{-11}$m for the given simulation time of 10s. The deviations of both pendulum angles are also very small. They do not exceed 10^{-11}rad. Hence, these differences can be interpreted as numerical errors of the simulator, because for simulations with a lower error tolerance, the deviations decrease.

For a comparison even a third implementation within the simulation system Matlab (refer to [21]) was consulted that led to very similar results.

4.2 Fourfold Pendulum on Two Movable Sliders

The second example is a fourfold pendulum. It consists of two trolleys and a chain of four rigid bodies between them. Both trolleys are guided along straight tracks (see Fig. 10). Hence, this example contains a *closed kinematic loop*. Similar to the foregoing example, the pendulum moves from an initial deflection due to the gravity field of the earth and is damped by a viscous friction in every joint. The initial values for the pendulum angles are

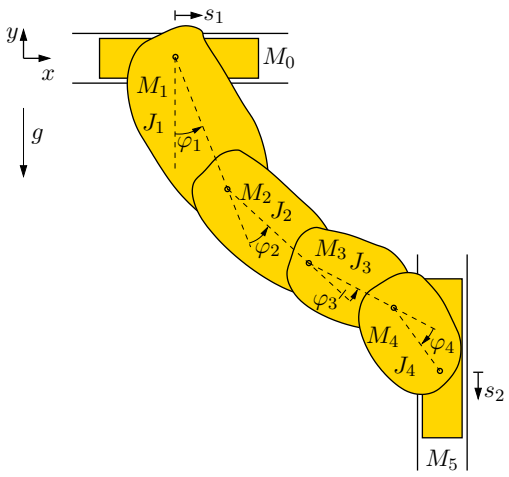


Figure 10. Sketch of fourfold pendulum

$$\varphi_1(0) = 45\,\text{deg}, \qquad \varphi_2(0) = -15\,\text{deg},$$
$$\varphi_3(0) = 30\,\text{deg}, \qquad \varphi_4(0) = -37.5\,\text{deg}.$$

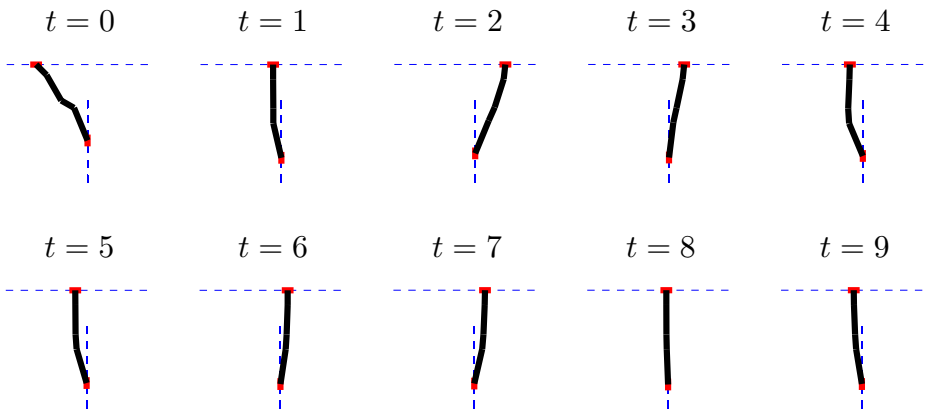


Figure 9. Sequence of configurations of the fourfold pendulum

As before, the pendulum system was implemented twice. The first pendulum system works on the basis of the modified Mechanical Multibody Library, while the second one uses the Multibody Standard Library and serves as a reference. Hence, the deviations to the modified model can be calculated. They have the same order of magnitude as in the example before and can thus be explained by numerical errors.

For the rough illustration of the simulation results, Fig. 9 shows the configuration of the pendulum at ten different time instances (the time interval is 1 s). The dashed lines show the tracks of both trolleys. The bold plotted polygon consists of four segments. It represents the idealized shape of the chain.

In Fig. 11, the position of both trolleys are plotted against the time.

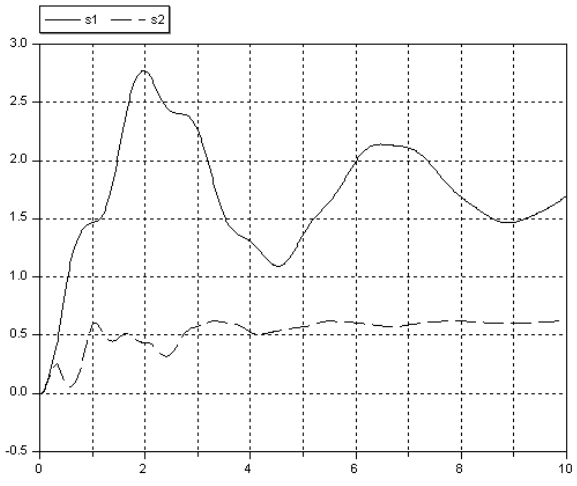


Figure 11. Position of both trolleys for the motor calculus implementation

4.3 Fourbar Mechanism

The last example is a so-called fourbar mechanism from the Modelica Standard Library, that, again, sets up a closed kinematic loop (see Fig. 12). However, in this example, the rigid bodies do not perform planar motions any more

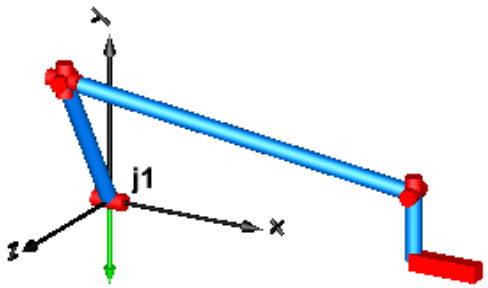


Figure 12. Sketch of fourbar mechanism

and, hence, the whole complexity of the three-dimensional mechanics is necessary.

The fourbar mechanism moves under the influence of the earth's gravitational field. The initial condition of the angular velocity of the first revolute joint ($j1$) is set to $300\,\frac{\text{deg}}{\text{s}}$. In opposit to the foregoing examples, this system is completely undamped.

As in the paragraphs before, the example was implemented twice in one model. One system has just been kept in its original form while in the second system, all `BodyCylinder` objects have been replaced by the modified `BodyCylinder` objects. The difference between both implementations is shown in Fig. 13. The numerical

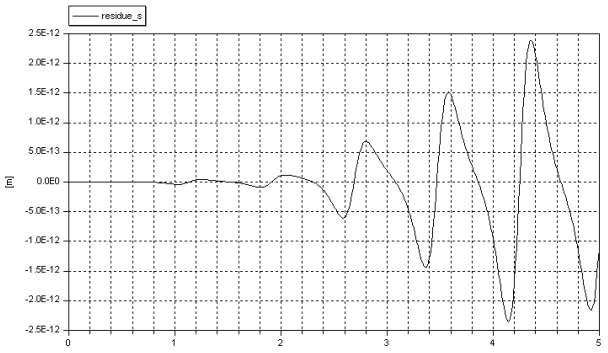


Figure 13. Deviation of the slider position between the motor calculus and the Modelica Standard Library implementation

results of the simulation show an increasing deviation with advancing time. The reason for this fact may be the absence of any damping elements. Indicated by this result, further investigations on numerical accuracy seem to be necessary for the future.

5. Summary and Outlook

The paper traces the idea of applying the so-called motor calculus within Modelica modelling language to handle models of spatial multibody systems in an efficient way. This method represents an alternative approach to modelling such systems. This approach is characterized by a clear and concise formulation of the equations of motion.

To get some experiences with possibilities and limits of this approach, a first test implementation was carried out. The Modelica Multibody Standard Library was used to implement appropriate extensions within some selected submodels. This implementation allows a comparison of the standard library implementation and the motor calculus implementation by means of simple simulation tasks. Appropriate results are presented in the paper.

These results seem to encourage the idea of motor calculus usage within Modelica. Nevertheless, there are open challenges to be solved in the future. Operand overloading would be a very helpful feature in this context. Furthermore, efficient methods have to be adapted to compute actual position and orientation of a rigid body from its velocity motor. That's why further investigations as well as implementation work will still have to be carried out for a full support of rigid-body motion equation by means of motor calculus in Modelica.

References

[1] R.S. Ball. *A Treatise on the Theory of Skrews*. Cambridge University Press, 1900.

[2] F. Casella and M. Lovera. High-accuracy orbital dynamics simulation through Keplerian and equinoctial parameters. In *6th International Modelica Conference, Bielefeld, Germany, March 3–4, 2008, Proc.*, pages 505–514. The Modelica Association, 2008.

[3] F.E. Cellier. *Continuous System Modeling*. Springer, 1991.

[4] W.K. Clifford. Preliminary sketch of bi-quaternions. *Proc. London Math. Soc.*, 4:381–395, 1873.

[5] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, 2003.

[6] J. Gallardo-Alvarado. Kinematics of a hybrid manipulator by means of screw theory. *Journal Multibody System Dynamics*, 14(3–4):345–366, 2005.

[7] A. Haumer, C. Kral, J.V. Gragger, and H. Kapeller. Quasi-stationary modeling and simulation of electrical circuits using complex phasors. In *6th International Modelica Conference, Bielefeld, Germany, March 3–4, 2008, Proc.*, pages 229–236. The Modelica Association, 2008.

[8] C. Heinz. Motorrechnung im X_{1+3+3}. *Zeitschrift für Angewandte Mathematik und Mechanik (ZAMM)*, 67(11):537–544, 1987.

[9] C. Knobel, G. Janin, and A. Woodruff. Development and verification of a series car Modelica/Dymola multibody model to investigate vehicle dynamics systems. In *5th International Modelica Conference, Vienna, Austria, September 4–5, 2006, Proc.*, pages 167–173. The Modelica Association, 2006.

[10] I.I. Kosenko, M.S. Loginova, YA.P. Obraztsov, and M.S. Stavrovskaya. Multibody systems dynamics: Modelica implementation and Bond Graph representation. In *5th International Modelica Conference, Vienna, Austria, September 4–5, 2006, Proc.*, pages 213–223. The Modelica Association, 2006.

[11] R. von Mises. Motorrechnung, ein neues Hilfsmittel der Mechanik. *Zeitschrift für Angewandte Mathematik und Mechanik (ZAMM)*, 4(2):155–181, 1924.

[12] R. von Mises. Anwendungen der Motorrechnung. *Zeitschrift für Angewandte Mathematik und Mechanik (ZAMM)*, 4(3):193–213, 1924.

[13] `http://www.modelica.org/documents`. seen on April 22nd, 2008.

[14] `http://www.modelica.org/libraries/Modelica/`. seen on April 22nd, 2008.

[15] M. Otter, H. Elmqvist, and S. E. Mattsson. The New Modelica MultiBody Library. In *3rd International Modelica Conference, Linköping, Sweden, November 3–4, 2003, Proc.*, pages 311–330. The Modelica Association, 2003.

[16] T. Pulecchi and M. Lovera. Object-oriented modelling of the dynamics of a satellite equipped with single gimbal control moment gyros. In *4th International Modelica Conference, Hamburg, Germany, March 7–8, 2005, Proc.*, pages 35–44. The Modelica Association, 2005.

[17] B. Stroustrup. *The C++ Programming Language – Special Edition*. Addison-Wesley, 2007.

[18] H. Stumpf and J. Badur. On the non-abelian motor calculus. *Zeitschrift für Angewandte Mathematik und Mechanik (ZAMM)*, 70(12):551–555, 1990.

[19] M.M. Tiller. *Introduction to Physical Modeling with Modelica*. Springer, 2001.

[20] L. Viganò and G. Magnani. Acausal modelling of helicopter dynamics for automatic flight control applications. In *5th International Modelica Conference, Vienna, Austria, September 4–5, 2006, Proc.*, pages 377–384. The Modelica Association, 2006.

[21] T. Zaiczek. Modellbildung für mechanische Systeme mit einer endlichen Anzahl von Freiheitsgraden und Steuerungsentwurf mithilfe von $m \geq 1$ Aktuatoren. Technical report, 2006.

[22] T. Zaiczek. Modellierung, Regelung und Simulation mechanischer Starrkörpersysteme im dreidimensionalen Raum. Diploma thesis, TU Dresden, Germany, 2007.