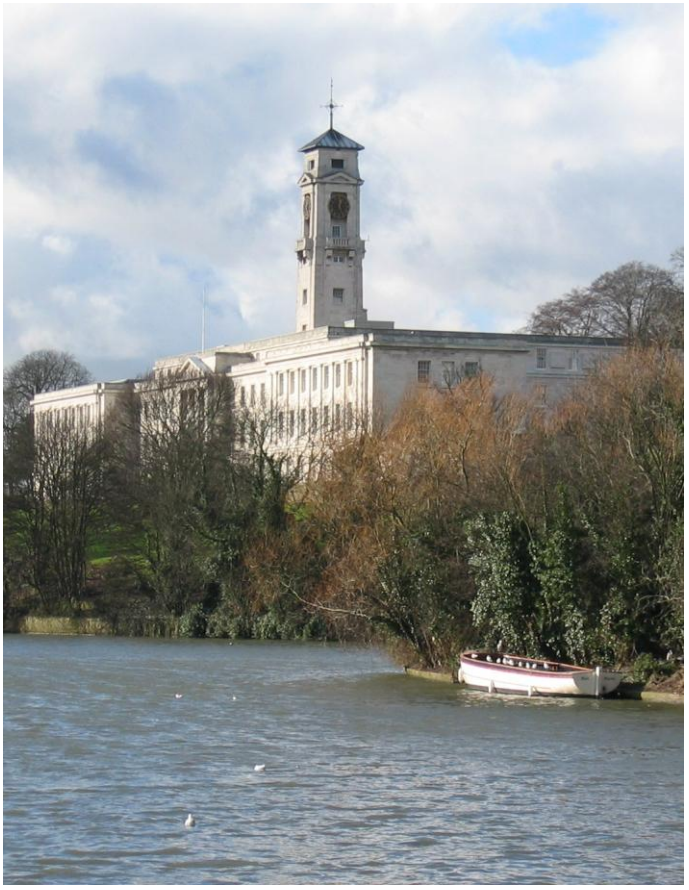
Proceedings of the

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools

University of Nottingham, UK, 19 April, 2013



Editor
Henrik Nilsson

Supported by The Functional Programming Laboratory, School of Computer Science, University of Nottingham

ISSN: 1650-3686

EOOLT 2013

5th International Workshop on

Equation-Based Object-Oriented Modeling Languages and Tools

19 April 2013, Nottingham, UK

Proceedings

Edited by Henrik Nilsson

Series: Linköping Electronic Conference Proceedings, No. 84
ISSN (print): 1650-3686
ISSN (online): 1650-3740
ISBN (print): 978-91-7519-621-3
ISBN (online): 978-91-7519-617-6
`http://www.ep.liu.se/ecp_home/index.en.aspx?issue=084`

Printed by AlphaGraphics, Nottingham, UK, 2013

Chair's Welcome

It is my great pleasure to welcome you to EOOLT 2013, the 5th International Workshop on Equation-Based Object-Oriented Languages and Tools! Equation-based modeling and simulation languages with hybrid capabilities (that is, supporting both continuous-time and discrete-time aspects) enable high-level reuse and integrated modeling capabilities for physical systems, embedded systems software, as well as their combination. They thus offer considerable advantages for many application areas, including complex cyber-physical systems. Consequently, this class of languages has gained significant and increasing attention over the last decade. Examples include Modelica, SysML, VHDL-AMS, and Simulink/Simscape. EOOLT is a forum for researchers with interests in all aspects of equation-based modeling languages and their supporting tools, including design, implementation, open issues limiting their expressiveness or usefulness, novel applications, and their relation to other approaches broadly addressing similar needs, such as synchronous and actor-oriented languages.

EOOLT 2013 takes place in Nottingham, UK, 19 April, hosted by the School of Computer Science at the University of Nottingham. It follows on from a successful series of earlier EOOLT workshops that took place in Berlin, Germany in 2007; Paphos, Cyprus in 2008; Oslo, Norway in 2010; and Zürich, Switzerland in 2011. For further general information about EOOLT, see `http://www.eoolt.org`.

In all, 13 papers were submitted and ultimately accepted for presentation after thorough peer reviewing. Each paper received at least 3 independent reviews. The full, final versions of these papers can all be found in this volume, along with abstracts for the invited talk and two tool demonstrations. I believe the result is a very exciting and strong program for EOOLT 2013!

As always, making an event like EOOLT 2013 happen is very much a team effort. First of all, I would like to thank the authors for their contributions: without you, there would not be any EOOLT 2013 in the first place! Then I would like to thank the Program Committee and the additional reviewers who through thorough reviewing and engaged discussions set very high standards for the accepted contributions. The Steering Committee provided timely and helpful advice and other support throughout the organisational effort. I am particularly indebted to David Broman for providing various templates for the proceedings and the website. I would further like to thank Peter Berkesand and Linköping University Electronic Press for their invaluable help in putting together and publishing the proceedings. As to the local organisation, I owe a big thank you to John Capper and Nadine Holmes for help with countless practical arrangements. Additionally I gratefully acknowledge the support for EOOLT 2013 offered by Prof. Graham Hutton and the Functional Programming Laboratory, as well as the School of Computer Science for hosting the event and providing administrative support.

Henrik Nilsson (Chair)
Nottingham, March 2013

Table of Contents

EOOLT 2013 Organisation vi

Session I: Verification and Validation
Chair: Henrik Nilsson *(University of Nottingham)*

- Invited Talk: Enclosing Hybrid Behavior 3
  Walid Taha *(Halmstad University and Rice University)*
- Static Validation of Modelica Models for Language Compliance and Structural Integrity 5
  Roland Samlaus and Mareike Strach *(Fraunhofer Institute for Wind Energy and Energy System Technology)*
- Modeling System Requirements in Modelica: Definition and Comparison of Candidate Approaches 15
  Andrea Tundis *(University of Calabria)*, Lena Rogovchenko-Buffoni *(Linköping University)*, Peter Fritzson *(Linköping University)*, and Alfredo Garro *(University of Calabria)*

Session II: Parallel Simulation
Chair: Dirk Zimmer *(DLR Oberpfaffenhofen)*

- Parallelization Approaches for the Time-Efficient Simulation of Hybrid Dynamical Systems: Application to Combustion Modeling 27
  Abir Ben Khaled *(IFP Energies nouvelles)*, Mongi Ben Gaid *(IFP Energies nouvelles)*, Daniel Simon *(INRIA and LIRMM)*
- Automating Dynamic Decoupling in Object-Oriented Modelling and Simulation Tools 37
  Alessandro Vittorio Papadopoulos and Alberto Leva *(Politecnico di Milano)*
- A Strategy for Parallel Simulation of Declarative Object-Oriented Models of Generalized Physical Networks 45
  Francesco Casella *(Politecnico di Milano)*

Session III: Diagnosis and Debugging
Chair: Peter Fritzson *(Linköping University)*

- Functional Debugging of Equation-Based Languages 55
  Arquimedes Canedo and Ling Shen *(Siemens Corporation)*
- Toward an Equation-Oriented Framework for Diagnosis of Complex Systems 65
  Alexander Feldman and Gregory Provan *(University College Cork)*

Session IV: Simulation Methods
Chair: Francesco Casella *(Politecnico di Milano)*

- Using Artificial States in Modeling Dynamic Systems: Turning Malpractice into Good Practice 77
  Dirk Zimmer *(German Aerospace Center (DLR))*
- Simplification of Differential Algebraic Equations by the Projection Method 87
  Elena Shmoylova, Jürgen Gerhard, Erik Postma, and Austin Roche *(Maplesoft)*
- Initialization of Equation-Based Hybrid Models within OpenModelica 97
  Lennart A. Ochel and Bernhard Bachmann *(University of Applied Sciences, Bielefeld)*

Session V: Other Topics

Chair: Walid Taha *(Halmstad University and Rice University)*

- Tool Demonstration Abstract: OpenModelica and CasADi for Model-Based Dynamic Optimization 107
  Alachew Shitahun *(Linköping University)*, Vitalij Ruge *(Univ. of Applied Sciences, Bielefeld)*, Mahder Gebremedhin *(Linköping University)*, Bernhard Bachmann *(Univ. of Applied Sciences, Bielefeld)*, Lars Eriksson *(Linköping University)*, Joel Andersson *(K.U. Leuven)*, Moritz Diehl *(K.U. Leuven)*, and Peter Fritzson *(Linköping University)*
- Tool Demonstration Abstract: OpenModelica Graphical Editor and Debugger 109
  Adeel Asghar and Peter Fritzson *(Linköping University)*
- Modelica on the Java Virtual Machine 111
  Christoph Höger *(Technische Universität Berlin)*
- An Approach to Cellular Automata Modelling in Modelica 121
  Victorino Sanz and Alfonso Urquia *(ETSI Informtica, UNED)*
- Models for Distributed Real-Time Simulation with Vehicle Co-Simulator Setup 131
  Anders Anderson *(Swedish National Road and Transportation Institute)* and Peter Fritzson *(Linköping University)*

EOOLT 2013 Organisation

Chair:	Henrik Nilsson	University of Nottingham
Steering Committee:	David Broman	UC Berkeley and Linköping University
	François Cellier	ETH Zürich
	Peter Fritzon	Linköping University
	Edward A. Lee	UC Berkeley
Local Arrangements:	John Capper	University of Nottingham
	Nadine Holmes	University of Nottingham
	Henrik Nilsson	University of Nottingham
Program Committee:	Bernhard Bachmann	University of Applied Sciences, Bielefeld
	Bert van Beek	Eindhoven University of Technology
	David Broman	UC Berkeley and Linköping University
	Francesco Casella	Politecnico di Milano
	François Cellier	ETH Zürich
	Olaf Enge-Rosenblatt	Fraunhofer Institute, Dresden
	Peter Fritzson	Linköping University
	Michaela Huhn	Clausthal University of Technology
	Edward A. Lee	UC Berkeley
	Pieter Mosterman	MathWorks
	Ramine Nikoukhah	INRIA Rocquencourt and Altair
	Henrik Nilsson (Chair)	University of Nottingham
	Chris Paredis	Georgia Institute of Technology
	Peter Pepper	TU Berlin
	Walid Taha	Halmstad University and Rice University
	Alfonso Urquia	UNED, Madrid
	Hans Vangheluwe	McGill University and University of Antwerp
	Justyna Zander	MathWorks and Gdansk University of Technology
	Dirk Zimmer	DLR Oberpfaffenhofen
Additional Reviewers:	Christoph Höger	TU Berlin
	Alexandra Mehlhase	TU Berlin

Session I: Verification and Validation

Invited Talk: Enclosing Hybrid Behavior

Walid Taha

Halmstad University, Sweden and Rice University, USA
Walid.Taha@hh.se

Abstract

Rigorous simulation of hybrid systems relies critically on having a semantics that constructs enclosures. Edalat and Pattinson's work on the domain-theoretic semantics of hybrid systems almost provides what is needed, with two exceptions.

First, domain-theoretic methods leave many operational concerns implicit. As a result, the feasibility of practical implementations is not obvious. For example, their semantics appears to rely on repeated interval splitting for state space variables. This can lead to exponential blow up in the cost of the computation.

Second, common and even simple hybrid systems exhibit Zeno behaviors. Such behaviors are a practical impediment because they make simulators loop indefinitely. This is in part due to the fact that existing semantics for hybrid systems generally assume that the system is non-Zeno.

The feasibility of reasonable implementations is addressed by specifying the semantics algorithmically. We observe that the amount of interval splitting can be influenced by the representation of function enclosures. Parameterizing the semantics with respect to enclosure representation provides a precise specification of the functionality needed from them, and facilitates studying their performance characteristics. For example, we find that non-constant enclosure representations can alleviate the need for interval splitting on dependent variables.

We address the feasibility of dealing with Zeno systems by taking a fresh look at event detection and localization. The key insight is that computing enclosures for hybrid behaviors over intervals containing multiple events does not necessarily require separating these events in time, even when the number of events is unbounded. In contrast to current methods for dealing with Zeno behaviors, this semantics does not require reformulating the hybrid system model specifically to enable a transition to a post-Zeno state. The new semantics does not sacrifice the key qualities of the original work, namely, convergence on separable systems.

Keywords hybrid systems semantics, hybrid systems implementation, Zeno behavior, event localization

Acknowledgments

Joint work with Michal Konecny (Aston), Jan Duracz (Halmstad), and Aaron Ames (Texas A&M).

Biography

Walid Taha is a Professor of Computer Science at Halmstad University. He is interested in the design, semantics, and implementation of programming and hardware description languages. His current research focus is on modeling, simulation, and verification of cyberphysical systems, and in particular the Acumen modeling language.

Taha is credited with developing the idea of multi-stage programming (or "staging" for short), and is the designer of several systems based on it, including MetaOCaml, ConCoqtion, Java Mint, and the Verilog Preprocessor. He contributed to several other programming languages innovations, including statically typed macros, tag elimination, tagless staged interpreters, event-driven functional reactive programming (E-FRP), the notion of exact software design, and gradual typing. Broadly construed, his research interests include cyberphysical systems, software engineering, programming languages, and domain-specific languages.

Taha was the principal investigator on a number of research awards and contracts from the National Science Foundation (NSF), Semi-conductor Research Consortium (SRC), and Texas Advanced Technology Program (ATP). He received an NSF CAREER award to develop Java Mint. He founded the ACM Conference on Generative Programming and Component Engineering (GPCE), the IFIP Working Group on Program Generation (WG 2.11), and the Middle Earth Programming Languages Seminar (MEPLS). Taha chaired the 2009 IFIP Working Conference on Domain Specific Languages.

According to Google Scholar, Taha's publications had over 2,400 citations and an h-index of 26.

Prof. Taha holds an Adjunct Professor position at Rice University.

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. 19 April, 2013, University of Nottingham, UK.

 Proceedings available at:
http://www.ep.liu.se/ecp_home/index.en.aspx?issue=084
EOOLT 2013 website:
http://www.eoolt.org/2013/

Static Validation of Modelica Models for Language Compliance and Structural Integrity

Roland Samlaus Mareike Strach

Turbine Simulation, Software Development and Aerodynamics
Fraunhofer Institute for Wind Energy and Energy System Technology, Germany
{roland.samlaus,mareike.strach}@iwes.fraunhofer.de

Abstract

The increasing importance of the simulation of physical systems models demands enhanced support for developers. Models do not only increase in terms of quantity, but also complexity. Hence, libraries need to be created containing valid models for re-use. It is crucial for library developers to get immediate feedback about errors regarding the language specification. Moreover, users of libraries need to know immediately if existing components are misused.

When using Modelica as the modeling language the models are validated at compilation time by recent development environments. This decreases the development speed as developers recognize errors in their models late and therefore need to recapitalize the design decisions made in order to maintain the intent of the code during error fixing.

In this paper we present two implementations, i.e. Object Constraint Language (OCL) and Java, for Modelica code validation that can be triggered during model editing. Both variants are compared to each other regarding readability of constraints as well as execution performance. Therefore, rules are extracted from the Modelica language specification asserting that the models are correct. Furthermore, custom rules are defined restricting library models such that they can only be used in the intended way.

Keywords Modelica model validation, static source code analysis, constraint languages

1. Introduction

In the past years the open modeling language Modelica has become widely used by engineers for physical model development. The benefit of an open language approach is that the user is not dependent on a single tool vendor and can influence the further development of the language standard. The development of the language was accompanied by tool vendors, providing environments accelerating the development process compared to plain text editors. Furthermore, libraries are being developed, open source as well as proprietary, enabling re-use of components for further model design.

At the Fraunhofer Institute for Wind Energy and Energy Systems Technology (IWES), a development environment is being developed, aimed at extensive support for physical model developers. The application of modern model driven technologies allows one to quickly create an Integrated Development Environment (IDE) with advanced tools support for Modelica users. Utilizing the popular Eclipse Modeling Framework (EMF) [1] is advantageous since additional tools built for the Framework can directly be used. As an example, implementations of Object Constraint Language (OCL) interpreters are available, supporting the interpretation of OCL constraints on any language whose meta model is based on EMF.

The development environment is used by engineers at Fraunhofer IWES to create a Modelica library for wind turbines [11]. The demand for immediate validation[1] arose due to the extent of the library. Changes in a model can affect others, but since full validation is only done during simulation, errors are often detected late and therefore the reason may not be obvious any more. Subsequent fixes can contain further semantic errors leading to more design iterations. The errors discussed here are caused by violation of rules defined by the Modelica language specification. Since models are usually composed of various components and the extension of components is allowed and desired, validation of models is often expensive. Models concerning a wide range of extended models need to be flattened in order to be validated. Hence, validation of models needs to be performed sufficiently well. Additionally, fast constraints need to be distinguished from slow constraints in order to be checked by separate triggers - expensive constraints may only be checked when the user saves the edited document or even may need to be triggered by hand, while fast constraints can be checked during editing. For the checks existing compilers could be employed, but since we aim at being independent from any third party tool, this solution

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. 19 April, 2013, University of Nottingham, UK.

[1] All validations and constraints described in this paper target the correctness regarding the Modelica language specification or Modelica framework design. It is not intended to validate or to constraint physical models.

is not applicable. Additionally, the Modelica code needs to be parsed and the models linked by any tool used, which would lead to performance loss and increased consumption of resources and may cause delayed error feedback during editing.

Another aspect of model validation appeared at Fraunhofer IWES, especially for engineers unfamiliar with Modelica. Many errors are made by combining predefined components, e.g. from a library, that physically do not fit together. This is because Modelica is a language for mathematical modeling and thus does not restrict the use for domain specific design aspects. There is no support yet for the definition of semantic rules regarding the combination of components, which is of special interest for frameworks. At the moment, two arbitrary Modelica components can be connected to each other, no matter if they fit, as long as the connector types match. This can either lead to a model that cannot be simulated — e.g. because it is structurally singular — or to a phyiscally incorrect model. In this case, the error messages do not provide sufficient help for the user as the source of errors are on a physical level that cannot be captured by library unspecific error messages.

This paper describes an approach for static model validation, enforcing the rules defined by the Modelica language specification. Previous work forming the basis for the validation is shown and important parts of the Modelica meta model definition needed to understand the discussed constraints are described. The rules of the Modelica language specification relate to the meta model of Modelica and thus need to be explicitly defined. Additionally, methods are described that were added to the meta model classes in order to simplify the access to model data, which is mainly used when flattening models. The constraint definition language OCL is introduced and a selection of constraints checking the conformance of models to the Modelica language specification is explained. Furthermore, the implementation of the constraints in Java is explained and compared to OCL regarding readability, reusability and performance. Further rules regarding establishing structural constraints are discussed and the implementation of a prototype with Java for wind turbine models is presented. Finally, conclusions are drawn and an outlook to future work is given.

2. Related Work

The Object Management Group (OMG) specifies meta modeling by the Meta Object Facility (MOF) [8] that is implemented as the EMF for Eclipse. MOF defines four layers (see Figure 1) of modeling where layer 0 represents the objects of the real world, e.g. physical systems like a wind turbine. Layer 1 (model) contains the models of real world objects, e.g. instances of Modelica classes representing the behavior of wind turbines. In Layer 2 (meta model), the structural properties of instances are described (i.e. concepts like Modelica classes, extend clauses or equations are described). Layer 3 (meta meta model) provides concepts for the definition of layer 2 elements.

When using EMF these concepts can be classes, references or attributes that are used to describe the structure of a Domain-Specific Language (DSL). Constraints are defined based on layer 2 and establish semantic rules, e.g. rules defined by the Modelica language specification. Additionally, rules can be user defined, e.g. to establish code styles or to prevent the combination of incompatible components.

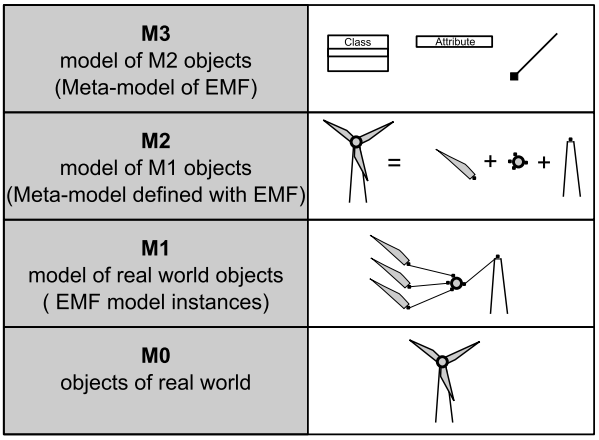


Figure 1. Layers of the Meta Object Facility

OneModelica [9] is a Modelica IDE that is implemented using Model Driven Software Development (MDSD). The validation aims at enriching the IDE and therefore builds on the technologies being provided by the IDE. Since Modelica documents are parsed and represented in a EMF based tree, the constraints can be checked directly on the tree. Thereby, it is possible to either interpret constraints defined in special constraint languages like OCL or to validate instances in the tree directly by general purpose languages like Java. Besides its use as an editor and a validation interface, the IDE provides views to the Modelica developer, helping to understand complex models by viewing information regarding certain aspects in a simplified way. This includes an outline view, displaying the structural content of a document, a documentation view, a hint display for the developer on how to use predefined models, and others. Furthermore, linking is implemented by connecting related documents to each other, e.g. by linking types of components to their respective class declarations. Linking is essential for the validation of Modelica models since many constraints define restrictions on the basis of inherited classes.

RestrictED [10] provides constraint checking with OCL for arbitrary DSLs defined with EMFText. Constraints are defined as queries, collecting all objects violating the constraint in the parsed tree of the document. The constraints can be defined by the user during runtime. However, defining queries instead of invariants can be misleading. Moreover, it is harder to define them such that all erroneous objects are collected. This limits the complexity of usable languages and only a few constraints were defined for the example languages.

ModIM [4] — a front-end tool for processing Modelica models — allows to statically analyze models based

on a syntax tree representation. For the analysis the visitor pattern [3] is applied and custom analysers can be implemented that visit the nodes of the syntax tree. A sample implementation is presented that performs a type check. However, the performance of the tool is not addressed, it is only stated that redeclarations perform badly with the current implementation.

In [6] an aspect based validation framework is presented that allows to define elements of the Modelica language that are involved in the validation (*join points*). The elements can be queried by an aspect language (*point cut expressions*) and an action language can be used to define what shall be done with the elements of the models (*advice*). The languages are defined by re-using paradigms of logic programming and can be transformed into a format that can be evaluated by Prolog. The provided examples show how custom rules like naming conventions and the number of classes defined inside a package can be checked.

For the validation of equation-based components focusing on numerical inconsistencies [2] proposes a graph-based methodology that provides users with information about under- and overconstraint equation systems. However, the general validity regarding the Modelica language specification is not aimed at.

3. Modelica Meta Model

The Modelica meta model that is the basis for the constraint definitions is described in this section. However, Xtext [5], which is used as the framework for the meta model definition, will not be explained for the sake of brevity. A short introduction describing the Modelica IDE [9] can be found in related work and on the project's web site[2]. The Modelica meta model structure will not be explained in detail, since it is quite complex. However, a simplified representation is displayed in Figure 2, giving an overview of the most important elements needed for the constraint definitions in Section 4.

The root element (`AstModelicaSourceFile`) of the language definition represents a stored Modelica program that can contain arbitrary class declarations and a statement (`within`) declaring in which package these class declarations are contained. Classes can again contain class declarations and other elements like `components`, `extends clauses` and `algorithm sections`.

Classes extending other classes inherit the declared components and the behavior defined by `equations` and `algorithms`. This makes the validation of objects harder since the inherited attributes need to be taken into account. Collecting all attributes and behavioral elements and merging them into one class representation is called *flattening*.

`Components` have a type defined by referencing a class declaration. Again, no restrictions are made by the syntax definition of the language. But this needs to be restricted since type compatibility must be enforced, especially when using equations. Here it is important to check whether the types fit in order to be able to do calculations. Furthermore, connect statements connect `components`

[2] http://www.onewind.de/OneModelica.html

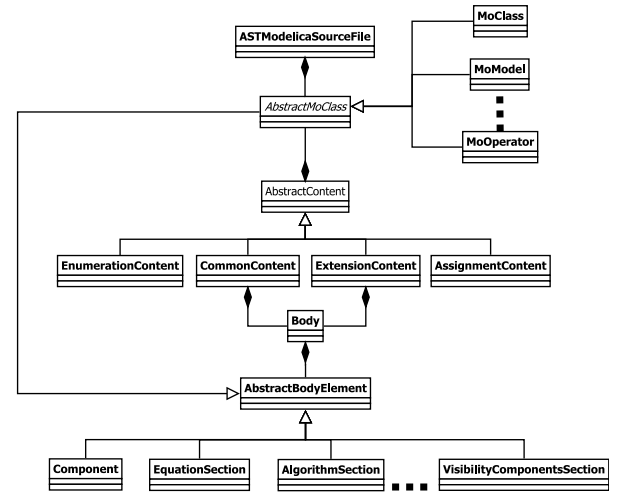


Figure 2. Simplified Modelica meta model

to each other that must be of type `connector` [7]. Components and subclasses are stored in a class `body` that is only present in classes where the content type is `CommonContent` or `ExtensionContent`. Classes with an `EnumerationContent` contain only enumerations while an `AssignmentContent` assigns a present class declaration to a new declaration. This is commonly used when new types are defined in order to make the code easier to understand (the `Modelica.SIunits` package defines new types like `Length` extending `Real` and setting the quantity to *Length* and the unit to *meter*).

The class concept in the grammar is not restrictive. Hence, every class type (class, package, function, record, block, ...) can contain other classes, algorithms and so on. The restrictions are defined in the Modelica language specification in textual form. Hence, constraints needed to be extracted and must be checked on instances of the meta model. By parsing a Modelica document and representing it as an Abstract Syntax Tree (AST), it is possible to check the semantic constraints on that representation.

Since the parsed tree can be large, querying is a bottle neck and makes the definition of constraints complex and error-prone. When using an interpreter like OCL, querying will significantly slow down the validation process. Since OCL performs badly when processing large tree-based data structures, methods for easier access to the elements of the AST were implemented with Java. Xtext provides the possibility of adding methods to elements of the meta model with the language Xtend. The methods that were added mainly implement flattening of classes, i.e. collecting all components defined in a class including its extended classes. With the help of the additional methods, the size of the constraints is significantly reduced and the readability enhanced. Details about the validation of Modelica models are discussed in Section 4.

4. Validating Modelica Language Specification Compliance

This section describes the static Modelica model validation regarding correctness as defined by the Modelica lan-

guage specification. Two methods of constraint definitions are compared and implemented with both OCL and Java. The two approaches are compared to each other in order to check whether dedicated languages for constraint definition, like OCL, are better suited regarding readability and re-usability then general purpose languages like Java. Finally, the performance loss caused by interpretation of OCL constraints on models compared to Java is analyzed. The constraints that are validated are arbitrarily chosen to reflect a wide range of constraint types of the specification. Type checking is not performed at the moment since the specification is not clear in any points and the effort for the implementation is high as first attempts showed. Nevertheless, type checking should be possible and will be addressed in future work.

4.1 OCL Constraints

OCL was initially designed by the OMG to constrain Unified Modeling Language (UML) diagrams. The standard was later extended and can now be used with various meta modeled languages. Invariants can be specified, checking if a condition is met by a `context` object (the object being validated). Furthermore, queries can be defined for collecting and analyzing structured data. OCL is also used in transformation languages for the definition of transformation rules between two meta models. In this work, constraints are defined by invariants that sometimes make use of queries in order to collect elements of the Modelica AST. Three OCL constraints are explained, enforcing correctness of Modelica models for the following rules defined by the language specification [7] (numbers in parentheses denote the page of the definitions):

- Operators may only be placed in an operator record or in a package inside an operator record (42)
- A function can have at most one algorithm section (135)
- A stream connector must have exactly one scalar variable with the flow prefix (175)

The three constraints are used because they present different kinds of constraints that a) check in which program part the context object can be used, b) analyze what language elements the context object is allowed to define inside its content and c) check whether conditions are met that need to be fulfilled when a conditional aspect is met.

```
context MoOperator
 inv operator_only_in_record_or_package:
 let cls: AbstractMoClass = getAbstractMoClass()
 in
   not cls.oclIsUndefined() and
   (cls.oclIsKindOf(MoRecord) and
    cls.oclAsType(MoRecord).operator)
  or
   (cls.oclIsKindOf(MoPackage) and
    cls.parentIsOperatorRecord())
```

Listing 1. OCL constraint restricting the use of operators

The first constraint (Listing 1) analyzes whether an operator is located in a permitted enclosing class. The language specification defines that an `operator` can only be declared inside an `operator record` or inside a `package` that itself is defined in an operator record. Therefore, declaring an operator elsewhere (e.g. inside a function) is not allowed. The context object of this constraint is defined by the keyword `context` and is only applied to objects of the stated type. An invariant is defined by the keyword `inv` and assigns a unique name to the invariant that is later also used as an identifier to retrieve a comprehensive error message in case of a violation. A local variable is defined by the keyword `let` and, in this constraint, represents an object of type `AbstractMoClass`.

```
context AbstractMoClass
 def: parentIsOperatorRecord(): Boolean =
  not getAbstractMoClass().oclIsUndefined()
  and getAbstractMoClass().oclIsKindOf(MoRecord)
  and getAbstractMoClass()
          .oclAsType(MoRecord).operator
```

Listing 2. OCL helper method for checking whether a class is a operator record

The method `getAbstractMoclass()` is implemented with Java, as described in Section 3, and returns the enclosing class of an AST element. In the case that the operator is defined in the top level of a document the object may be `null`. Hence, a check by the built-in OCL function `oclIsUndefined()` needs to be performed resulting in an error displayed indicating that the restriction is not met. The next part of the OCL constraint validates whether the enclosing class is of type `MoRecord` and whether the operator keyword is used for that instance. Another allowed use of the operator is when the enclosing type is a `MoPackage` and the package's parent is defined inside an operator function. This is implemented by a separate function defined using OCL (Listing 2).

The second constraint, as shown in Listing 3, analyses whether a declaration of a `MoFunction` contains at most one `AlgorithmSection`. The syntax definition in the Modelica language specification makes no distinction between the different class concepts. Hence, the definition of several algorithm sections inside a function syntactically conforms to the Modelica specification and thus is not marked as an error by the parser. However, this is semantically incorrect and must be prevented.

```
context MoFunction
 inv function_no_multiple_algorithms:
 let b: Body =
  if content.oclIsKindOf(CommonContent) then
   content.oclAsType(CommonContent)
          .getContentsBody()
  else
   content.oclAsType(ExtensionContent)
          .getContentsBody()
  endif
 in
  b.oclIsUndefined() or
  b.bodyelements->select
  (oclIsKindOf(AlgorithmSection))->size()<2
```

Listing 3. OCL constraint restricting functions to have at most one algorithm section

For validation, the body of the context object is selected. Bodies are only available in type `CommonContent` and `ExtensionContent` (see Section 3). If no body is avail-

able (checked by `b.oclIsUndefined()`), the constraint can not be violated. If a body is available, all elements of type `AlgorithmSection` are selected from the list of elements defined inside the body. This is done by the built-in OCL operation `select`. It can be applied to collections and selects all elements satisfying a user defined boolean expression. Here, every object of the list is analyzed to be of kind `AlgorithmSection`. In case the resulting collection contains more then one object, the constraint is violated.

```
context MoConnector
 inv stream_connector_exactly_one_flow:
 let components: Collection(Component)
       = getAllComponents()
 in
   components->exists(cIsStream()) implies
   (components->forAll(cIsFlow() implies
    componentnames->size() = 1)
  and
   components->select(cIsFlow())->size() = 1)
```

Listing 4. OCL constraint checking whether a stream connector has exactly one scalar variable with the flow prefix

The third OCL constraint (Listing 4) checks whether a stream connector has exactly one scalar variable with flow prefix. A stream connector is a class of type `connector` defining a component (of type Real or an extension of type Real) that is prefixed with the keyword `stream`. In this case exactly one component with the keyword `flow` must be present inside the class declaration. The constraint first collects all components of the connector object inside the variable `components`. For convenience, this method (`getAllComponents()`) again is implemented in Java, since it is used frequently and thus an inefficient implementation may lead to slow processing of the constraints.

After retrieving all components it is checked whether any of the components is a `stream` variable. In this case, all components that are flow variables (`cIsFlow()`) are analyzed if they define exactly one variable name. The count of variable names needs to be taken into account since multiple components can be defined by a list expression in Modelica. Additionally, all components that are flow variables are selected in order to assure that the size of the resulting collection is exactly 1. The OCL helper functions `cIsStream()` and `cIsFlow()` are defined in Listing 5.

```
context Component
def: cIsFlow(): Boolean =
 (not connectorprefix.oclIsUndefined())
  and connectorprefix.isFlow()
def: cIsStream():Boolean =
 (not connectorprefix.oclIsUndefined())
  and connectorprefix.isStream()
context ConnectorPrefix
def: isFlow(): Boolean =
 (not value.oclIsUndefined()) and
  value = 'flow'
def: isStream(): Boolean =
 (not value.oclIsUndefined()) and
  value = 'stream'
```

Listing 5. OCL helper methods checking whether a component is a flow or stream variable

4.2 Java Constraints

To compare the performance of the interpreted language OCL to a general purpose language, the constraints are also defined with Java. The structure of the constraints is somehow comparable. However, it is possible to optimize the performance, since the use of local variables and conditional return statements can be used. A Java constraint that has a similar structure as the corresponding OCL definition is displayed in Listing 6.

```
public boolean isValid(EObject eObject) {
    MoOperator operator = (MoOperator) eObject;
    AbstractMoClass enclosingClass = operator.
        getAbstractMoClass();
    if (enclosingClass != null) {
      if (enclosingClass instanceof MoRecord
          && ((MoRecord) enclosingClass).
                isOperator()) {
        return true;
      }
      if (enclosingClass instanceof MoPackage) {
        enclosingClass = enclosingClass.
                  getAbstractMoClass();
        if (enclosingClass == null) {
          return false;
        }
        if (enclosingClass instanceof MoRecord
            && ((MoRecord) enclosingClass).
                    isOperator()) {
          return true;
        }
      }
      return false;
    }
    return false;
  }
```

Listing 6. Java constraint checking whether a stream connector has exactly one scalar variable with the flow prefix

In contrast, Listing 7 displays a constraint that benefits from the additional language constructs of Java.

```
  public boolean isValid(EObject eObject) {
    MoConnector conn = (MoConnector) eObject;
    boolean isStream = false;
    int numberScalar = 0;
    for (Component component :
      conn.getAllComponents()) {
      if (isStream && numberScalar > 1) {
        // break if too many scalar variables
        // are defined
        return false;
      }
      if (cIsStream(component)) {
        isStream = true;
      } else {
        if (cIsFlow(component)) {
          numberScalar += component.
              getComponentnames().size();
        }
      }
    }
    if (isStream && numberScalar != 1) {
      return false;
    }
    return true;
  }
```

Listing 7. Optimized Java constraint checking the stream connector restriction

The iteration over components of a class can be interrupted when a violation is detected since the number of scalar components can be checked every iteration. The collection of components needs to be iterated only once since both conditions can be checked inside the loop. Hence, they are checked to see whether a component is `stream` and as soon as more than one scalar variable is found, a violation of the constraint is indicated. The methods `cIsStream()` and `cIsFlow()` are implemented with Java checking for the connection type of the component similar to the OCL functions previously defined.

4.3 Language Concept Comparison

When the constraints are compared, it is obvious that the readability of OCL constraints is very good for rules that can be defined in a short form. The stated context makes it obvious which object type is being constrained. Local variables can be defined and used for the validation of the context object. Functions allow the definition of re-usable common functionality. Multiple constraints can be defined in the same file allowing the accumulation of constraints targeting the same context object in one document.

On the other hand, Java as the constraint language can be understood by more software developers. Furthermore, object oriented development is more common to developers than the functional programming representation of OCL. It is possible to define local variables inside the constraint definition causing validation to be quicker, since checks can be done inside loops to return a validation result early.

The most obvious benefit of Java as a constraint language compared to OCL is the tool support. Although OCL editors exist, they mostly lack support for automatic refactoring and robust referencing or only support syntax highlighting. But these features become vital when the meta model of a language that is being constrained is altered.

Hence, it may be beneficial to use OCL constraints for the restriction of languages, where the grammar definition is finally set. On the other hand, constraints defined with Java may be a better solution when the grammar is still under development or high performance is required when validating models.

4.4 Performance

When validation takes place while a user develops a model, high performance is vital. In this section, all constraints defined using OCL are compared to their equivalent constraints defined with Java. The performance time is compared by validating the Modelica standard library[3], which contains models for various fields of physical modeling. The library contains all sorts of language constructs defined by the language specification and hence provides good feedback regarding the performance of the constraints. The included models are complex and frequently extend classes, causing more effort in resolving references and elements that need to be considered during the validation.

The Modelica standard library used for the performance measurement is version 3.2 beta 5. It consists of:

- 1432 Functions
- 1282 Models
- 694 Types
- 651 Packages
- 302 Blocks
- 289 Classes
- 278 Records
- 108 Connectors
- 3 Operators

This includes the definitions of base types like `Real`, `Integer` or `Complex` that are actually not included in the library but are added for convenience in our IDE. Parsing all 220 files takes approximately 6200 ms. Linking (resolving references, e.g. references between used components and their declaration) is done in about 14000 ms. This may be enhanced in the future since the linking mechanism is not optimal regarding performance at the moment. Table 1 contains the measured performance results of 10 of 37 available constraints in both OCL and Java. The number of calls and the execution time for the invoked constraints are stated. The bottom line displays the number of calls and the overall performance for all 37 constraints. The validation was performed on a computer with an Intel Core I7 870 CPU (4 cores, max 2.93 GHz) with 8 GB of RAM.

As we can see from the results, there is a tremendous performance difference between both kinds of constraints. This would be even worse if the flattening of classes was performed by OCL instead of using the helper methods implemented in Java as mentioned in Section 3.

The differences in performance originates in the higher efficiency of Java when handling collections. This is particularly clear when investigating the performance of the constraint `unique_element_names_comp` since the implementations have to iterate over lists of components and compare the names in order to check whether a name has been used multiple times. This validation is performed for each of the 5039 types of classes. Since the check needs to be done for flattened classes to check if an extended class already defines a component with the same name, the number of components can be very high.

For less extensive constraints in which few context objects were called, OCL performs sufficiently. The constraint `function_no_multiple_algorithms` that validates the 1432 functions of the Modelica standard library (defined in Section 4) takes only 77 ms. But even for this simple task, Java performs almost 4 times faster. The performance comparison shows the drawback of an interpreted language compared to a native one. Overhead caused by the interpretation and, in the case of OCL, the lack of efficient collection handling and early returns on violation of invariants reduces the performance.

With the fact that about 200 restrictions were found in the Modelica language specification, it is obvious that the performance will not be sufficient for automatic validation during editing with the recent interpreter implementation for OCL included in the Eclipse MDT project. Even for

[3] https://modelica.org/libraries/Modelica

Constraint	OCL Calls	OCL Time (ms)	JAVA Calls	JAVA Time (ms)
unique_element_names_comp	5039	9896	5039	140
protected_variables_dot_reference	225240	3842	225240	731
prefixes_structured_component_flow	22186	97	22186	3
function_no_multiple_algorithms	**1432**	**77**	**1432**	**15**
flow_subtype_of_real	22186	76	22186	47
stream_only_in_connector	22186	28	22186	15
stream_connector_exactly_one_flow	**108**	**2**	**108**	**0**
function_no_equations	1446	1	1446	0
nested_when_equations	35	0	35	0
operator_only_in_record_or_package	**3**	**0**	**3**	**0**
...				
	94677	**20063**	946774	3330

Table 1. OCL and Java validation performance of selected constraints

Java, if the user does not want to be disturbed by long lasting validations taking place while editing Modelica models, focus on high performance is needed during further implementations of the missing constraints.

5. Validation of Structural Constraints

As mentioned before, the validation of constraints that originate from the physical properties of the model is very beneficial for the user. In the following section, existing components of the OneWind Modelica library according to [11] are used to introduce these kind of constraints. If library developers would provide domain-specific constraints for their models, the intended use of the models could be enforced. By standardising the constraint definition, e.g. by providing OCLconstraints inside annotations, all Modelica tools could benefit from the added semantics. However, for the proposed solution the tools must be able to process the same meta model of Modelica, since the constraints are defined based on the AST representation of Modelica models.

5.1 Possible Model Structures for a Horizontal-Axis Wind Turbine

Using the OneWind Modelica libary, a conventional three-bladed, horizontal axis wind turbine can be modeled through parametrizing and connecting components that have a specific physical meaning. The structure of such a model is displayed in Figure 3.

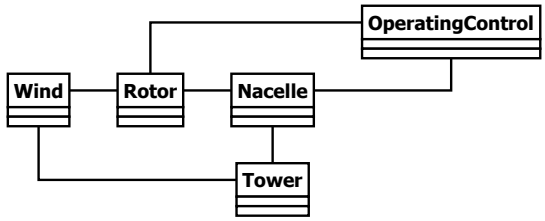


Figure 3. Model structure of a horizontal axis wind turbine using OneWindModelica library components

As can be seen in Figure 3, the `Rotor` object is connected to the `Nacelle` object, which is then connected to the `Tower` object. However, a user of the library could connect the `Rotor` object to the `Tower` object directly. In both objects `connector` instances are used which makes this connection correct according to the Modelica language specification. From an engineering perspective, however, this is not reasonable. Furthermore, the components of the OneWind Modelica libary are not set up to cover this case, although there is always the possibility to modify the library components correspondingly.

If the user made this direct connection of `Rotor` and `Tower` object with the existing library components, the simulation of the model would result in an error due to a division by zero in the tower shadow calculation. However, this error only shows up when the model is already compiled. This takes valuable time from the model development. Depending on how experienced the user is in the field of wind energy, the given error message does not even give a direct hint on the true source of the error.

5.2 Introduction of Structural Constraints

To support the library user during the model development, a constraint that gives a warning can be defined to avoid the direct connection of `Rotor` and `Tower` object. This is realized utilizing Java. For the sake of brevity, only the constraint of the `Tower` object needing to be connected to the `Nacelle` object is covered. The library developer can define this constraint himself using simple annotations directly in the Modelica code during the development of library components.

The definition as well as the appearance of the constraint in the GUI of OneModelica is shown in Figure 4. The library developer defines the `topFrame` of type `Frame_b` in the `PartialTower` model as a restricted element through the annotation shown in the upper right side of Figure 4. If the user tries to connect the `Tower` object to a `Rotor` object as it is shown in the upper left side of Figure 4, an error message will appear (cf. bottom part of Figure 4). The connection of the `Tower` object to the `Nacelle` object does not result in this error.

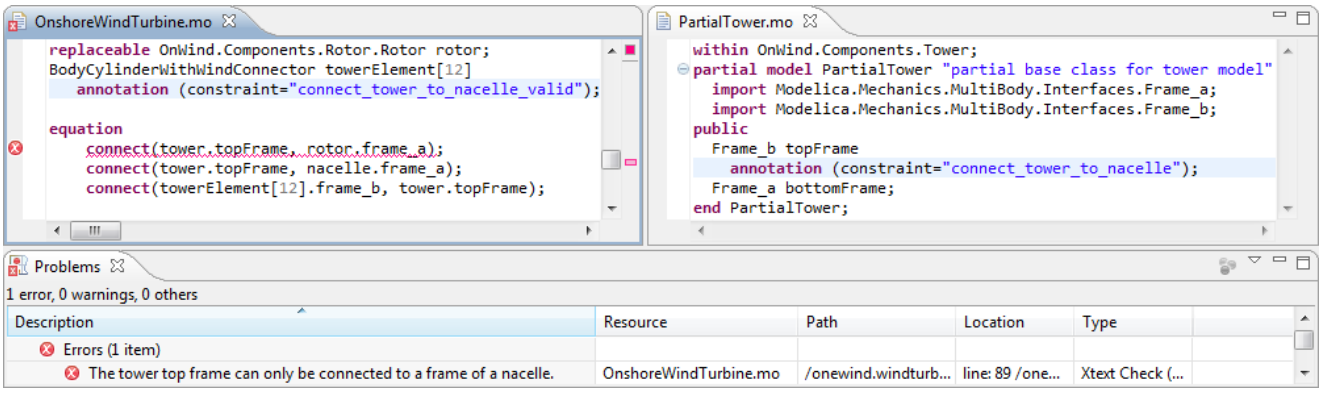


Figure 4. Structural constraint for connection of `Tower` and `Nacelle` object

However, if the user decides that he would like to connect an object to the `Tower` object that is not of type `Nacelle`, he can avoid the error message by defining another annotation. This is shown also in the upper left part of Figure 4. In this way, the user is not prohibited from using the library components to his wishes. But the constraint helps especially unexperienced users to avoid mistakes which is of high importance.

For this kind of constraints following steps needed to be performed. For the context type `ConnectClause` a Java validator has been registered. The validator then checks whether one of the connected elements defines a constraints inside its annotation (*constraints="constraintname"*). If this is true, the annotation of the other connected element is queried and it is checked whether it defines a valid constraint accordingly (*constraints="constraintname_valid"*). If the constraint does not exist, an error marker is created.

6. Conclusion and Future Work

This work shows that the validation of Modelica models is possible by the definition of constraints that check models on the basis of their tree based representation (AST). The constraint language OCL and Java are utilized for the validation. It becomes clear that many constraints can also be checked in an efficient way. By implementing the flattening of classes with Java, the extension of the constraint definition is reduced and the validation process accelerated. The integration of the validation is possible and can enhance the development of Modelica models heavily, since errors can be immediately displayed to the user.

However, we are able to point out that the interpretation of OCL constraints is time consuming, although the AST access has been enhanced. Therefore OCL should only be used for fast constraints. Since Java is up to 6 times faster when validating all currently available constraints, it is advantageous to implement the remaining constraints using Java in order to keep the performance fast. Another way to gain performance could be achieved by transforming the OCL constraints to Java code [12]. This would however outweigh the benefit of the interpreted language that constraints can be defined and checked during runtime.

Beyond validation against the Modelica language specification, structural constraints can be checked. This allows developers to define restrictions preventing errors that are obvious to library designers but that may lead to problems when done by library users. The approach may also allow the developer to restrict the usage of components that are known not to be compatible but can not be restricted by the modeling language itself.

In future work, more constraints will be implemented for the validation against the Modelica language specification. If all 200 constraints found so far can be implemented in an efficient way, validations can be recognized immediately by the developer. Furthermore, compatibility for various simulators can be established more easily since violations of the language specification that are accepted by some simulators can be identified easily.

The access to the AST maybe further enhanced by adding additional methods. Furthermore the introduction of caching, e.g. by remembering all base types of a class, which is necessary for fast type checking, would accelerate the validation with a reasonably cost of memory. This however would further reduce the complexity of constraint definition and may enable the use of OCL which seems to be more promising if e.g. library providers shall define constraints on how their models can be used.

The idea of further structural restrictions is promising and can enhance the definition of Modelica libraries. Besides the correct use of components, constraints may also be used to identify possible combinations of components, since the selection of usable items is reduced. This may help the user to find suitable solutions more efficiently by selecting components that are suggested by the IDE. Constraints may be defined for specific design aspects. Additionally, rules for connections may need to be expressed, e.g. regarding the cardinality of connections. An implementation based on role models is currently developed and will simplify the definition of structural constraints in the future.

References

[1] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[2] Peter Bunus and Peter Fritzson. Automated static analysis of equation-based components. *Simulation*, 80(7-8):321–345, 2004.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Amsterdam, 1 edition, 1995. 37. Reprint (2009).

[4] Christoph Höger. Modim - a modelica frontend with static analysis. In *MATHMOD 2012 - 7th Vienna International Conference on Mathematical Modelling*, 2012.

[5] Jan Köhnlein and Sven Efftinge. Xtext 2.1 documentation, October 31, 2011.

[6] Malte Lochau and Henning Günther. A static aspect language for modelica models. In Peter Fritzson, FranÃ§ois E. Cellier, and David Broman, editors, *EOOLT*, volume 29 of *Linköping Electronic Conference Proceedings*, pages 47–57. Linköping University Electronic Press, 2008.

[7] Modelica Association. Modelica: A unified object-oriented language for physical systems modeling, language specification version 3.3, 2012.

[8] OMG. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.

[9] Roland Samlaus, Claudio Hillmann, Birgit Demuth, and Martin Krebs. Towards a model driven modelica IDE. In *8th International Modelica Conference*, 2011.

[10] Mirko Seifert and Roland Samlaus. Static Source Code Analysis using OCL. In Jordi Cabot and Pieter Van Gorp, editors, *OCL'08*, 2008.

[11] M. Strobel, R. Vorpahl, C. Hillmann, X. Gu, A. Zuga, and U. Wihlfahrt. The OnWind modelica library for offshore wind turbines – implementation and first results. In *Proceedings of the Modelica Conference*, 2011.

[12] Claas Wilke. Java code generation for dresden ocl2 for eclipse. Großer beleg (minor thesis), Technische Universität Dresden, Dresden, Germany, February 2009.

Modeling System Requirements in Modelica: Definition and Comparison of Candidate Approaches

Andrea Tundis[1] Lena Rogovchenko-Buffoni[2] Peter Fritzson[2] Alfredo Garro[1]

[1]Department of Computer Engineering, Modeling, Electronics, and System Sciences (DIMES), University of Calabria, Italy, {garro,atundis}@dimes.unical.it

[2]Department of Computer and Information Science (IDA), Linköping University, Sweden, {peter.fritzson,olena.rogovchenko}@liu.se

Abstract

The modeling of system requirements deals with formally expressing constraints and requirements that have an impact on the behavior of the system to enable their verification through real or simulated experiments. The need for models representing system requirements as well as for methods and techniques centered on model-based approaches able to support the modeling, evaluation, and validation of requirements and constraints along with their traceability is today greater than ever. In this context, this paper proposes a *meta-model* for modeling the requirements of physical systems. Furthermore, different approaches for integrating the modeling of system requirements in the Modelica language and their verification during the simulation are proposed and, then, evaluated and compared through a case study.

Keywords Requirements, Properties, Modeling, Assertions, Modelica, Safety, Verification, Validation

1. Introduction

In the systems engineering context, although several research activities are focused on the system design phases, there is still a lack of practices and approaches that specifically deal with the analysis, modeling, and verification of requirements in an integrated framework. One of the main open issues concerns the support provided during the design for the verification and validation (V&V) of the system under consideration. Indeed, it is crucial not only to represent in detail both the structural and behavioral design of a system, but also to ensure the proper operation of the overall system and of each individual component to guarantee their functional correctness in compliance with the requirements. Moreover in several industrial domains such as nuclear plants, medical appliances, avionics, and automotive industry, some requirements such as safety requirements must be compliant to standard specifications (see IEC 61508) and norms to allow the commercial release of a system.

In order to add support for verification and validation during the design stage of the systems engineering process we formalize a set of concepts that will allow us to model *system requirements*, in [9] called properties. In the following we will use the term *requirement*, which is defined [9] as an expression that specifies a condition that must hold true at given times and places. As a rule, their identification and definition is neither a trivial nor a unique process, and can significantly depend on the reference domain and application context. Similarly, their formalization and modeling can vary with respect to the objectives to be reached.

In general, the first step of a systems engineering process is concerned with the analysis of informal *User Requirements* (*URs*). These are typically *problem-oriented* statements and they focus on the required capabilities. Thus, they need to be converted into *solution-oriented* statements. The *System Requirements* (*SRs*) are then derived by decomposing the *URs* into sets of basic required functionalities. *SRs* form the basis for the subsequent system functional analysis and design synthesis phases [8]. In particular, in the System Design phases, *SRs* are used to define both the structure and the behavior of the System under development. Specifically, in an equation-based context, the behavior of each system component, as well as the behavior of the entire system, is represented by a set of equations defined using component attributes (such as variables, parameters and constants).

Starting from the *SRs* and according to the defined System Design (*SD)*, additional *mechanisms* called *Requirement Assertions* can be defined in order to verify as well as to trace through the simulation the fulfillment of the *SRs*. Indeed a *requirement assertion* can be associated with a real system, subsystem, equipment or component, or with a model of the real system, subsystem or component and it defines what the system should guarantee, or the validity domain for the behavior of the system. In particular, in our context a *requirement* is represented by an *assertion* that is related to a specific *physical component* and which exploits the attributes of the component in order to verify and trace the fulfillment of some *SRs* related to a specific *component*. It worth to notice that while *user* and *system requirements* (both

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, 19 April, 2013, University of Nottingham, UK.

EOOLT 2013 website: http://www.eoolt.org/2013/

functional and non-functional) are used in the analysis and design phases for the development of the system under consideration; formalized requirements as *requirement assertions* are exploited during the verification phases for evaluating if the system requirements are satisfied by a specific system design model. Consequently, an appropriate approach to define formalized requirements along with the possibility to retrieve information about their status is crucial for the overall development process.

Few works are currently available addressing the modeling of requirements which was one of the goals addressed in ModelicaML [11] during the OPENPROD project [4]. Specifically, our proposal is strongly related to: (i) [9] in which the representation of the requirements is closely bound and restricted to the exploitation/implementation of a software library; (ii) [14, 15] where the communication processes and evaluation mechanisms among requirements, in order to enable the propagation of assessments among them, are not properly dealt. Furthermore, well-known simulation environments exploit assertions to verify system requirements; for example, MathWorks Matlab/Simulink provides assertions and bound checking blocks as configurable components. However, they are able to face only a limited set of specific aspects (e.g. zero/nonzero signal, threshold values). In this context, our aim is twofold: (i) to develop a comprehensive approach for the definition and modeling of requirements of a physical system in a clear and well-defined way, (ii) to define a mechanism to enable their traceability in order to support the verification process through simulation. To address these issues, a *meta-model* to represent system requirements along with some different solutions to model them are described in an equation-based context. On the basis of this meta-model, several *extensions* of the Modelica language [3] (an object-oriented modeling language to describe physical systems by differential, algebraic and discrete equations), to model requirements in a more flexible way, are introduced.

In Section 2 the proposed meta-model is described, in Section 3 both its use and its possible integration in the Modelica context are illustrated along with some notation extensions, in Section 4 a case study for the evaluation of the various approaches is presented and discussed whereas in Section 5 conclusions are drawn and future works outlined.

2. A Meta-model for representing System Requirements as RequirementAssertions

The concepts required for modeling system requirements are clearly identifiable and their representation can be generalized. For this purpose we define formal meta models [1].

Even though the notions of model and meta-model are crucial when we talk about representation and modeling, often these terms generate confusion, so it is necessary to clarify the difference between them and the context for the use of each of them.

Firstly we can define the concept of *subject* as the main thing that we want to think/reason about and on which we perform experiments. This usually belongs to the real world. To solve a problem we construct a simplified representation of the subject, called *model*, to which different experiments can be applied, in order to answer questions aimed at the subject. Since a model captures only a part of the complete subject, it is possible to define many models which represent the same subject but that are able to capture different characteristics, aspects, variables and parameters. In order to perform reasoning on a model it is necessary to know exactly which variables are available, furthermore, it is necessary to know the structure of the model. Such information can be expressed through *meta-data* by defining a higher abstraction level called *meta-model*. Hence, a meta-model is a model that defines the structure of valid models (see Figure 1).

In the following the definition and description of the proposed *meta-model* (see Figure 2) is provided. It is a combination of two main parts: the *Physical Meta-Model* (in the left-side) and the *Requirement Meta-Model* (on the right-side).

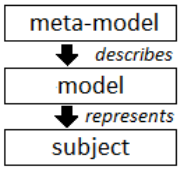


Figure 1. *Meta-model*, *model* and *subject* abstraction levels.

As previously stated, before defining *System Requirements*, it is necessary to build a representation of the physical model. Thus, the *meta-data* of the *Physical Meta-Model* are used to describe and to represent one among all the possible physical models of a specific actual system, whereas the *meta-data* of the *Requirement Meta-Model* are exploited to represent *System Requirements* in terms of *requirement assertions* by defining a possible *requirement-model* on a specific physical model representation.

Starting from the *Physical Meta-Model* side, the main concept is the *Attribute,* which represents a characteristic (i.e. temperature, pressure, level of liquid, age) of an entity (i.e. a system, a sub-system, a component); in the proposed meta-model, it is defined by (i) a *Name* (by which it is referred) (ii) a *Type* (type of value which is expected), (iii) a *Value* (a possible value among all the range of values related to a specific *Type*) and (iv) (optionally) a *Unit* of measure. Each *Attribute* is associated with one specific *Variability* which in turn can be (i) *Constant* which means that its *Value* never changes, (ii) *Variable* which means that its *Value* depends on other attributes, (iii) *Parameter* which means that its *Value* can be properly tuned. Moreover, each *Attribute* has to specify its access level called *Visibility* which, according to the meta-model, could be either *Private,* if accessible only internally to the component in which it has been defined, *Protected*, if accessible by the descendants, or *Public*, if accessible externally.

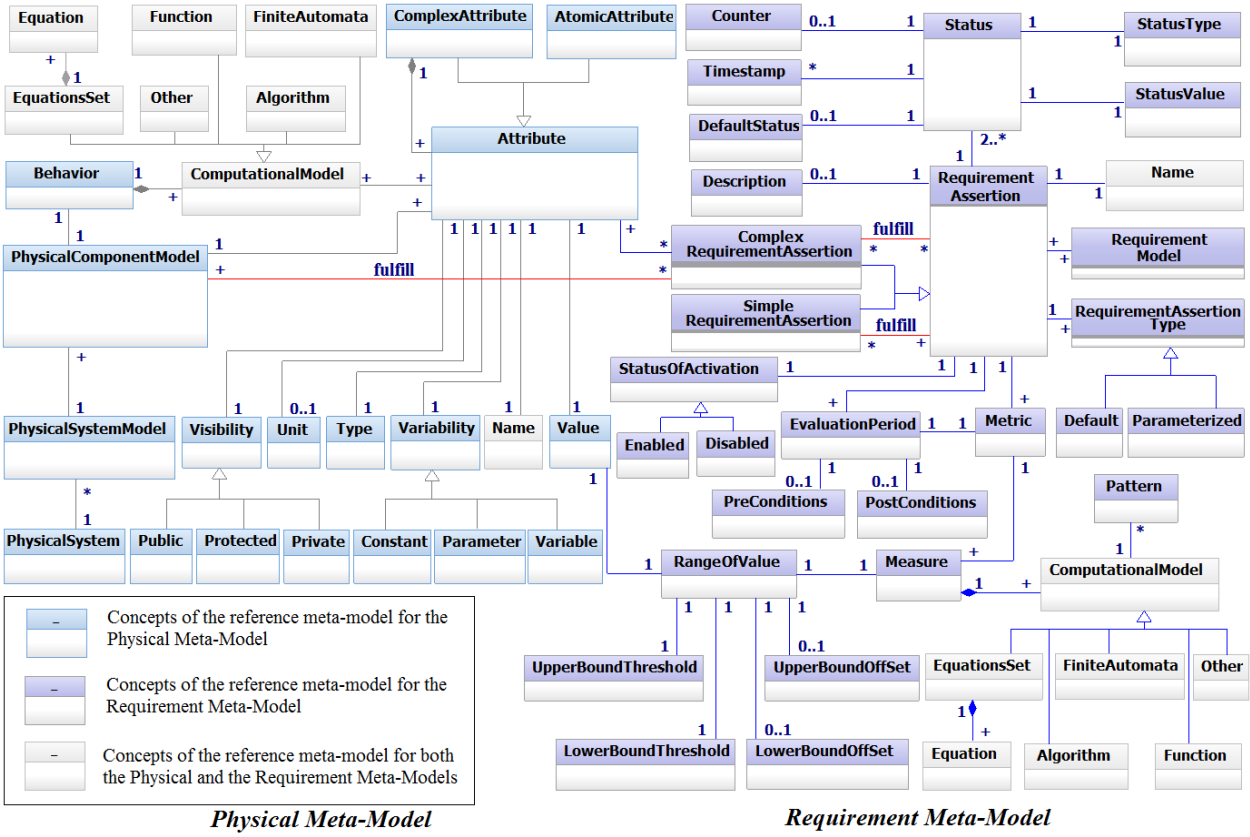


Figure 2. A meta-model for modeling *System Requirements*.

An *Attribute* can be (i) an *AtomicAttribute*, which means it cannot be further decomposed, (ii) a *ComplexAttribute,* that is, composed by other attributes.

A *ComputationalModel*, which could be represented through an a *Algorithm*, a *FiniteAutomata* (e.g. Timed Automata, Hybrid Automata, etc.), a *Function*, an *EquationsSet* (i.e. a set of *Equation* concepts), or by their combination as well as by *Other* kinds of computational models, defines the behavior of a *PhysicalComponentModel*. An *Attribute* has to belong at least to one *ComputationalModel* as well as a *ComputationalModel* has to use at least one *Attribute*. One or more *PhysicalComponentModels* compose a *PhysicalSystemModel,* which in turn is one of the many possible models to describe an actual system called *PhysicalSystem*.

While the *meta-data* on the left side of the figure is used for the description of the physical model, moving to the right side of the *meta-model*, we can see the concepts used for the modeling of *System Requirements*. Among these, the main concept is the *RequirementAssertion,* which is used to describe a *Requirement* of a system. A *RequirementAssertion* can be (i) a *SimpleRequirementAssertion*, that means it doesn't receive any input from any *PhysicalComponentModel*, (ii) a *ComplexRequirementAssertion,* which is connected directly to at least one *Attribute* and to one *PhysicalComponentModel*; this means that a *ComplexRequirementAssertion* is based on at least a *PhysicalComponentModel* and it is able to receive one or more input values coming from several attributes of the physical model; moreover, a *RequirementAssertion* (*SimpleRequirementAssertion* or *ComplexRequirementAssertion)* could be defined in terms of other *RequirementAssertions* whereas on a single *PhysicalComponentModel*, different *RequirementAssertions* can be defined.

According to the meta-model a *RequirementAssertion* belongs at least to one possible *RequirementModel* as well as a *RequirementModel* has to define at least one *RequirementAssertion*; each *RequirementAssertion* being characterized by:

- a *Name* and a possible *Description* in a text format by using the natural language;

a *RequirementAssertionType* which specifies the type of the role played by the *RequirementAssertion*; in particular a *RequirementAssertion* can have (i) a *Default* behavior type: it is allowed only to monitor a *PhysicalComponentModel* without influencing its evolution; (ii) a *Parameterized* behavior type: it is able to alter the value of a *PhysicalComponentModel* and influence its evolution (the *RquirementAssertion* has both *read* and *write* capabilities);

- at least two *Status* in order to represent the status of fulfillment of the requirement, which in turn is defined in terms of a *StatusType* and a *StatusValue.* The first concept defines the type of value that a state can take (i.e. a Boolean type, a real type, etc.) whereas the second one represents the value which is related to a specific *StatusType* (such as True/False

for a Boolean or NotEvaluated/Satisfied/NotSatisfied for a three valued logic, etc.). Each *Status* could be associated to both a *Counter* counting how many times the *RequirementAssertion* has gone in a specific state and a *Timestamp* in order to register each occurrence of the event. Moreover, a status can be defined as a *DefaultStatus* (useful, for example, in the initialization phase when none value is still provided to the *RequirementAssertion*). A *RequirementAssertion* has a *StatusOfActivation*, that means it can be *Enabled* and *Disabled* in order to decide if it takes/doesn't take part in a specific scenario or simulation run.

- at least one *EvaluationPeriod* to indicate when the *RequirementAssertion* has to be evaluated according to possible *PreConditions* and *PostConditions* that could be based on temporal values or on values coming from *Attributes*. Moreover for each *EvaluationPeriod* a *Metric* must be associated.
- at least a *Metric* to describe the objective to be verified for which the *RequirementAssertion* has been defined (e.g. Mean Time To Failure for the Reliability); each metric has to define a way which objectively allows its evaluation in terms of *Measure* (e.g. the number of failures in a period of time to measure the Mean Time To Failure). Specifically, a *Measure* can be expressed by adopting an appropriate *ComputationalModel*; moreover, one or more *Patterns* could be applied for representing such *ComputationalModels* when a sort of recurrent structure occurs (e.g. a threshold pattern, a derivative pattern, a delay pattern, etc.). Furthermore, each measure should define a *RangeOfValue*, within the *Value* of the *Attribute* which is related to, in which it is valid. Such *RangeOfValue* is specified by: (i) a *LowerBoundThreshold*: minimum value of validity in the range; (ii) *UpperBoundThreshold*: maximum value of validity in the range; moreover, further thresholds as *LowerBoundOffSet* and *UpperBoundOffSet* can be exploited when the *Value* of a *RequirementAssertion* is respectively below/above the *LowerBoundThreshold* and *UpperBoundThreshold* for a limited time.

RequirementAssertions can describe the state and the *intended behavior* [6, 7] of *PhysicalComponentModels,* i.e. the expected behavior for which components are designed. Both *Physical Meta-Model* and *Requirement Meta-Model* are jointly exploited to describe the overall model (hereafter called *Extended System Design – ESD*) of an actual system.

To further clarify the meta-model above described, a simple exemplification is provided below, where some of the above described concepts are exploited in order to define an *requirement model* upon a *physical model* in compliance with the proposed meta-model.

The *PhysicalSystem* under consideration is a Water System whose model, i.e. one among all possible *PhysicalSystemModels,* called WaterSystemModel is simply composed by a single *PhysicalComponentModel* of a Tank. The Tank is modeled through different *Attributes* such as the current level of liquid *levelInTank* as well as the height of the tank *tankHeight* (both as a Real *type* and *unit*="m"). Such attributes can be accessed externally (*Public Visibility*), whereas other *Attributes* can be used by the descendants of the Tank (*Protected Visibility*). All those *Attributes* (both with *Public* and *Protected Visibility*) are exploited into a *ComputationalModel* which is defined through different equations (*EquationsSet*) in order to model the *Behavior* of the *Tank.*

Let us assume to define a *RequirementModel* on this specific *PhysicalSystemModel* (the above described WaterSystemModel), in order to verify the following *RequirementAssertion* of a Tank (hereafter we refer to the model of the Tank), whose *Description* is: "The level of liquid in the tank shall never exceed 80% of the tank height" and its *Name* is "LevelOfLiquidInTank". According to the meta-model the status of activation (*StatusOfActivation*) of this *RequirementAssertion* is enabled (*Enabled*) for all the simulation time, and its evaluation period (*EvaluationPeriod*) has a duration equal to the duration of the simulation run without further specific *PreConditions* or *PostConditions*. The *Status* of the *RequirementAssertion* has a *StatusType* set to Boolean, consequently, the allowed status value (*StatusValue*) will range between true and false (or satisfied and notSatisfied).

The fulfillment of this *RequirementAssertion* is defined by a metric (*Metric*) based on the current level of fluid in the Tank, which is measured (*Measure*) as a percentage according to the maximum height of the tank. Consequently, the definition of the *RequirementAssertion* exploits the *levelInTank* and *tankHeight* that are both two *Public Attributes* of the Tank, moreover, an internal parameter, equal to 0.8, is used to express the percentage. Finally, this *Measure* is expressed by adopting as *ComputationalModel* a set of equations (*EquationsSet*). In particular, in this case by a single *Equation,* which is defined according to a threshold *Pattern* (e.g. levelInTank<0.8*tankHeight); a fragment of the possible Modelica (psedo) code is reported below.

```
requirement LevelOfLiquidInTank
  Real levelInTank(unit="m");
  Real tankHeight(unit="m");
  parameter Real limit (start=0.8);
equation
  levelInTank<limit*tankHeight;
end LevelOfLiquidInTank;
```

In the following section some approaches for modeling *System Requirements* through *RequirementAssertions*, based on the presented meta-model, are proposed.

3. Extending the Modelica language for Modeling System Requirements

In this Section different approaches for modeling *system requirements* and how they can be used to verify the intended behavior of the system and validate it through simulation are described. All the approaches are equation-

based and, in particular, based upon the Modelica language and ModelicaML (Modelica Modeling Language).

Modelica is a language for equation-based object-oriented mathematical modeling of physical systems (e.g., systems containing mechanical, electrical, electronic, hydraulic, thermal, control, electric power components) with *acausal* features, which allows defining models in a declarative manner [3].

ModelicaML is an UML profile, which is based on the SysML/UML profile and reuses its artifacts required for system specification. ModelicaML reuses several diagrams types from SysML without any extension, extends some of them, and also provides several new ones. ModelicaML diagrams are grouped into four categories: *Requirement, Structure, Behavior* and *Simulation* [13].

Although both Modelica and ModelicaML are expressly designed for modeling systems in a coherent framework based on an equation approach, they do not yet provide concepts to be used in order to represent and trace the occurrence of dysfunctional/abnormal behavior (such as faults and failures), that is to say, an observable deviation from the intended behavior at the system boundary [2, 6, 7].

In this perspective, the exploitation of the *meta-model* presented in the previous Section can be used to enrich both the Modelica language and ModelicaML to provide them with the capability of modeling system requirements and to enable model checking. In particular, different approaches are proposed and discussed in the following subsections based on the two main concepts of *requirement assertion* (see Section 2) and *fulfill* and some variants of them.

3.1 Approach A

In this approach the formal concepts of *requirement* and *fulfill* are defined as follows:

- *requirement:* which is represented by a *RequirementAssertion* able to validate the *behavior* of a specific *PhysicalComponentModel* which is related to, or to validate interactions among different *PhysicalComponentModels* (according to the *SRs* and the *SD*).
- *fulfill:* which expresses the entailment relationship between *PhysicalComponentModels* and a *requirement,* as well as among *requirements*. Moreover, it provides the propagation process of an assessment among *RequirementAssertions*.

An example model, which illustrates these concepts, is shown in Figure 3. In particular, after the declaration of the instances of both *PhysicalComponentModels* and *RequirementAssertions* their relationship is established according to the following *five connection-rules*:

1. the connections enabled through the *connect* construct among *PhysicalComponentModels* are defined to build the *SD* of the *PhysicalModel;*
2. the connections enabled through the *connect* construct among a *PhysicalComponentModel* and an *RequirementAssertion* are used to provide outputs coming from *PhysicalComponentModels* in input to *RequirementAssertions.*
3. the exploitation of the *fulfill* keyword is used to define which instance of an *RequirementAssertion* has to be satisfied/related from at least one specific instance of a *PhysicalComponentModel.*
4. the exploitation of the *fulfill* keyword is used among *RequirementAssertions* to enable the propagation mechanisms of assessment among them;
5. If *A1,..,An* are *RequirementAssertions* and *C1,..,Cm* are *PhysicalComponentModels*, then we can define *(A2,..,An,C1,..,Cm)fulfill(A1)*, where *A1* is satisfied if and only if *C1,..,Cm* satisfy *A1* as well as *A2,..,An* are all satisfied (*fulfill* follows the rule of the And logic).

As we can see in Figure 3, the *connect* construct, which is already available in the Modelica language, is used not only among *PhysicalComponentModels* but also between a *RequirementAssertion* and a *PhysicalComponentModel*. Even though the *connect* construct allows to define connections among attributes of two or more components in an acausal way [3], in this approach some restrictions are defined on it. As an example, the connection is only able to provide inputs from a physical component towards a *RequirementAssertion*. The reason for such a restriction is to prevent a *RequirementAssertion* from providing input to a *PhysicalComponentModel* and consequently affecting its behavior.

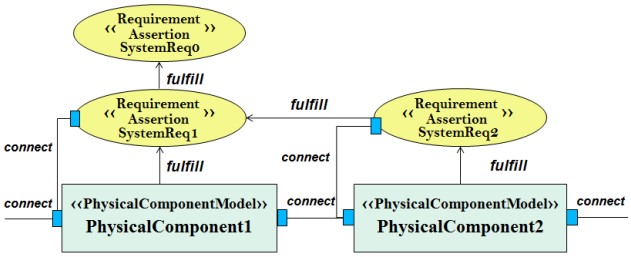


Figure 3. A verification model based on *requirement assertion* and *fulfill*.

3.2 Approach B

Whilst the above mentioned approach allows to model requirements in a simple and intuitive fashion, with the help of a minimal set of new concepts (i.e. *requirement assertion* and *fulfill*), the addition of extra connections between *requirement assertions* and components through *connect*, could make the *ESD* overly verbose and difficult to read from a visual representation point of view, thus complicating the maintainability of the source code.

Therefore, an alternative approach is a variant of the previous one in which along with the keyword *requirement*, another concept (and another keyword) called *On,* which is only visible in the source code of a *RequirementAssertion,* is introduced. Similar to the *extends* construct, but with some restrictions on the

inherited elements, the *On* keyword enables a *requirement* to be defined on specific *PhysicalComponentModels*. Such a *requirement* will inherit the *attributes* on which it will carry out the processing.

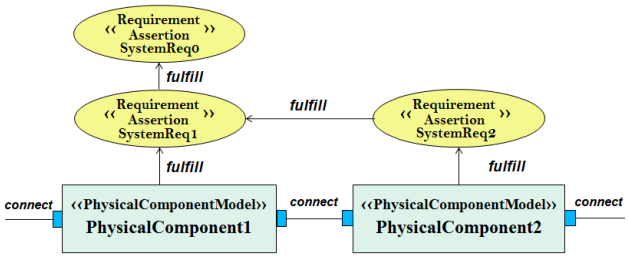


Figure 4. Modeling Requirements using the On construct.

The process to build the *ESD* follow the *five-connection-rules*, which have been described in Section 3.1 except for the rule number 2; in this way:

- it allows to avoid the exploitation of extra *connect* (between *PhysicalComponentModels* and *RequirementAssertions*) in order to provide input values coming from *constants*, *parameters* or *variables* of physical components towards an *requirement*. Indeed, such relationships are established during the definition of the *RequirementModel* through the exploitation of the *On* keyword;
- it allows to avoid of having too many connections into a graphical representation, as it is in Figure 4, by also reducing the lines of the source code of the *Extended System Design*.

The concept of *fulfill* is that explained in the previous section.

3.3 Approach C

Often, it is necessary to have additional mechanisms for generating dysfunctional/abnormal behavior in a physical component, so as to assess the consequences on the whole system.

To this end, approach C proposes the possibility of altering the values of the components starting from the B approach and adding the new notions of *tester entity/component entity* and the *supersede* keyword. The *tester entity* can be seen as a specific component that is defined on a *PhysicalComponentModel* and which is able to generate outputs (e.g. signals, events or values) according to specific functions and inject them into the *PhysicalComponentModel* in order to alter its intended/nominal behavior (expected values). The *supersede* keyword enables the mechanism to create a reference between an instance of a *tester entity* and an instance of a *PhysicalComponentModel*. In particular, the following rules define the semantics of the *supersede* keyword and how to use it:

1. the exploitation of the *supersede* keyword is used to define which specific instance of a *PhysicalComponentModel* could be compromised by which specific instance of a *Tester component*;
2. If *T1,..,Tn* are *Tester components* and *C* is a *PhysicalComponentModel*, then we can define *(T1,..,Tn)supersede(C)*, where the operation work of *C* could be influenced only by one among the *T1,..,Tn Tester components* (*supersede* follows the rule of the XOr logic).

RequirementAssertions can monitor the occurrence of abnormal/dysfunctional behavior in physical components; the *fulfill* relationship is exploited by the *RequirementAssertion* to check the impact and the consequently propagation of possible unexpected values in a component on other components (see Figure 5). The *On* keyword enables both *RequirementAssertions* and *Tester components* to have access directly to the attributes of the physical component models on which they are defined.

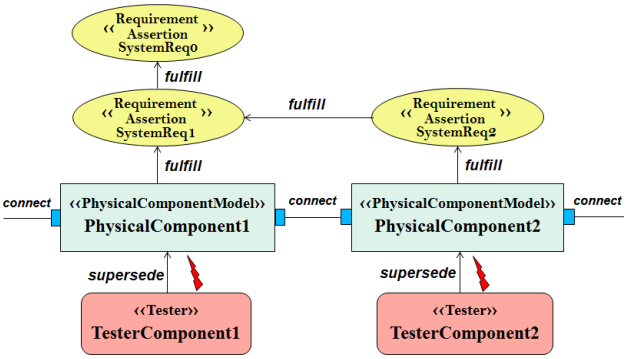


Figure 5. Requirements and Tester component for the dysfunctional behavior analysis.

4. A case study

In this Section, a case-study is first described and then used to evaluate some of the solutions which have been proposed in the previous Section; for this purpose, both the ModelicaML diagrams and the Modelica code are presented; finally, the pros and cons of each solution are discussed.

4.1 System Description

The possible implementation of the previously presented approaches along with the significant reduction of programming and implementation efforts to model *system requirements* as well as the increased readability, are demonstrated through a typical case study of a Tank System.

The Tank System is composed by four main physical components: a *Source* component, a *Tank* component, a *LevelController* component and a *Sink* component. The *Source* component produces a flow of liquid, which is taken in input by the *Tank* component. The *Tank*, which is managed through the *LevelController* component, provides in output a liquid flow according to the control law defined by the *LevelController*. The *Sink* is the component where a part of liquid is sent.

After an analysis of the *URs*, the following main *SRs* (and many others) have been identified:

- *System_Requirement_1*: the system has to be composed by one Source Component, one Sink Component, at least one Tank Component and at least one LevelCotroller Component;
- *System_Requirement_2:* each tank has to provide one port called *qIn* in order to receive flow from another possible Tank Component (or from the Source component if it is the first Tank component in the chain);
- *System_Requirement_3:* each tank has to be connected to its own LevelController component;
- *System_Requirement_3_1*: each Tank component has to provide a port called *tSensor* in order to provide signal to the LevelController component;
- System_Requirement_*4*: the Source component has to provide a flow port called *qOut*;
- *System_Requirement_4_1*: the liquid flow produced by the Source component has to be equal three times the initial flow after 150 seconds;
- *System_Requirement_5_1*: the liquid flow produced by the Source component should be less than 10 m^3/s.
- *System_Requirement_5_2*: the role of the LevelController should be verified by exploiting both the *h* level from the Tank component and the *qOut* flow.
- *System_Requirement*_5_3: the validity of both the *tActuator* (Out-flow) and the *outFlowArea* values should be checked according to a specified function;
- *System_Requirement_5_4*: both the *h* level and the *tSensor* should provide the same values;
- *System_Requirement_5_5*: the *h* level coming from the Thank should be checked according to a specified function.

Starting from the SRs above described, the SD of the Tank System has been defined as shown in Figure 6, whereas in the following, a fragment of the Modelica code used to implement the Tank System is reported.

```
package PhysicalComponentModel
model Source;
  LiquidFlow qOut;
  parameter Real flowLevel=0.02;
equation
  qOut.lflow = if time>150 then
  3*flowLevel else flowLevel;
end Source;
model Tank
  ReadSignal tSensor;
  ActSignal tActuator;
  LiquidFlow qIn;
  LiquidFlow qOut "Connector, flow (m3/s)
  through output valve";
  parameter Real area(unit="m2")=0.5;
  parameter Real flowGain(unit="m2/s")=
  0.05;
  parameter Real minV = 0,maxV = 10;
  Real actuatorControllerV;
  Real outFlowArea(unit="m");
  Real h(start=0.0, unit="m");
equation
  der(h)=(qIn.lflow-qOut.lflow)/area;
  actuatorControllerV=flowGain*
  tActuator.act;
  qOut.lflow = LimitValue(minV, maxV,
  actuatorControllerV);
  tSensor.val=h;
  outFlowArea=-qOut.lflow/flowGain;
end Tank;
…
end PhysicalComponentModel;
```

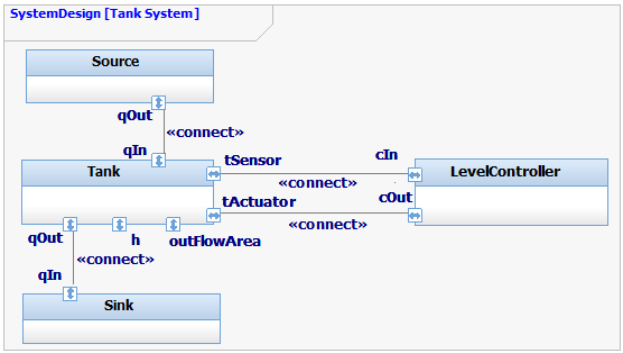


Figure 6. The System Design (*SD*) of the Tank System.

Moreover, starting from the *SD* of the Tank System, the following *Requirement Assertions* have been defined; they should be represented and verified during simulation in order to ensure the proper operation of the system. In the next subsections some of the proposed approaches are applied for the modeling of requirements.

4.2 Exploiting the A Approach

In this example, starting from the *System Requirements* specified in the previous subsection, a set of *RequirementAssertions* can be defined on the *SD* of the Tank System by exploiting the Approach A; in particular:

- *RequirementAssertion_1*: *LimitInFlow*, which takes in input the value of the *qOut* port of the Source component. It is satisfied if the liquid flow produced by the Source component is less than a specific "maxLevel" (i.e. liquidFlow<=maxLevel, in our case maxLevel =10).
- *RequirementAssertion_2*: *ControlOutFlow*, which takes in input the *h* level from the Tank component and the *qOut* flow to validate the role of the *LevelController*; moreover, to be valid it must be fulfilled by both the *RequirementAssertion_3* and the *RequirementAssertion_4*.
- *RequirementAssertion_3*: *ActuateOutFlow*, which takes in input both the *tActuator* (Out-flow) and the *outFlowArea*, checks if the *outFlowArea* value is proportional at the *tActuator* signal.
- *RequirementAssertion_4*: *SenseLevel*, which takes in input both the *h* level and the *tSensor*, checks if the sensor output is equals to the *h* level (i.e. lLevel=sensorOuput).
- *RequirementAssertion_5*: *ControlLevel*, which takes in input the *h* level coming from the Tank component,

checks if hLevel<9 and hLevel>5; moreover, to fulfill the *RequirementAssertion_5,* both the state of *RequirementAssertion_1* and of *RequirementAssertion_2* have to be satisfied.

Figure 7 shows an example of ModelicaML-based notation for the different concepts. In the following, some code fragments of the *RequirementModel* and, in particular, the implementation of *RequirementAssertion_1* and of *RequirementAssertion_5,* introducing the new keyword *requirement*, are reported.

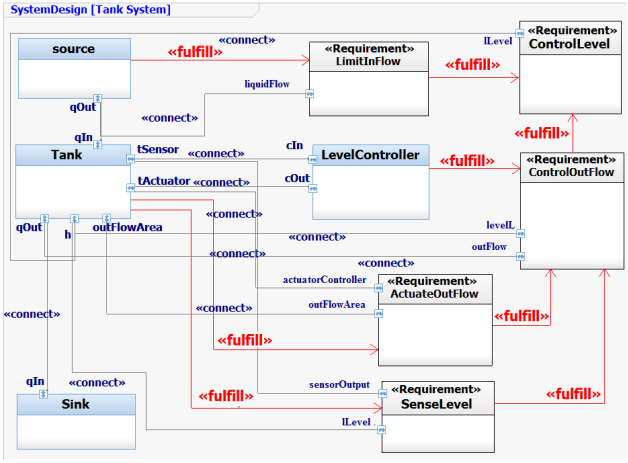


Figure 7. Approach A for modeling requirements of the Tank System.

```
package RequirementModel
requirement Requirement1
  Real liquidFlow; "qOut of Source"
  parameter Real maxLevel=10;
equation
  if liquidFlow<=maxLevel then
    Status.satisfied;
    …
end Requirement1;
…
requirement Requirement5
  Real lLevel;
  parameter Real Lmin=5, Real Lmax=9;
equation
  if lLevel<Lmax and lLevel>Lmin then
    …
end Requirement5;
end RequirementModel;
```

In the snippet of code shown subsequently, both the *PhysicalSystemModel (or SD)* and the *RequirementModel* are composed.

```
model ExtendedSystemDesign
  //PhysicalComponentModels
  Source source;
  Tank tank1(area=1);
  …
  //RequirementComponents
  Requirement1 limitInFlow;
  …
  Requirement5 controlLevel;
equation
  //Connection among PhysicalComponents
  connect(source.qOut,tank1.qIn);
  …
  //fulfill connections
  (source)fulfill(limitInFlow);
  (tank1)fulfill(actuateOutFlow);
  (tank1)fulfill(senseLevel);
  (limitInFlow,controlOutFlow)
  fulfill(controlLevel);
  (levelController,actuateOutFlow,
  senseLevel)fulfill(controlOutFlow);
  //connection between physical
  //components and requirements
  connect(tank1.h,controlLevel.L);
  connect(tank1.h,senseLevel.lLevel);
  connect(source.qOut,limitInFlow.
  liquidFlow);
  …
end ExtendedSystemDesign;
```

By adopting this approach, the *RequirementModel* is completely decoupled from the *PhysicalSystemModel* of the system under consideration. Indeed, a *requirement model* only requires input values of specific types, regardless of the type and the number of components that the values come from. This means that a *requirement model* could be re-used to validate physical components belonging to different *SD*, although the semantics of such physical components could be completely different. The link between the *RequirementModel* and *PhysicalSystemModel*, occurs only in the *ESD*, through the *fulfill* relationships which govern the assignment of a component to a requirement, while the inputs to be sent to the requirement are provided by the *connect* construct.

4.3 Exploiting the B Approach

In this example the Approach B is exploited to represent the same Tank System including the *RequirementAssertions* described in the previous subsection. Figure 8 shows the related ModelicaML-based notation of such a modeling approach. As we can see, the diagram is less crowded with connections and consequently easier to read.

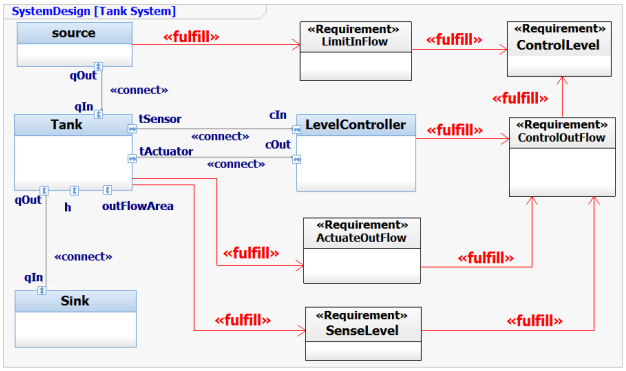


Figure 8. Approach B for modeling requirements of the Tank System

As it is shown in the next code fragments illustrating the source code of *Requirement_Assertion_1* and of *Requirement_Assertion_5*, both the keyword *requirement* along with the *On* keyword are combined for the definition of each *requirement*. Specifically, starting from the *Source* model, *Requirement1* is defined on it; this means that *Requirement1* is able to use (read-only) all the

Public attributes, which have been defined by the *Source* model. In particular, the *qOut* attribute of the *Source* model can be used by *Requirement1* without further referencing or connections with the *Source* model.

```
package RequirementModel
requirement Requirement1 On Source
  parameter Real maxLevel=10;
equation
  if Source.qOut<=maxLevel then
    Status.satisfied;
  else
    Status.notSatisfied;
  end if;
…
end Requirement1;
…
requirement Requirement5 On Tank
  parameter Real Lmin=5, Lmax=9;
equation
  if Tank.h<Lmax and Tank.h>Lmin then
  …
end Requirement5;
end RequirementModel;
```

As for the previous example a fragment of source code combining both the *PhysicalSystemModel* and the *RequirementModel* is presented. As we can see, no connections which use the *connect* construct between a *PhysicalComponentModel* and a *requirement component*, are present in the source code of the *ESD* model.

```
model ExtendedSystemDesign
  //PhysicalComponentModels
  Tank tank1(area=1);
  Sink sink1;
  …
  //RequirementComponents
  Requirement1 limitInFlow;
  …
  Requirement5 controlLevel;
equation
  //Connections among PhysicalComponents
  connect(source.qOut,tank1.qIn);
  …
  //fulfill relationships
  (source)fulfill(limitInFlow);
  (tank1)fulfill(actuateOutFlow);
  (tank1)fulfill(senseLevel);
  (levelController,actuateOutFlow,
  senseLevel)fulfill(controlOutFlow);
  (limitInFlow,controlOutFlow)
  fulfill(controlLevel);
end ExtendedSystemDesign;
```

By adopting this approach, the *RequirementModel* is not completely decoupled from the *PhysicalSystemModel* (this make *requirement assertions* less flexible and less reusable) as it knows *Public Attributes* that are defined in the *PhysicalSystemModel*. On the other hand, it allows for a more immediate exploitation making the *ESD* model easier to read by hiding the details of the matching between the *PhysicalSystemModel* and the *RequirementModel*. Indeed, as it is shown both in Figure 9 and through the code of the *ESD*, only the *fulfill* relationships are visible, while the *connection* (through the *connect* construct) among *PhysicalComponentModels* and *RequirementAssertions* are not part of the *ESD*.

4.4 Exploiting the C Approach

The Approach C is adopted to model the previously described *requirement assertions* on the Tank System. Additionally, the possibility of modeling entities that alter the intended behavior of components, and consequently of the system, is taken into account by exploiting *tester entities/components.*

In this section, three *tester components* have been defined in order to illustrate their use:

- *AlterSourceFlow* and *AlterSourceFlow2* on the *Source* component, respectively producing the double of the liquid in the first case and producing a negative value of liquid in the second case.
- *AlterOut* on the *Tank* component, where the LimitValue function has been removed from the behavior of the tank.

In the following, some code fragments describing the *TesterModel* and, specifically, the source code of the *AlterSourceFlow* and of the *AlterOut* are reported.

```
package TesterModel
tester AlterSourceFlow On Source
  parameter Real flowLevel=0.04;
  …
equation
  qOut.lflow=flowLevel;
end AlterSourceFlow;
tester AlterOut On Tank
  …
equation
  actuatorControllerV=-
  flowGain*tActuator.act;
  qOut.lflow = actuatorControllerV;
  tSensor.val = h;
  outFlowArea=-qOut.lflow/flowGain;
end AlterOut;
end TesterModel;
```

As we can see in the source code below, the link between *PhysicalSystemModel* and *TesterModel* is defined in the *ESD* through the keyword *supersede.* In Figure 9 a ModelicaML-based notation for such a modeling approach, introducing both *Requirement* and *Tester components* as well as physical components is depicted.

```
model ExtendedSystemDesign
  //PhysicalComponentModels
  Tank tank1(area=1);Source source;
  …
  //RequirementComponents
  …
  //TesterComponents
  AlterSourceFlow alterSourceFlow;
  AlterSourceFlow2 alterSourceFlow2;
  AlterOut alterOut;
equation
  //supersede relationships
  (alterSourceFlow,
  alterSourceFlow2)supersede(source);
  (alterOut)supersede(tank1);
  //fulfill relationships
  …
end ExtendedSystemDesign;
```

It is worth noting that one possible variant of the Approach C consists in defining the relationships between a *PhysicalComponentModel* and a *Tester component* in the *ESD* by using the construct *connect*, in order to avoid the exploitation of the *On* keyword during the definition of the tester components in the *TesterModel.* By adopting this version (similar to the A Approach), the *PhysicalSystemModel* will be completely decoupled from both the *RequirementModel* and the *TesterModel*.

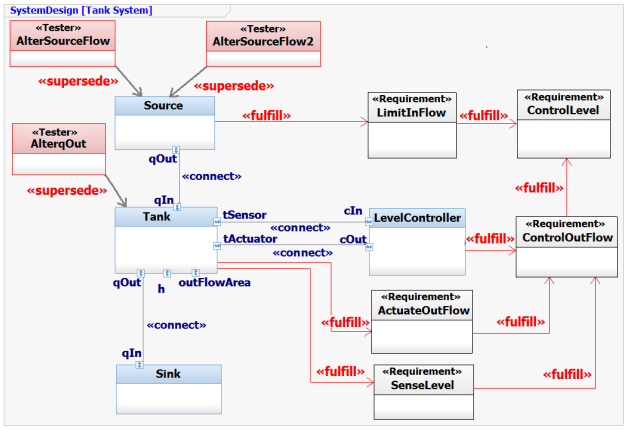


Figure 9. Approach C for modeling requirements of the Tank System.

5. Conclusions and future works

The paper focused on the modeling of requirements in an equation-based context. In particular, a reference *meta-model* for representing *System Requirements* in terms of *RequirementAssertions* has been defined. Then, three different approaches for the modeling of *System Requirements* that adhere to the proposed *meta-model*, have been outlined. All of them aim to provide support for model verification by defining extensions of the Modelica language, and, one of them also aim to extend such model verification by supporting the modeling of system failures and thus allowing to analyze the behavior of the system in presence of faults.

Finally, the exploitation of the proposed approaches in a case study concerning a Tank System has allowed to compare their advantages and disadvantages as well as to appreciate their effectiveness and usability in the system modeling phases.

This work is part of an ongoing research project (MODRIO project – ITEA 2) [10] aiming at developing a model-based approach for system requirements verification and fault tree analysis through Modelica extensions for Requirements modeling and Safety analysis.

Ongoing research efforts are devoted to improving the proposed approaches through both their implementation in *OpenModelica* [12] and their integration in a full-fledged Systems Engineering development process [5] along with an extensive experimentation in the analysis of systems in different application domains such as automotive, railway, avionics and energy.

Acknowledgements

This work has been supported by ITEA 2 MODRIO project. Andrea Tundis has been supported by a grant funded in the framework of the "POR Calabria FSE 2007/2013".

References

[1] T. Clark, P. Sammut, and J. Willans. Applied metamodelling: a foundation for language driven development (Second Edition), 2008.

[2] R. Cressent, V. Idasiak, F. Kratz, and P. David. Mastering safety and reliability in a model based process. *Proc. of the Reliability and Maintainability Symposium (RAMS)*, Lake Buena Vista (FL, USA), January 2011.

[3] P. Fritzson, Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, Wiley IEEE Press, 944 pages, February 2004.

[4] P. Fritzson. Integrated UML-Modelica Model-Based Product Development for Embedded Systems in OPENPROD. *Proc. of the 1st Workshop on Hands-on Platforms and tools for model-based engineering of Embedded Systems (Hopes'2010),* Paris, June 15, 2010.

[5] A. Garro and A. Tundis. Enhancing the RAMSAS method for Systems Reliability Analysis through Modelica. *Proc. of the 7th Workshop on Model-Based Product Development (MODPROD)*, Linköping (Sweden), 5-6 February, 2013.

[6] A. Garro and A. Tundis. Modeling and Simulation for System Reliability Analysis: The RAMSAS Method. *Proc. of the 7th IEEE International Conference on System of Systems Engineering (IEEE SoSE)*, Genova (Italy), July 16-19 2012.

[7] L. Grunske and B. Kaiser. Automatic Generation of Analyzable Failure Propagation Models from Component-Level Failure Annotations. *Proc. of the 5th Int. Conf. on Quality Software (QSIC)*, Melbourne (Australia), September 2005.

[8] H. P. Hoffmann. System Engineering Best Practices with Rational Solution for Systems and Software Engineering. February 2011. http://www.ibm.com/.

[9] A. Jardin, D. Bouskela, T. Nguyen, N. Ruel, E. Thomas, R. Schoenig, S. Loembé and L. Chastanet. Modelling of System Properties in a Modelica Framework. *Proc. of the 8th International Modelica Conference*, TU Dresden, March 20-22, 2011.

[10] ITEA 2 Projects: MODRIO - http://www.itea2.org/.

[11] F. Liang, W. Schamai, O. Rogovchenko, S. Sadeghi, M. Nyberg and P. Fritzson. Model-based Requirement Verification: A Case Study. *Proc. of the 9th International Modelica Conference (Modelica'2012*), Munich (Germany), September 3-5, 2012.

[12] OpenModelica - Open Source Modelica Consortium (OSMC) - https://www.openmodelica.org/.

[13] OpenModelica Project: ModelicaML - A UML Profile for Modelica. www.openmodelica.org/modelicaml.

[14] W. Schamai, P. Fritzson, C.J.J. Paredis, P. Helle. ModelicaML Value Bindings for Automated Model Composition. *Proc. of the Symposium on Theory of Modeling and Simulation (DEV'12),* Orlando, FL (USA) March 26-29, 2012.

[15] W. Schamai, P. Helle, P. Fritzson, and C. Paredis. Virtual Verification of System Designs against System Requirements. *Proc. of 3rd International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES'2010)*, Oslo (Norway), October 4, 2010.

Session II: Parallel Simulation

Parallelization Approaches for the Time-efficient Simulation of Hybrid Dynamical Systems: Application to Combustion Modeling

Abir Ben Khaled[1] *Mongi Ben Gaid*[1] *Daniel Simon*[2]

[1]IFP Energies nouvelles, 1-4 avenue de Bois-Préau, 92852 Rueil-Malmaison, France,
{abir.ben-khaled, mongi.ben-gaid}@ifpen.fr
[2]INRIA and LIRMM-CNRS-Université Montpellier Sud de France, DEMAR team, 95 rue de la Galéra,
34090 Montpellier, France, daniel.simon@inria.fr

Abstract

The need for time-efficient simulation is increasing in all engineering fields. Potential improvements in computing speeds are provided by multi-core chips and parallelism. However, the efficient numerical integration of systems described by equation oriented languages requires the ability to exploit parallelism. This paper investigates the problem of the efficient parallelization of hybrid dynamical systems both through the model and through the solver. It is first argued that the parallelism is limited by dependency constraints between sub-systems, and that slackened synchronization between parallel blocks may provide speed-ups at the cost of induced numerical errors, which are theoretically examined. Then two methods for automatic block diagonalization are presented, using bipartite graphs and hypergraphs. The application of the latter method to hybrid dynamical systems, both from the continuous state variables and discontinuities point or view, is investigated. Finally, the model of a mono-cylinder engine is analyzed from equations point of view and a possible split using the hypergraph method is presented and discussed.

Keywords Parallel computing, model decomposition, delay error, dependencies constraints, multicore simulation

1. Introduction

The design and validation of complex systems, like cyber-physical systems which include both physical and computational devices, is costly, time consuming and needs knowledge and cooperation of different disciplines [15]. For example, vehicle power-trains belong to such category and require the coordinated design of both mechanical, electrical, thermodynamic and chemical models (from a physical point of view) and many-sided controllers (from a computational point of view).

A global simulation is needed at an early stage to speed-up the design, development and validation phases. The main purpose of the numerical simulation is, when an analytic solution cannot be derived, to approximate as faithfully as possible the behavior of the complex dynamic system. In other words, bounding and minimizing the simulation errors is an important goal of numerical simulations so that the designers can be confident with the prediction.

Simulating complex systems is time consuming in term of calculations, and reaching real-time is often out of the capabilities of single processors. Parallel computing can be performed by splitting the models into several sub-models that are concurrently simulated on several processors to ensure the compliance with the real-time constraints.

The data dependencies due to the coupled variables between sub-models lead for waiting periods and idle processor time, decreasing the efficiency of threaded parallelism on the multicore platform. Therefore these dependency constraints should be relaxed as far as possible. However, to avoid too large numerical errors in the simulation results, a minimal synchronization between sub-models must be achieved, and the parallelism between computations must be carefully restricted.

The relaxation of the synchronization constraints, while guaranteeing correct simulation results, needs to split the model properly before distribution over several CPUs. An efficient decomposition relies on knowing how and where to cut in order to decouple subsystems as far as possible. Relaxed data dependencies may lead to slack synchronization between sub-models, until reaching an acceptable trade-off between the computation costs and the simulation precision.

The aim of this paper is to propose approaches for the time-efficient and real-time simulation of hybrid dynamical systems, showing two techniques of parallelization which can be combined to reach complementary objectives:

- Parallelization across the model where the delay error due to the model decomposition is evaluated to determine the involvement and the weighting of each of its elements, and consequently to know how to act for the error reduction. The parallelization approach is based on a system

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. 19 April, 2013, University of Nottingham, UK.
 Proceedings available at:
http://www.ep.liu.se/ecp_home/index.en.aspx?issue=084
EOOLT 2013 website:
http://www.eoolt.org/2013/

splitting using the block-diagonal forms of state and event incidence matrices.

- Parallelization through the solver where a new method of parallelization at the event detection and location level is presented. The parallelization approach is situated on the solver level using the block-diagonal form of the event incidence matrices.

This paper is organized as follows. First, a formal model of a hybrid dynamic system is established. After that, delay errors due to parallelization across the model are evaluated in the context of real-time simulation. Then a study of system splitting using the block-diagonal form of incidence matrices, related to states and events, is performed. Finally, a case study concerning a mono-cylinder engine model is presented and test results are performed by analyzing the relationships between the states, the events and the states/events together.

2. Related Work

In the context of the co-simulation or the parallelization across the model, where the system is split into several sub-systems and simulated in parallel on different processors, several works already targeted the real-time distributed simulation of complex physical models. In [9], the study focused on the case of fixed-step solvers. Then, in [2], the study was extended to examine the case of variable-step solvers. Besides, in [16] distributed simulation was performed in Modelica [11], using a technology based on bilateral delay lines called transmission line modeling (TLM) combined with solver in-lining. The TLM technique consider the decoupling point when the variables change slowly and the time-step of the solver is relatively small, so that the decoupled subsystems can be considered as connected by constant variables.

Another kind of parallelization is called parallelization through the solver. It concerns the execution of the model on a single thread while its numerical resolution is parallelized. It relies on using well know parallel approaches for solving the required steps of the used solver. For example, parallelizing matrix inversions, which are needed when using an implicit method (see [17] and [12]) and parallelizing operations on vectors for ODEs resolution by separating them into modules (see PVODE in [7] implemented using MPI (Message-Passing Interface) technology).

3. Problem Formalization

In this kind of cyber-physical systems the physical part is modeled in the continuous-time domain using hybrid ODEs. It belongs to the hybrid systems category because of some discontinuous behaviors, that correspond to events triggered off when a given threshold is crossed. Controllers, which interact with physical parts, represent computational models. They are modeled on the discrete-time domain and sampling is a mixture of time-driven and events-driven features.

Let us provide a formal model, considering a hybrid dynamic system Σ whose continuous state evolution is governed by

$$\dot{X} = f(t, X, D, U) \quad \text{for } t_n \leq t < t_{t+1}, \tag{1a}$$

$$Y = g(t, X, D, U). \tag{1b}$$

where $X \in \mathbb{R}^{n_X}$ is the continuous state vector, $D \in \mathbb{R}^{n_D}$ is the discrete state vector, $U \in \mathbb{R}^{n_U}$ the input vector, $Y \in \mathbb{R}^{n_Y}$ is the output vector and $t \in \mathbb{R}^+$ is the time.

$(t_n)_{n\geq 0}$ is a sequence of strictly increasing time instants representing discontinuity points called "state events", which are the roots of the equation

$$h(t, X, D, U) = 0. \tag{2}$$

h is usually called zero-crossing function or event indicator, used for *event detection* and *location* [18].

At each time instant t_n, a new continuous state vector may be computed as a result of the *event handling*

$$X(t_n) = I(t_n, X, D, U), \tag{3}$$

and a new discrete state vector may be computed as a result of discrete state update

$$D(t_n) = J(t_{n-1}, X, D, U), \tag{4}$$

If no discontinuity affects a component of $X(t_n)$, the right limit of this component will be equal to its value at t_n.

This hybrid system model is adopted by several modeling and simulation environments and is underlying the functional mock-up interface (FMI) specification [6].

We assume that Σ is well posed in the sense that a unique solution exists for each admissible initial conditions $X(t_0)$ and $D(t_0)$ and that consequently X, D, U, and Y are piecewise continuous functions in $[t_n, t_{n+1})$.

The system must be split for numerically integration on a set of $\mathcal{P}$ cores. The objective is to speedup the simulation through the exploitation of parallelism, while keeping the following objectives in mind [2]:

- minimization of the delay errors, which are related both to the dependency constraints and to the distribution of the state variables;
- optimization of the computational resource usage through load balancing and idle time avoidance: high CPU utilization is influenced by the size and independence of the computational tasks;
- minimization of the number of solvers interrupts due to discontinuities unrelated with the local sub-system: discontinuities need that the corresponding numerical solver must be stopped and restarted, therefore introducing computation overheads;
- keeping an acceptable level of numerical stability and accuracy.

4. Evaluation of Delay Errors due to Decoupling

4.1 Context and Motivation

To begin with something simple, assume now that the system can be split into two subsystems as in Figure 1.

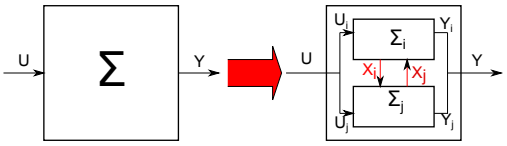


Figure 1. System splitting for parallelization

For simplicity, the analysis is focused only on continuous time-invariant models. In the following, we will denote by $x(n) = x(t_n)$. Therefore, the system can be written as:

$$\begin{cases} \dot{x}_i &= f_i(x_i, x_j, U_i) \\ Y_i &= g_i(x_i, x_j, U_i) \end{cases} \quad and \quad \begin{cases} \dot{x}_j &= f_j(x_j, x_i, U_j) \\ Y_j &= g_i(x_j, x_i, U_j) \end{cases} \tag{5}$$

with $X = [x_i \;\; x_j]^T$

Here U_i are the inputs needed for $\sum_i$ and U_j are the inputs needed for $\sum_j$. In other words, $U_i \cup U_j = U$ and $U_i \cap U_j$ can be an empty set or not according to the achieved decoupling.

In the same way, Y_i are the outputs produced by $\sum_i$ and Y_j are the outputs produced by $\sum_j$. In other words, $Y_i \cup Y_j = Y$ and $Y_i \cap Y_j = \oslash$.

After parallelization, as shown in Figure 1, each subsystem Σ_i have a private subset U_i as input vector and a private subset Y_i as output vector, and the direct interaction between subsystems are only due to a subset made of variables of the state vector. Hence the focus is now on the internal data flow, i.e. the shared state variables to be identified, and whose number should be minimized to enhance the independence of parallel branches.

In general, in order to compute the next state $x_i(n+1)$, numerical integrators use [1, 8]:

- Always, the current value of the state $x_i(n)$ and the derivative $f_i(X(n))$.
- For multi-step methods, the m previous values of the state $x_i(n-\alpha)$ and/or the derivatives $f_i(X(n-\alpha))$ (with $\alpha = 1, .., m$).
- For methods with order higher than one, higher derivative such as $f_i(X(n) + \beta.f_i(X(n)))$ for a 2^{nd} order.
- For implicit methods, the future value of the derivative $f_i(X(n+1))$ computed through iterations.

Consequently, in order to compute the next state value $x_i(n+1)$, the numerical solver needs at least values for $x_i(n)$ and $\dot{x}_i(n) = f_i(X(n)$ (see Figure 2).

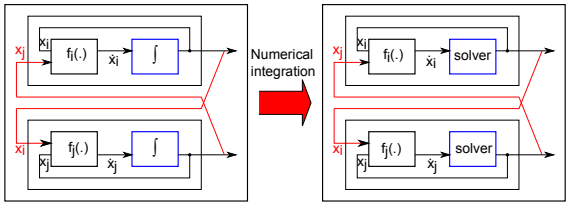


Figure 2. System's internal composition

However, when the computation of $\dot{x}_i(n) = f_i(X(n))$ is required, the value of $x_i(n)$ is already available, but this assertion is invalid for $x_j(n)$ with $j \neq i$. In fact, x_j is only updated every synchronization interval P that is larger than the integration step. In other words, $x_j(n)$ can be available only if the time t_n corresponds to the synchronization time, otherwise its value will be the one of last updated. Thereby, the focus will be then in $\dot{x}_i(n) = f_i(x_j(n))$ for $j \neq i$.

4.2 Evaluation of Delay Errors in a Basic Problem

In order to evaluate the influence of using a delayed value of x_j in $f_i(.)$ due to the lack of communication at processing time, let us start by a simple case with a single state variable x. The analysis and results carried out in this case will be further applied to the large problem with N state variables $X = [x_1 \ldots x_N]^T$. Therefore, a backup of x correspond to a synchronization between dependent variables x_i and x_j where $j \neq i$.

Assume that x is integrated every step h_k. As a first case, we consider that the value of the state variable x is saved every integration step in order to be available for the next computation. As a second case, we just consider the availability of an old saved (delayed) value at $n - \alpha$.

$$\textbf{Case 1:} \quad x(n+1) = x(n-\alpha) + \sum_{k=n-\alpha}^{n} h_k.\dot{x}(k) \tag{6}$$

$$\textbf{Case 2:} \quad \tilde{x}(n+1) = x(n-\alpha) + \sum_{k=n-\alpha}^{n} h_k.\dot{x}(n-\alpha) \tag{7}$$

The error resulted from the difference between (6) and (7) (see Figure 3) is represented by e:

$$\begin{aligned} e(n+1) &= x(n+1) - \tilde{x}(n+1) \\ &= \sum_{k=n-\alpha+1}^{n} h_k.(\dot{x}(k) - \dot{x}(n-\alpha)) \\ &= e(n) + h_n.(\dot{x}(n) - \dot{x}(n-\alpha)) \end{aligned} \tag{8}$$

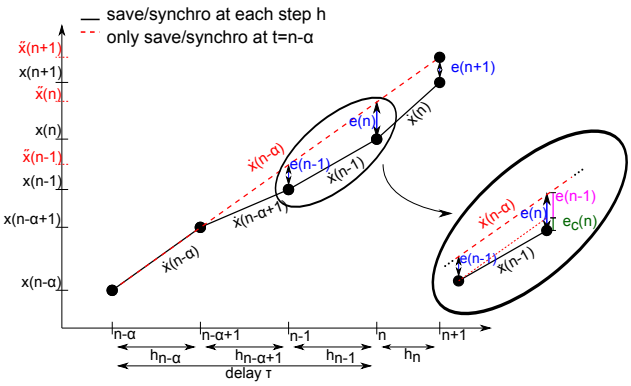


Figure 3. Errors due to delays

To show up the delay τ in the expression of e, (8) is rewritten in the temporal form

$$e(t_{n+1}) = e(t_n) + h_n.(\dot{x}(t_n) - \dot{x}(t_n - \tau)) \text{ where } \tau = t_n - t_{n-\alpha} \tag{9}$$

Let $e_c(t_{n+1}) = h_n.(\dot{x}(t_n) - \dot{x}(t_n - \tau))$. Here $e_c(t_{n+1})$ is the current error generated at t_{n+1}. So, the final error $e(t_{n+1})$ is the result of the accumulation of a past error $e(t_n)$ and the current error $e_c(t_{n+1})$. As a conclusion, two conditions must be met to achieve a correct result:

- $e_c(t_{n+1}) < \epsilon_{loc}$: allowed local error
- $e(t_{n+1}) < \epsilon_{glo}$: allowed global error

4.3 Evaluation of Delay Errors in a General Problem

The analysis and results made in the previous case (the basic problem) will be applied to the following larger problem with N state variables $X = [x_1 \dots x_N]^T$.

Assume that a given system Σ is split to two separated subsystems Σ_A and Σ_B with state vectors X_A and X_B such as $X_A \cup X_B = X$ and $X_A \cap X_B = \oslash$. Then assume that they are integrated with a variable time-step h_k for X_A and $h_{k'}$ for X_B. Besides, they are synchronized every period P, where $P >> h_k$, in order to exchange some updated data.

The aim is to find an evaluation of the error, caused by the lack of an updated data during a delay τ, at any given time t_k for X_A and $t_{k'}$ for X_B, which may be a synchronization time or to an integration time between two checkpoints.

Thereby, the delay τ is represented by the difference between the current integration time t_k and the last synchronization time t_s as follow

$$\tau = t_k - t_s \tag{10}$$

where t_s is computed as follow

$$t_s = P.\left\lfloor \frac{t_k}{P} \right\rfloor \tag{11}$$

in order to have

$$t_s = \begin{cases} l.P & when \quad t_k = l.P \quad l \in \mathbb{N}^* \\ (l-1).P & when \quad t_k < l.P \quad l \in \mathbb{N}^* \end{cases} \tag{12}$$

which leads to

$$\begin{cases} \tau = 0 & when \quad t_k = t_s \\ \tau > 0 & when \quad t_k > t_s \end{cases} \tag{13}$$

Therefore, the resulted error at t_{n+1} in the subsystem X_A denoted by $E_A(t_{n+1})$ will be the difference between $X_A(t_{n+1})$ and $\tilde{X}_A(t_{n+1})$ deduced from (14) and (15).

$$X_A(t_{k+1}) = X_A(t_k) + h_k.f_A(X_A(t_k), X_B(t_k)), \quad k \in \{0, \dots, n\} \tag{14}$$

$$\tilde{X}_A(t_{k+1}) = \begin{cases} X_A(t_{k+1}) & k = 0 \\ \tilde{X}_A(t_k) + h_k.f_A(\tilde{X}_A(t_k), \tilde{X}_B(t_k - \tau)) & k \geq 1 \end{cases} \tag{15}$$

In other words,

$$\begin{aligned} E_A(t_{n+1}) &= \sum_{k=0}^{n} E_A(t_k) \\ &+h_n.[f_A(X_A(t_n), X_B(t_n)) - f_A(\tilde{X}_A(t_n), \tilde{X}_B(t_n - \tau))] \\ &= E_{A,p}(t_n) + E_{A,c}(t_{n+1}) \end{aligned} \tag{16}$$

where

$$\begin{aligned} E_{A,c}(t_{n+1}) &= h_n.[f_A(X_A(t_n), X_B(t_n)) - f_A(\tilde{X}_A(t_n), \tilde{X}_B(t_n - \tau))] \\ E_{A,p}(t_n) &= \sum_{k=0}^{n} E_A(t_k) \end{aligned} \tag{17}$$

Here $E_{A,c}(t_{n+1})$ is the current error generated at t_{n+1} whatever a synchronization or not. So, the global decoupling error $E_A(t_{n+1})$ is the result of the accumulation of a past errors $E_{A,p}(t_n)$ and the current error $E_{A,c}(t_{n+1})$.

As a conclusion, to achieve a correct result, two conditions must be met for the current (local) error and the global error:

- $E_{A,c}(t_{n+1}) < \epsilon_{loc}$
- $E_A(t_{n+1}) < \epsilon_{glo}$

These conditions can be satisfied by acting on some parameters. In fact, in (17), the delay error depends on the integration steps h_k and on the delay τ. The delay τ depends on the last synchronization time t_s, which itself depends on the synchronization period P, and on the current integration time t_k. In other word, the delay error depends on the size and number of integration steps since the last synchronization. Indeed, the step size gives an idea of the sharpness of the state variations, and numerous small integration steps denotes large variations between successive updates. Fast variations of the state variables are not only related to the system stiffness, but also especially to the presence of discontinuities. Therefore delay errors are strongly related with discontinuities, whose location and dependencies deserve to be carefully examined when splitting the system.

In addition, the delay error size increases with the number of dependencies between the subsystems. Obviously, if X_A is independent from X_B, no synchronization error can be generated. If conversely X_A depends on all the state variables of X_B, the errors may increase considerably. Thus the objective is to choose cuts in the system to minimize the data exchanges and to control the growth of the integration errors.

4.4 Delay Errors in a Real-time Simulation

Up-to-now, the delay errors are evaluated in the case where the integration of all subsystems was exactly finished at the synchronization point. In fact we consider that the synchronization points are deadlines, without worrying about what happens inside at each integration step. The previous study just treated the case of hard real-time constraints where all the deadlines are met.

To speed-up the simulation, waiting periods should be eliminated. These idle times occur at the synchronization points when one or more sub-systems are waiting for the end of integration of the other sub-systems. Therefore, this time can be decreased by using slackened synchronization [5]. It means that, at the synchronization points, a subsystems which needs variable does not wait for its integration to be completed, but uses the last available value. Indeed this relaxation induces a cost in term of delay error, which must be re-evaluated to check the trade-off between simulation speed and accuracy.

Assuming that only the last integration step before the synchronization can be missed, the value and production time of the last completed integration step before the synchronization, (denoted t_p for X_A and $t_{p'}$ for X_B) must be known: $t_p = t_s - h_p$.

Assume that the synchronization deadlines are always missed, so that the error E_A is always computed from the

last completed integration step. Thereby, the delay τ computed beforehand in (10) is no longer valid but is as follow:

$$\tau = t_k - t_p \tag{18}$$

In other words, instead of $\tilde{X}_B(t'_k - \tau) = \tilde{X}_B(t_s)$, now $\tilde{X}_B(t'_k - \tau) = \tilde{X}_B(t'_p)$ leading to

$$\begin{aligned} E_{A,c}(t_{n+1}) &= h_n.[f_A(X_A(t_n), X_B(t_n)) - f_A(\tilde{X}_A(t_n), \tilde{X}_B(t'_p))] \\ E_{A,p}(t_n) &= \sum_{k=1}^{n} E_A(t_k) \end{aligned} \tag{19}$$

Here the two equations (17) and (19) are equivalent, the only difference lies in the expression of τ.

5. System Splitting Using Block-diagonal Forms

As mentioned before, the purpose is to optimize the exploitation of the parallelism of the sub-systems while keeping the previously evaluated delay error due the decoupling under control. Two methods have been analyzed for this aim, the first is related to the states to reduce the data-flow due to coupling variables between sub-systems. The second one is related to the events, to reduce integration interrupts, and also to minimize event detection and location via a complementary kind of parallelization through the solver.

5.1 Accounting for the State Variables

To reduce the data exchange between two sub-models and to prioritize these swaps inside one sub-model, the dependencies between the state variables must be evaluated. It can be done either by a direct access to the incidence matrix that describes the coupling between the state variables and their derivatives, or by computing the Jacobian matrix.

A Jacobian matrix is a matrix of all first-order partial derivatives of a vector function $f = [f_1 f_2 \dots f_N]^T$ regarding another vector $X = [x_1 x_2 \dots x_N]^T$. An $N \times N$ Jacobian matrix denoted by J has the form:

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_N}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_N}{\partial x_N} \end{pmatrix}$$

If there is a zero element in the Jacobian, i.e. $\frac{\partial f_i}{\partial x_j} = 0$, it means that f_i is not influenced by x_j. However, f_i is actually $\dot{x}_i$. In other words, x_i does not depend on x_j. In the same way if $\frac{\partial f_i}{\partial x_j} \neq 0$, it means that x_i depends on x_j. Moreover, the numerical value of $\frac{\partial f_i}{\partial x_j}$ gives a measure of the sensitivity of $\dot{x}_i$ w.r.t. x_j.

This leads to conclude that the Jacobian matrix can be seen as an incidence matrix which provides useful information about data dependencies between state variables. This could be used for an effective system splitting. So that, when transforming the matrix into a block-diagonal form by permuting rows and columns, the blocks represents the independent subsystems. It may happen that a total block-diagonalization is not possible so that the final transformed matrix presents some coupling rows and/or column, this denotes the presence of irreducible dependencies between subsystems.

5.2 Accounting for the Discontinuities

To minimize the delay error while optimizing the exploitation of the parallelism across the model, it is also crucial to reduce the number of discontinuities inside each sub-model, so that stiff variations of the state variables are limited. This procedure induces another benefit, as reducing the number of interrupts for each solver reduces re-starting overheads and improves the integration speed.

The events incidence matrix describes the relationships between events. Block-diagonalizing this matrix allows for separating the discontinuities and scatter them in the different sub-models.

Furthermore, the events incidence matrix block-diagonalization also leads to a kind of parallelization across the solver. In fact, the system resolution, including events handling, consists of 4 steps as mentioned in Figure 4.

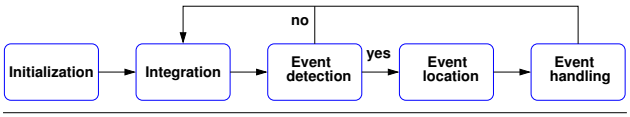


Figure 4. Events handling operations flow

Event detection and location can be an expensive stage for hybrid systems (and for the addressed combustion models in particular). Especially, the event location (i.e. solving the zero-crossing equation (2)) can take a long time through an iterative process, and it is difficult to bound this step. By using the event incidence matrix, solving for a particular event can be localized in a subset of the global system through parallelization, thus shortening the zero-crossing function solving.

5.3 Permuting Sparse Rectangular Matrices for Block-diagonal Forms

Two methods and associated software tools have been evaluated to perform the system diagonalization. Note that the original state variables of the system are preserved and that diagonal forms are produced only through permutations.

5.3.1 Bipartite Graph Model

A matrix A is transformed to a bipartite graph model. This graph is used by a specific tool to partition it, then to get a doubly bordered block-diagonal matrix A_{DB}, i.e. the matrix has a block-diagonal form with non-zero elements on its last rows and columns as in Figure 5.

MeTiS [13] is a software aimed to partition large graphs. The used algorithms are based on multilevel graph partitioning, which means reducing the size of the graph by collapsing vertices and edges, then partitioning the smaller graph, and finally uncoarsening it to construct a partition for the original graph.

The block-diagonal form is performed by permuting rows and columns of a sparse matrix A to transform it into a K-way doubly bordered block-diagonal (DB) form A_{DB}. It has a coupling row and a coupling column.

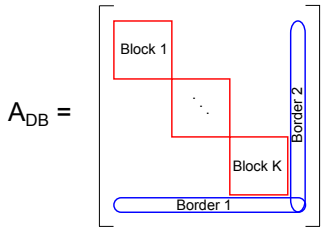


Figure 5. Doubly bordered block-diagonal matrix

The representation of the nonzero structure of a matrix by a bipartite graph model reduces the permutation problem to those of graph partitioning by vertex separator (GPVS).

For example, let A the following matrix:

$$A = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad (20)$$

An undirected graph $G = (V, E)$ is defined as a set of vertices V and a set of edges E. The corresponding bipartite graph for MeTiS is built by replacing the rows and the columns by vertices and the non-zeros are represented by edges. After transformation, MeTiS partitions the graph as shown in Figure 6.

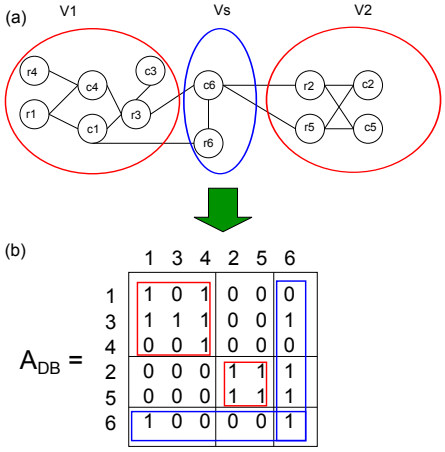


Figure 6. (a) Bipartite graph representation of the matrix A and 2-way partitioning of it by vertex separator Vs; (b) 2-way DB form of A induced by (a)

The objective of MeTiS when partitioning is to:

- Minimize the size of the separator because it implies the minimization of the border size.
- Balance among sub-bipartite graphs because it implies a balance among diagonal sub-matrices.

5.3.2 Hypergraph Model

A matrix A is transformed to a hypergraph model. An hypergraph $H = (U, N)$ is defined as a set of nodes (vertices) U and a set of nets (hyper-edges) N among those vertices. This hypergraph is used by a specific tool to partition it, then to get a singly bordered block-diagonal matrix A_{SB} as in Figure 7, where the matrix has a block-diagonal form with non-zero elements only on its last rows.

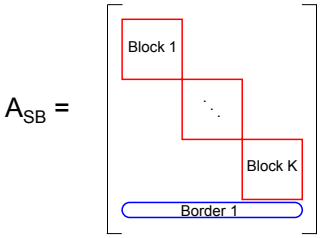


Figure 7. Singly bordered block-diagonal matrix

PaToH (Partitioning Tools for Hypergraphs) [19] is a multilevel hypergraph partitioning tool that consist of 3 phases as for MeTiS: coarsening, initial partitioning, and uncoarsening. In the first phase, a multilevel clustering, that correspond to coalescing highly interacting vertices to super-nodes, is applied on the original hypergraph by using different matching heuristics until the number of vertices drops below a predetermined threshold value. Then, the second phase corresponds to partition the coarsest hypergraph using diverse heuristics. Finally, in the third phase, the obtained partition is projected back to the original hypergraph by refining the projected partitions using different heuristics.

The block-diagonal form is performed by permuting rows and columns of a sparse matrix A in order to transform it into a K-way singly bordered block-diagonal (SB) form A_{SB}. It has only a coupling row. For this reason, this method of block-diagonalization will be selected for the later study.

The representation of the nonzero structure of a matrix by an hypergraph model reduces the permutation problem to those of hypergraph partitioning (HP).

The corresponding hypergraph graph of the matrix A (20) for PaToH is build by replacing the rows and the columns of the matrix by nets and nodes respectively. The number of pins is equal to the number of non-zeros in the matrix. After the transformation, PaToH partitions the hypergraph as as it is shown in Figure 8.

The objective of PaToH when partitioning is to:

- Minimize the cut size because it implies the minimization of the number of coupling rows.
- Balance among sub-hypergraphs because it implies a balance among diagonal sub-matrices.

In conclusion, the method using the bipartite graph model as MeTiS generates a doubly bordered block-diagonal matrix. To further reduce the coupling row and the coupling column to a single coupling row, the Ferris-Horn (FH) algorithm [10] uses a column splitting method. Unfortunately, the number of rows and columns of the matrix must be increased. In contrast, the method using the hypergraph model as PaToH directly generates a singly bordered block-diagonal matrix which means only a coupling row without adding an intermediate method. Therefore PaToH

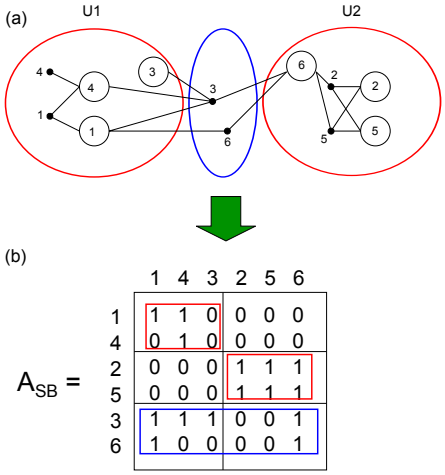


Figure 8. (a) Row-net hypergraph representation of the matrix A and 2-way partitioning of it; (b) 2-way SB form of A induced by (a)

will be used for the block-diagonalization of matrices in the following case study.

6. Analysis of System Splitting using an Hypergraph Model through a Case-study

In this study, a Spark Ignition (SI) mono-cylinder engine has been modeled (see Figure 9) with 3 gases (air, fuel and burned gas). It was developed using the ModEngine library [3]. ModEngine is a Modelica [11] library that allows for the modeling of a complete engine with diesel and gasoline combustion models. It contains more than 250 sub-models. A variety of elements are available to build representative models for engine components. ModEngine is currently functional in the Dymola tool [1].

6.1 Engine Modeling

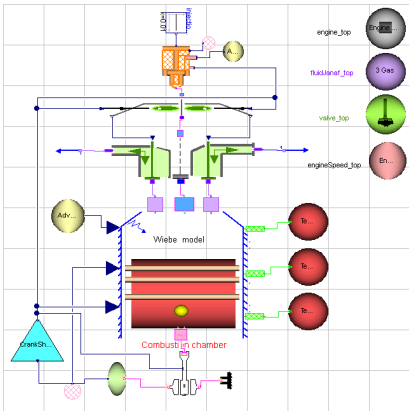


Figure 9. Mono-cylinder engine modeled in Dymola

In the following tests, relationships between state variables and events as well as their behaviors are essential to study how to split the system at wisely chosen joints.

For this aim, the mono-cylinder model, written in Modelica language, was translated to a simpler language called Micro-Modelica (μ-Modelica) [4], which is understandable by the stand-alone Quantized State Systems (QSS) tool [14] as shown in Figure 10. The QSS solver is not used here, only a related tool is used to generate a so-called simulation file which contains important information about the system and relationships between states and events. These data are extracted thereafter by a custom dedicated tool, and translated both to a matrix form for visualization and to an hypergraph file for the PaToH tool. Finally PaToH generates a partitioned hypergraph file that describes how the graph is decomposed and transformed subsequently to a matrix form for visualization.

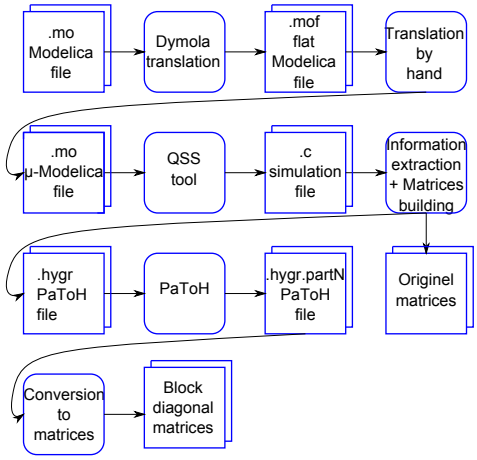


Figure 10. Software tool-chain

The considered mono-cylinder model is characterized by a number of:

- State variables: $n_X = 15$
- Events: $n_Z = 111$
- Discrete variables: $n_D = 93$

The state variables x_i $(i = 1, .., n_X)$ are defined as follows:

ID	Name	Details
X_0	CrankAngle	Crank Shaft angle
$X_{1\to3}$	mEvapo[3]	Gas mass evaporated due to injection in global Mass balance equation
X_4	qvAlfa	Current released heat generated by the combustion process
X_5	mrefAlfa	Burned mass fraction during combustion process
X_6	combuHeatRelease	Output current combustion heat released
$X_{7\to9}$	mCombu[3]	Gas mass derivatives due to combustion in global Mass balance equation
$X_{10\to12}$	M[3]	Mass of gas
X_{13}	Energy	Energy contained in the cylinder
X_{14}	cylinderTemp	Output temperature in the cylinder

[1] http://www.3ds.com/products/catia/portfolio/dymola

The events z_i ($i = 1, .., n_Z$) and discrete variables d_i ($i = 1, .., n_D$) are defined by the "when" blocks as follows:

```
when (z_i) then
 d_i = ... ;
elsewhen !(z_i) then
 d_i = ... ;
end when;
```

The statements that are between the "then" and the "elsewhen" or the "end when" are called the event handler, it represents the consequence of the event.

6.2 State and Derivatives Incidence Matrices

At first glance, the number of coupled state variables is 6 among 15. In fact, $\dot{X}_{13}$ is only influenced by the state variables X_0, X_{10}, X_{11}, X_{12}, X_{14} as shown in Figure 11.

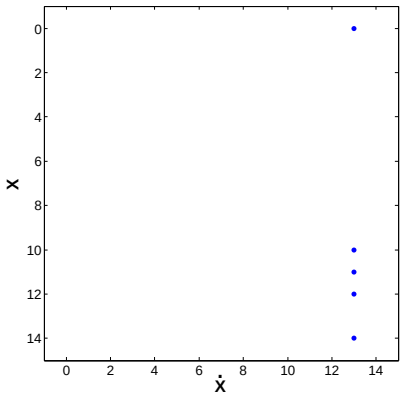


Figure 11. Incidence state matrix: derivatives of state variables $\dot{X}$ depending on state variables X

Thus far, considering only the incidence state matrix, only 40% of the state variables are directly computed from the other states, while the others depend on external inputs (or even remain constant on some particular trajectories of the state space, e.g. when imposing a constant velocity of the crank).

The same result is found for events. In fact, the number of active events is 39 among 111, as the previous cited involved state variables directly affect values of 39 events as shown in Figure 12.

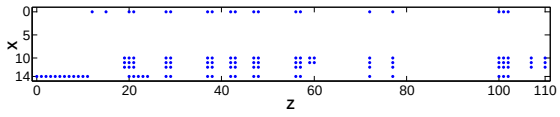


Figure 12. Incidence matrix: events Z depending on the state variables X

This number represents only 35% of the total number of events, while the rest is only used to activate other events. In fact these 72 events are defined in the ModEngine library to be used in more general systems, not for the particular mono-cylinder use case. In consequence only the subset of active events must be detected.

However, if the state variables X can affect the events Z, the events can also change the state variables values. In order to construct its corresponding matrix, both the incidence matrix that defines the discrete variables D influenced by the events Z: $Z \rightarrow D$ (see Figure 13) and the incidence matrix that defines the derivatives of the state variables $\dot{X}$ influenced by the discrete variables D: $D \rightarrow \dot{X}$ (see Figure 14) are carried out.

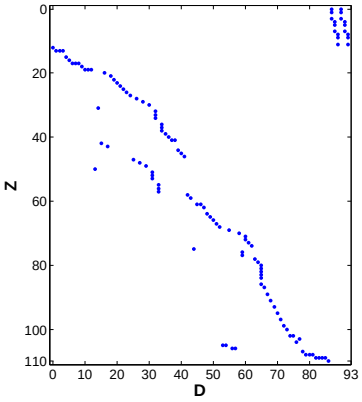


Figure 13. Incidence matrix: discrete variables D influenced by the events Z

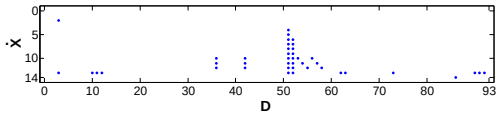


Figure 14. Incidence matrix: derivatives of state variables $\dot{X}$ influenced by discrete variables D

Thus the incidence matrix $Z \rightarrow \dot{X}$ is deduced by transitivity in Figure 15.

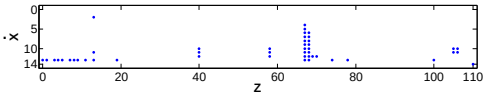


Figure 15. Incidence matrix: derivatives of state variables $\dot{X}$ influenced by the events Z

Figure 15 shows that the previous identification of some state variables as not coupled (based on incidence state matrix Figure 11) and events influenced by state variables (Figure 12), is no longer true. In fact, now 13 state variables among 15 appear in this incidence matrix. Note that now only the state variables corresponding to X_1 and X_3 do not appear in this incidence matrix, this is due to the fact that these variables are inhibited momentarily to test a particular scenario.

By combining the two matrices in Figures 12 and 15, an incidence matrix between events and state variables can be achieved as in Figure 16.

Once unnecessary states and events are eliminated and only involved ones are kept, the intrication between state

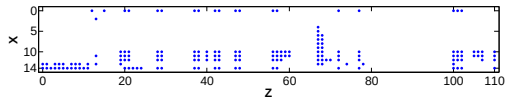


Figure 16. Incidence matrix: data exchange between events Z and state variables X

variables and events in both directions shows that it is difficult to separate or split the system.

Besides, from Figure 12 ($X \rightarrow Z$) and Figure 15 ($Z \rightarrow \dot{X}$), the state incidence matrix can be built differently than in Figure 11, by passing through the events as it is shown in Figure 17.

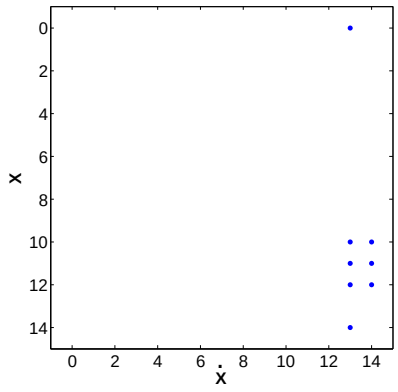


Figure 17. Incidence state matrix: derivatives of state variables $\dot{X}$ influenced by state variables X

Using this construction through the events Z, it appears that the state derivative $\dot{X}_{14}$ is also depending on X_{10}, X_{11} and X_{12}. Therefore, in order to determine correctly the relationships between the variables, it is important to use all the available system data, directly and by transitivity.

6.3 Incidence Event Matrix

The incidence event matrix can be built by transitivity. In fact, using the incidence matrix that define $Z \rightarrow D$ (see Figure 13) and conversely the matrix that define $D \rightarrow Z$, the incidence event matrix $Z \rightarrow Z$ can be deduced as it is shown in Figure 18.

As shown previously, it is hard to split the system based on the relationship between events Z and discrete variables D. However, with the incidence event matrix, it is possible to transform it into a block-diagonal form with three blocks using PaToH and to consider each block as a subsystems where all the related discontinuities belong to the same entity (see Figure19).

These blocks can be parallelized and we can hope the execution time to be reduced. In fact, the event detection, the event location and the restart of the solver increase the integration time as shown in Figure 20. In short, for the mono-cylinder integrated by the variable-step solver LSODAR, the average execution speed drops down to 4 times in case of events handling, and sometimes even up

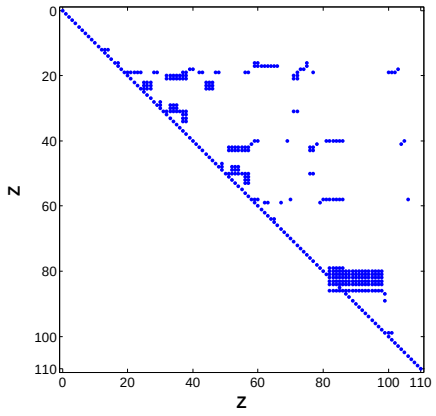


Figure 18. Incidence event matrix: events Z in columns influenced by events Z in rows

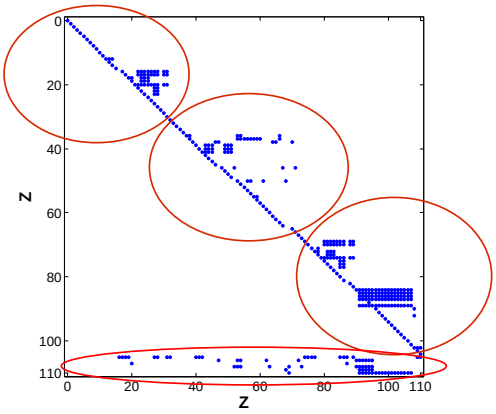


Figure 19. Block-diagonalized incidence event matrix: events Z in columns influenced by events Z in rows

to 60 times. This confirms the interest on both limiting the number of interrupts inside each block of the model due to the events and parallelizing the event location through the solver.

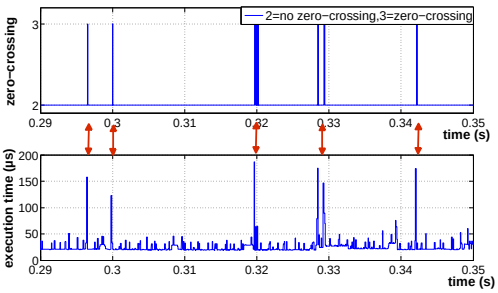


Figure 20. Effect of events handling on execution time

7. Summary and Future Directions

The various methods studied in the paper aim to contribute to speed-up the numerical integration of hybrid dynamical systems, eventually until reaching a real-time execution, while keeping the integration errors inside controlled bounds. Speed-ups can be achieved through an adequate partition of the original system where the interactions between the resulting sub-systems are minimized, so that they can be efficiently integrated in parallel.

Slackened synchronization, as analyzed theoretically in Section 4, allows for minimizing the number of integration interrupts and for using optimized integration parameters on each node (as illustrated experimentally in [2]) . However the corresponding induced delays between the subsystems must be limited to keep the integration errors under control.

The particular case study shows that it is not easy nor intuitive to know how to split a system, neither from a physical point of view nor from the relationship between the states and the events. In fact, the matrix between the coupled states and events is not sparse, so it is not possible to transform it into a block-diagonal form.

However, the incidence events matrix more likely seems to be sparse and its transformation to a block-diagonal form is feasible. Thus a relevant way to parallelize this particular system seems to perform it through the solver, leading to parallelize the steps corresponding to events handling which are costly for the numerical resolution (as already observed and plot in Figure 20).

Future works intend to practically evaluate the achievable speedups. This requires to extend the tool-chain of Figure 10, by developing a multi-thread runtime system able to take into account the parallelization choices presented in this paper. Then an engine with 4 cylinders will be studied to compare the split performed from a physical point of view in [2] to the combined use of the analytic approaches described in sections 4 and 5.

References

[1] U. M. Ascher and L. R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM, Philadelphia, PA, USA, 1st edition, 1998.

[2] A. Ben Khaled, M. Ben Gaïd, D. Simon, and G. Font. Multicore simulation of powertrains using weakly synchronized model partitioning. In *E-COSM'12 IFAC Workshop on Engine and Powertrain Control Simulation and Modeling*, Rueil-Malmaison, France, October 2012.

[3] Z. Benjelloun-Touimi, M. Ben Gaid, J. Bohbot, A. Dutoya, H. Hadj-Amor, P. Moulin, H. Saafi, and N. Pernet. From physical modeling to real-time simulation : Feedback on the use of modelica in the engine control development toolchain. In *8th Int. Modelica Conference*, Germany, March 2011. Linköping University Electronic Press, Linköpings universitet.

[4] F. Bergero, X. Floros, J. Fernández, E. Kofman, and F. E. Cellier. Simulating Modelica models with a stand-alone quantized state systems solver. In *9th Int. Modelica Conference*, Munich Germany, September 2012.

[5] G. Bernat, A. Burns, and A. Llamosi. Weakly hard real-time systems. *IEEE Trans. on Computers*, 50:308–321, April 2001.

[6] T. Blochwitz, T. Neidhold, M. Otter, M. Arnold, C. Bausch, M. Monteiro, C. Clauß, S. Wolf, H. Elmqvist, H. Olsson, A. Junghanns, J. Mauss, D. Neumerkel, and J.-V. Peetz. The functional mockup interface for tool independent exchange of simulation models. In *Proceedings of the 8th International Modelica Conference*. Linköping University Electronic Press, March 2011.

[7] G. D. Byrne and A. C. Hindmarsh. PVODE, an ODE solver for parallel computers. *International Journal of High Performance Computing Applications*, 13(4):354–365, Winter 1999.

[8] F. E. Cellier and E. Kofman. *Continuous System Simulation*. Springer, 1st edition, March 2006.

[9] C. Faure, M. Ben Gaïd, N. Pernet, M. Fremovici, G. Font, and G. Corde. Methods for real-time simulation of cyber-physical systems: application to automotive domain. In *IWCMC'11*, pages 1105–1110, 2011.

[10] M. C. Ferris and Jeffrey D. Horn. Partitioning mathematical programs for parallel solution. *Mathematical Programming*, 80:35–61, 1998.

[11] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica*. Wiley-IEEE Press, 2003.

[12] D. Guibert. *Analyse de méthodes de résolution parallèles d'EDO/EDA raides*. PhD thesis, Université Claude Bernard - Lyon I, Sep 2009.

[13] G. Karypis and V. Kumar. MeTiS : A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Technical report, Univ. of Minnesota, Dept. of Computer Science, 1998.

[14] E. Kofman and S. Junco. Quantized-State Systems: a DEVS approach for continuous system simulation. *Trans. of The Society for Modeling and Simulation International*, 18(3):123–132, September 2001.

[15] E. A. Lee. Computing foundations and practice for Cyber-Physical Systems: A preliminary report. Technical Report UCB/EECS-2007-72, Univ. of California, Berkeley, May 2007.

[16] M. Sjölund, R. Braun, P. Fritzson, and P. Krus. Towards efficient distributed simulation in Modelica using transmission line modeling. In *EOOLT*, pages 71–80, 2010.

[17] P. J. van der Houwen and B. P. Sommeijer. Parallel iteration of high-order Runge-Kutta methods with stepsize control. *J. Comput. Appl. Math.*, 29:111–127, January 1990.

[18] F. Zhang, M. Yeddanapudi, and P. Mosterman. Zero-crossing location and detection algorithms for hybrid system simulation. In *Proc. 17th IFAC World Congress*, pages 7967–7972, Seoul, South Korea, July 2008.

[19] Ü. V. Çatalyürek. *Hypergraph Models for Sparse Matrix Partitioning and Reordering*. PhD thesis, Computer Engineering and Information Science Bilkent University, November 1999.

Automating Dynamic Decoupling in Object-Oriented Modelling and Simulation Tools

Alessandro Vittorio Papadopoulos Alberto Leva

Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy,
{papadopoulos,leva}@elet.polimi.it

Abstract

This manuscript presents a technique that allows Equation-based Object-Oriented Modelling Tools (EOOMT) to exploit Dynamic Decoupling (DD) for partitioning a complex model into "weakly coupled" submodels. This enhances simulation efficiency, and is naturally keen to parallel integration or co-simulation. After giving an overview of the problem and of related work, we propose a method to automate DD by means of a novel structural analysis of the system – called "cycle analysis" – and of a mixed-mode integration method. Also, some considerations are exposed on how the presented technique can be integrated in EOOMT, considering as representative example a Modelica translator. Simulation tests demonstrate the technique, and the realised implementation is released as free software.

Keywords dynamic decoupling, model partitioning, efficient simulation code generation

1. Introduction and Motivation

Equation-based Object-Oriented (EOO) modelling languages are known to possess a number of interesting advantages, and in the context of this work, two are particularly relevant. First, the EOO modelling paradigm is inherently suited for building modular, multi-physic models. Second, the model designer has not to take care of how the system will be simulated, just focusing on how to write the equations of its components. In one word, with EOO Modelling Tools (EOOMT) one handles the complete model by just aggregating components and acting on them. The translator included in typical EOOMT is then in charge of manipulating all the gathered equations, and producing efficient simulation code [5, 8].

As long as the obtained simulation efficiency is sufficient, the possibility of managing complexity at the component level has practically no cost. However, there are some cases where to achieve the desired efficiency, approximations need introducing, and EOOMT are neither meant nor suited for that. As will be discussed in this work, approximations can be introduced either by altering some equations in some components, or by acting on the numerical solution of the complete model. And while with EOOMT the first action is natural, the second is not at all. This rules out several powerful approximation techniques aimed at enhancing simulation speed, e.g., the Dynamic Decoupling (DD) one [2–4] treated herein.

To enter the subject, it is convenient to specify why introducing approximation at the level of the solution is "unnatural" in EOO modelling. The main reason is that the possibility/opportunity of doing so depends on properties of the *whole* model, not of the individual components, and in the typical toolchain of EOOMT no user interaction is envisaged at that point. Moreover, assuming that the use of any approximation technique requires some parameters, it is necessary to provide the user with the necessary information to give them a value, and to accept his/her choices, in a comprehensible manner, manageable by people who are more experts of physics than of simulation theory.

In this work we refer as "EOO Modelling Tool" to a Modelica translator, to allow exemplifying the (more general) presented ideas. For a Modelica translator, the EOO modelling toolchain can be synthetically depicted as in Figure 1.

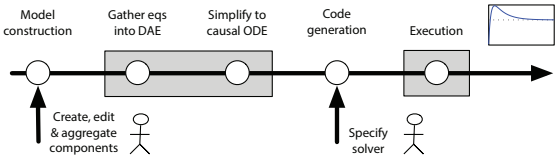


Figure 1. The typical EOO modelling toolchain.

For our purposes, said toolchain has to be extended – and in some sense "opened" – as suggested in Figure 2, introducing some (clearly optional) automatic system-wide analysis, and taking care of having the user interact with simple enough information despite operating at the whole system level. In this work we present a solution assuming that the desired type of approximation is DD, therefore tailoring the analysis and the use of the produced information to that case, but nonetheless the way of acting on the toolchain is general with respect to the approximation type.

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. 19 April, 2013, University of Nottingham, UK.

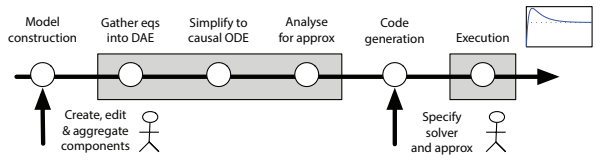


Figure 2. Extending the EOO modelling toolchain.

The rest of the manuscript is organised as follows. Section 2 reviews some relevant literature and provides a more detailed motivation for the presented work, specialising to the DD technique. Section 3 is devoted to the DD technique, and particularly to how it is structured – into an analysis and a simulation part – so as to be applicable to the addressed context. A few illustrative examples are then reported in Section 4. Some more general considerations are made in Section 5 on how to integrate the presented technique in modern EOOMT, ending with some details on an implementation offered as free software to the community. Finally, Section 6 draws some conclusions, and sketches out future research developments.

2. Related Work and Contribution

In this section, we motivate the presented research by relating DD to other approximation techniques for improving simulation efficiency, with specific emphasis on their applicability and convenience in EOOMT.

Referring to Figure 1, the chain of operations of EOO modelling translators, from component equations to simulation code, can be broadly divided into two parts.

The first part, which we call *acting on the continuous-time equations*, transforms the DAE system coming from the flattening phase, into a causal ODE one. This can be done without altering the equations' semantic, by resorting to techniques, such as the Tarjan algorithm, alias elimination, index reduction and so forth [5]. The same operation can also be done by accepting some semantic alteration in exchange for an efficiency improvement. The techniques of election for such a purpose are, e.g., MOR [1] or scenario-based [13] approximations.

The second part of the EOO modelling toolchain, which we call *acting on the discrete-time solution*, consists of taking the mentioned ODE model as the basis to generate suitable routines that, once linked to the numeric solver of choice, result in the required simulation code. Assuming that acting on the discrete-time solution is done "correctly", i.e., preserving numerical stability, also in this case two ways of operating can be distinguished. The first way does not alter the solution semantic, and the chosen discretisation method is applied as is. In this case, errors in the solution only come from the inherent imperfection of the considered method. The second way conversely alters the semantic of the discrete-time system, by deliberately deviating from the natural application of the chosen discretisation method. Notice that most of the co-simulation techniques fall in this class naturally.

In this manuscript we concentrate on this last type of operation, where a technique of election is DD [2, 14]. For the purpose of this section, suffice to say that this technique aims at partitioning the monolithic system into submodels, based on time-scale separation. The method is particularly of interest – as will be detailed better in Section 3 – because it can be divided into two well separated phases: an analysis part performed on the overall model, and a simulation part that either can be monolithic or makes use of co-simulation technique.

To motivate the choice of focusing on DD, we now briefly consider the major possible alternatives, and evidence the advantages of our proposal.

2.1 Alternatives Approaches

As already stated, among the techniques that act on the continuous-time equations, MOR ones are the most adopted, and there exists a vast literature on the matter. MOR is based on the idea of approximating a certain part of a high-dimensional state space of the original system with a lower-dimensional state space, performing a projection. Very roughly speaking, the main differences among MOR techniques come from the way the projection is performed.

Most MOR techniques have been developed for linear systems [1], and this hampers their application to object-oriented models, that usually are high-dimensional and nonlinear; developing effective MOR strategies for nonlinear systems is quite a challenging and relatively open problem.

In the literature, some extension to the nonlinear case are present, e.g., based on linearisation or Taylor expansion [7], or bilinearisation [15], as well as functional Volterra series expansion [10], followed by a suitable projection. Other interesting extensions worth mentioning are those based on Proper Orthogonal Decomposition (POD) [6], that produce approximate truncated balanced realisations for nonlinear systems [16], often exploiting POD to find approximate Gramians [11]. The main problem with those extensions is that, of the former, practical implementations typically stick to quadratic expansions, strongly limiting the simplification capabilities. As for the latter, the cost of evaluating the projected nonlinear operator often remains very high, and reduces computational performance.

Recently, other works dealing with model reduction specifically conceived for object-oriented models have appeared [12, 13]. The main idea is that one can define some operation to be performed on the nonlinear system, e.g., neglecting a "term", linearise a part of the model, and so on, and use some ranking metrics to identify *a priori* which is the "best" (single) manipulation that can be done on the model. Apparently, the limit of this approach lies in the fact that ranking all the possible manipulation combinations is not feasible. Moreover, there is no guarantee that performing the manipulations in the ranked order will bring to the optimal manipulation. Another problem is the high cost of generating the reduced order models, due to necessity of computing "snapshots" in the time domain, which in turn requires performing numerous simulations of the original nonlinear system. Furthermore, this approach is scenario-based, i.e., the simplified model is guaranteed to be good only for a set of initial conditions, a set of inputs and a time

span. If the scenario is changed, the overall manipulation must be performed again, limiting again the applicability of the method.

The old idea of DD has also been recently reconsidered, for example by the Transmission Line Modelling (TLM) approach of [18]. This, however requires that the analyst introduces decoupling by deliberately acting on the model based on his/her intuition. This work conversely aims at having decoupling emerge from an automated analysis of the model.

2.2 A Brief Comparison

Based on the previous discussion, we now point out the advantages of the proposed technique with respect to the analysed alternatives.

In comparison with MOR, our proposal does not alter the state vector, nor does it involve base changes in the state space. Also, instead of attempting to simplify the model in a view to monolithic solution, we go exactly in the opposite direction, as the model is not reduced but *partitioned*. This can in turn be exploited in two ways. One is to ease a monolithic solution, in some sense adapting the model to the used (single solver) architecture. The other is to conversely tailor the solution architecture to the model *as analysed and partitioned by the method*; this can be used to fruitfully employ parallel simulation, or even co-simulation. Finally, the proposed method is naturally keen to be applied in a nonlinear context.

With respect to scenario-based approximations, the most computing-intensive part of the proposal (as will be explained later on) is simply not scenario-based: information related to the considered *scenarii* come into play only at a later stage, and this separation results in lightening the computing effort. Furthermore, the proposal does not alter the model equations, thus being less exposed to the possible unpredictable effects of local modifications at the overall system level.

The next section will delve into details on the proposed technique, thereby providing evidence for the statements made so far.

3. Dynamic Decoupling

Multi-physics systems are usually made of parts evolving within different time-scales. For example, in mechatronic systems a "slow" mechanical part is often controlled by "fast" electric circuits or by a hydraulic drive. The underlying idea of DD is to find a way to separate the different dynamics present in the model and to numerically integrate them with a suitable mixed-mode method, in order to improve simulation efficiency.

DD is composed of two subsequent phases, termed here *analysis* and *decoupled integration*. The former consists in performing an offline structural analysis of the system and in identifying which are the time-scales involved in the model. The latter exploits the information coming from the analysis to improve simulation efficiency.

Both phases can be carried out with multiple techniques. For the analysis phase, we propose here a novel method, called *cycle analysis*, that carries most of the merit for the applicability of the entire technique to the nonlinear case. For the latter, we conversely resort to mixed-mode integration.

3.1 Cycle Analysis

Cycle analysis is based on the idea that explicit integration methods are the most suited to enhance simulation speed [5]. Their computational effort is relatively low and constant, and the number of calculations per step can be easily estimated. However, those methods show their limits when dealing with stiff-systems. For simplicity, in the following we consider the Explicit Euler (EE) integration method, but similar (and less restrictive) results can be obtained for other explicit single-step methods, e.g., Explicit Runge-Kutta of any order.

The main idea of the cycle analysis – and of DD, in general – is that in a causal ODE model, both each variable and each equation can be associated with a characteristic time-scale.

A first possible idea to achieve this is to perform an eigenvalue analysis of the linearised system, and to partition the state space, as spanned by the eigenvectors, on the basis of the corresponding time constants (or natural frequencies). However, this is not always a good idea, because it involves a coordinate transformation; if the linear system involved in the eigenvalue analysis comes from the linearisation of a nonlinear one, the management of that transformation results in additional computations at each integration step. Different criteria for state space partitioning are thus advisable, like that proposed in [17].

Coming to our proposal, consider the state space form of a continuous-time ODE system

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u})$$

that discretised with EE with an integration step h yields

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot \mathbf{f}\left(\mathbf{x_k}, \mathbf{u_k}\right). \tag{1}$$

Suppose now that the system is at an asymptotic stable equilibrium, i.e., $\mathbf{x_{k+1}} = \mathbf{x_k}$. If a small perturbation is applied to a single state variable $x_\mathbf{k}$, a transient occurs, and two things may happen:

1. the perturbation affects the other state variables, however without in turn re-affecting $x_\mathbf{k}$;
2. the perturbation, after some integration steps, re-affects $x_\mathbf{k}$.

In the first case, no numerical instability can occur, but in the second case there is a "dependency cycle" among some state variables that may lead to unstable behaviours, depending on how the perturbation propagates.

The proposed method detects the dependency cycles that are present in the system, and defines conditions under which the perturbation cannot lead to numerical instability.

The first step in cycle analysis is to build the dependency digraph $G = (N, E)$ associated with the ODE model. In particular, there is a node $n \in N$ for each state variable,

and the set of edges $E \subseteq N \times N$ is formed as

$$e_{i,j} = h \cdot \frac{\partial f_i}{\partial x_j}.$$

In other words, the Jacobian of the model corresponds to the adjacency matrix of the weighted graph. The weights come from (1) and characterise the way the perturbation propagates.

The second step is to detect the set $\mathcal{C}$ of all cycles contained in the digraph G. Unfortunately, the problem of finding all the cycles in a directed graph has complexity $\mathcal{O}\left(2^{|E|-|N|+1}\right)$, as shown by a vast research in the operation research domain [9, 19, 20]. This is of course a limitation with strongly connected graphs, but sparse ones are more common in real applications, especially if weak couplings exist. On the other hand, it is worth stressing that the cycle analysis must be performed only once for a given model, during an offline phase before the simulation is started. According to practical experience, spending some additional time to perform a structural analysis aimed at speeding up the simulation is an acceptable tradeoff, especially when the simulation is run many times. Anyway, in all the performed tests (with up to 100 state variables), the analysis phase always took less than one second on a notebook with a 2.4GHz Intel Core 2 Duo processor and 4GB of RAM.

At this point, for every cycle $c \in \mathcal{C}$ detected in G a *cycle gain* can be defined.

DEFINITION 1. *A* cycle gain μ_c *of a cycle* $c \in \mathcal{C}$ *is*

$$\mu_c = \prod_{x_i, x_j \in c} e_{i,j} = h^L \cdot \prod_{x_i, x_j \in c} \frac{\partial f_i}{\partial x_j}$$

where $e_{i,j}$ *are the edges involved in the cycles and* L *is the length of the cycle.*

The meaning of this gain is related to how the perturbation propagates. Starting from the computed cycle gains, for each cycle an inequality in the form

$$|\mu_c| \leq \alpha \quad \Rightarrow \quad 0 < h \leq \sqrt[L]{\alpha} \cdot \left| \prod_{x_i, x_j \in c} \frac{\partial f_i}{\partial x_j} \right|^{-\frac{1}{L}} \tag{2}$$

is written, where $\alpha > 0$ is a design parameter of the method, related – as discussed later on – to the required simulation accuracy. Suffice for now to say that lower values of α make the method less keen to consider a certain coupling "weak".

So far, each cycle has been associated with a constraint on the integration step, i.e., with an upper bound for the integration step which prevents the perturbation from producing unstable behaviours.

Finally, each variable x_i is associated with the most restrictive constraint on $\overline{h}_{x_i}$ among the set of cycles $\mathfrak{C}_{x_i} = \{c \in \mathcal{C} | x_i \in c\}$, i.e., formally

$$\begin{aligned} \overline{h}_{x_i} = \max \quad & h \\ \text{s.t.} \quad & h > 0, \\ & 0 < \overline{h}_i \leq \sqrt[L]{\alpha} \cdot \left| \prod_{x_j, x_k \in c} \frac{\partial f_j}{\partial x_k} \right|^{-\frac{1}{L}}, \forall c \in \mathfrak{C}_{x_i}. \end{aligned}$$

3.2 Decoupled Integration

The second phase of DD is decoupled integration, which exploits the partition coming from cycle analysis in a view to improve simulation efficiency. Among the various possible ways to do so, we consider here the use of a mixed-mode integration method. The underlying idea is that implicit methods are able to simulate stiff systems with larger integration periods, at the cost of solving a nonlinear set of algebraic equations at each step, while explicit ones are better in terms of performance but cannot deal with stiff systems equally well. Having separated the system in (at least) two parts with different time scales, it is possible to use an implicit method for the fast part(s), and an explicit one for the slow part(s), exploiting the advantages of both kinds of integration algorithms.

Coming back to the proposal, the discrete-time system associated with the continuous-time one reads

$$\begin{cases} \mathbf{x}^s_{\mathbf{k}+1} = \mathbf{x}^s_{\mathbf{k}} + h \cdot \mathbf{f}\left(\mathbf{x}^s_{\mathbf{k}}, \mathbf{x}^f_{\mathbf{k}}, \mathbf{u}_{\mathbf{k}}\right) \\ \mathbf{x}^f_{\mathbf{k}+1} = \mathbf{x}^f_{\mathbf{k}} + h \cdot \mathbf{f}\left(\mathbf{x}^s_{\mathbf{k}+1}, \mathbf{x}^f_{\mathbf{k}+1}, \mathbf{u}_{\mathbf{k}+1}\right) \end{cases}$$

showing that the fast and the slow parts are integrated with the Implicit Euler (IE) and the EE method, respectively.

This means that the fast component $\mathbf{x}^f_{\mathbf{k}+1}$ can be computed considering $\mathbf{x}^s_{\mathbf{k}+1}$ as an input. Figure 3 shows the resulting mixed-mode integration scheme.

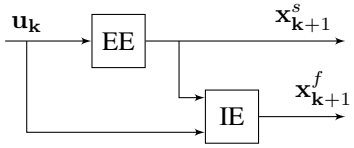


Figure 3. Explicit/Implicit Euler integration scheme.

4. Application Examples

4.1 DC Motor

The DC motor is a very simple example of system with two well separated time scales (electric and mechanic), and can be represented by a third order model in the form

$$\begin{cases} L \cdot \dot{I} &= -R \cdot I - k_m \cdot \omega + u(t) \\ J \cdot \dot{\omega} &= k_m \cdot I - b \cdot \omega - \tau(t) \\ \dot{\varphi} &= \omega \end{cases} \tag{3}$$

where $L = 3\,\text{mH}$ is the armature inductance, $R = 50\,\text{m}\Omega$ is the armature resistance, $J = 1500\,\text{kg}\,\text{m}^2$ is the inertia, $b = 0.001\,\text{kg}\,\text{m}^2\,\text{s}^{-1}$ is the friction coefficient, and $k_m = 6.785\,\text{V}\,\text{s}$ is the electro-motorical force (EMF) constant of the motor. These parameter values correspond to those of a

real system. The inputs are the armature voltage, $u(t)$, and the torque load, $\tau(t)$, respectively. In the given example, $u(t)$ is 500 V, and the torque is of 2500 N m.

The cycle analysis leads to the constraints

$$\begin{aligned} I: &\quad h \leq 0.060 \\ \omega: &\quad h \leq 0.313 \\ \varphi: &\quad h \leq +\infty \end{aligned}$$

Hence, choosing an integration step $h = 0.3$ induces a partition of the system that is natural, as it clearly separates the electric components from the mechanic ones. The constraint associated with φ comes from the fact that there is a pure integral action that does not influence the cycle analysis. Figure 4 shows the simulation results.

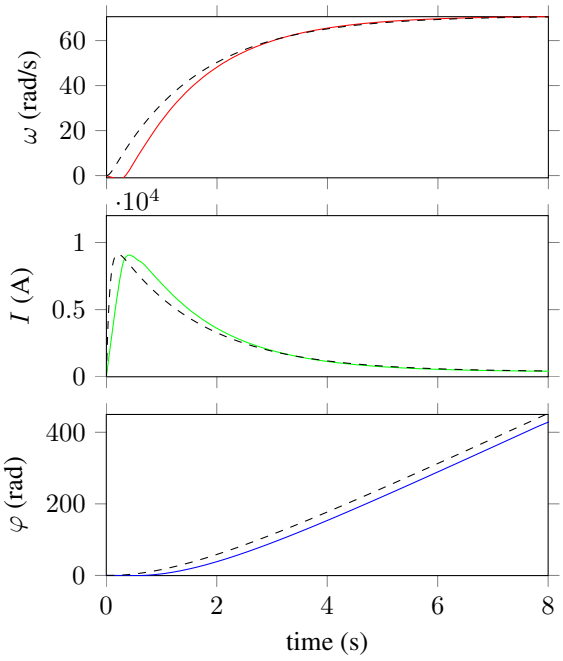


Figure 4. Simulation results of Model (3). Dashed lines represent the real trajectories, while the solid lines are the trajectories obtained with DD.

Table 1 shows the simulation statistics for different integration methods. It is worth noticing that the dimension of the system the Newton iteration has to solve is reduced from 3 to 1 in the mixed-mode method. Notice also that the EE method needs a smaller step size, hence $h = 0.05$ was chosen for numerical stability reasons. Apparently, using DD improves simulation efficiency also in this very simple case.

	Mixed-mode	**BDF**	**IE**	**EE**
# Steps	**28**	136	**28**	162
# Function ev.	**86**	157	**86**	–
# Jacobian ev.	**2**	3	**2**	–
# Fun. ev. in Jac. ev.	**4**	9	8	–
# Newton iterations	**58**	153	**58**	–
# Newton fail	0	0	0	–
Accuracy	**1.118**	–	1.213	10.043
Sim time	**0.04s**	0.05s	0.06s	**0.04s**

Table 1. Simulation statistics for Model (3).

4.2 Counterflow Heat Exchanger

This example refers to a counterflow heat exchanger with two incompressible streams (Figure 5).

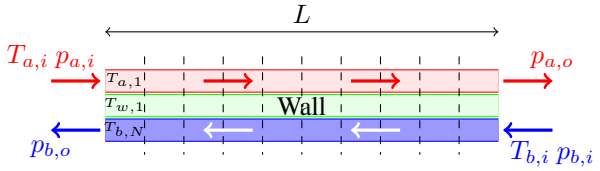


Figure 5. Counterflow heat exchanger scheme.

Both streams and the interposed wall are spatially discretised with the finite volume approach, neglecting axial diffusion in the wall – as is common practice – and also in the streams (zero-flow operation is not considered for simplicity). Taking ten volumes for both streams and the wall, with the same spatial division (again, for simplicity) leads to a nonlinear dynamic system of order 30, having as boundary conditions the four pressures at the stream inlets and outlets, and the two temperatures at the inlets. More precisely, the system is

$$\begin{cases} p_{a,i} - p_{a,o} = 2c_{f,a}L/(\rho_a \pi^2 r^5) \cdot w_a|w_a| \\ p_{b,i} - p_{b,o} = 2c_{f,b}L/(\rho_b \pi^2 r^5) \cdot w_b|w_b| \\ c_a \rho_a \pi r^2 \frac{L}{N} \dot{T}_{a,j} = w_a c_a \cdot (T_{a,j-1} - T_{a,j}) \\ \qquad\qquad + \gamma_a \left(\frac{w_a}{w_{a,nom}} \right)^{0.8} \pi r \frac{L}{N} \cdot (T_{w,j} - T_{a,j}) \\ c_w \rho_w \pi r^2 \frac{L}{N} \dot{T}_{w,j} = -\gamma_a \left(\frac{w_a}{w_{a,nom}} \right)^{0.8} \pi r \frac{L}{N} \cdot (T_{w,j} - T_{a,j}) \\ \qquad\qquad - \gamma_b \left(\frac{w_b}{w_{b,nom}} \right)^{0.8} \pi r \frac{L}{N} \cdot (T_{w,j} - T_{b,N-j+1}) \\ c_b \rho_b \pi r^2 \frac{L}{N} \dot{T}_{b,j} = w_b c_b \cdot (T_{b,j-1} - T_{b,j}) \\ \qquad\qquad + \gamma_b \left(\frac{w_b}{w_{b,nom}} \right)^{0.8} \pi r \frac{L}{N} \cdot (T_{w,N-j+1} - T_{b,j}) \end{cases} \tag{4}$$

where T stands for temperature, w for mass flowrate, p for pressure, c_f for the friction coefficient, c and ρ for (constant) specific heat and density, and γ the coefficient of heat transfer; the a, b and w subscripts denote respectively the two streams and the wall, while $j \in [0, N]$ ($j = 0$ for boundary conditions) is the volume index, counted for both streams from inlet to outlet, the wall being enumerated like stream a; the i and o subscripts, finally, stand for "inlet" and "outlet". Table 2 shows the parameter values used in the example.

Parameters					
$p_{a,i}$	1.216kPa	r	0.1m	ρ_b	3500kg/m^3
$p_{b,i}$	1.216kPa	s	0.005m	ρ_w	4200kg/m^3
$p_{a,o}$	1.013kPa	$c_{f,a}$	0.1	γ_a	100W/(m^{2}K)
$p_{b,o}$	1.013kPa	$c_{f,b}$	0.2	γ_b	100W/(m^{2}K)
$T_{a,i}$	323.15K	c_a	4200J/(kg K)	$w_{a,nom}$	0.5kg/s
$T_{b,i}$	288.15K	c_b	3500J/(kg K)	$w_{b,nom}$	0.5kg/s
N	10	c_w	3500J/(kg K)		
L	30m	ρ_a	4200kg/m^3		

Table 2. Parameter values of Model (4).

In this example, contrary to the previous one, there is no neat physical separation between the time scales of the

involved dynamics. In fact, the cycle analysis leads to the constraints

$$\begin{aligned} T_{a,j}: &\quad h \leq 10.383 \\ T_{b,j}: &\quad h \leq 13.327 \\ T_{w,j}: &\quad h \leq 13.658 \end{aligned}$$

which show that the time scales associated with the variables are quite close one to another. Due to the physical nature of this system, DD is not expected to take particular advantage of the partition.

Choosing an integration step $h = 13.0$ yields a partition that considers the $T_{a,j}$ as the fast while the $T_{b,j}$ and $T_{w,j}$ as the slow states. Figure 6 shows the simulation results — notice that the temperatures are reported with different scales.

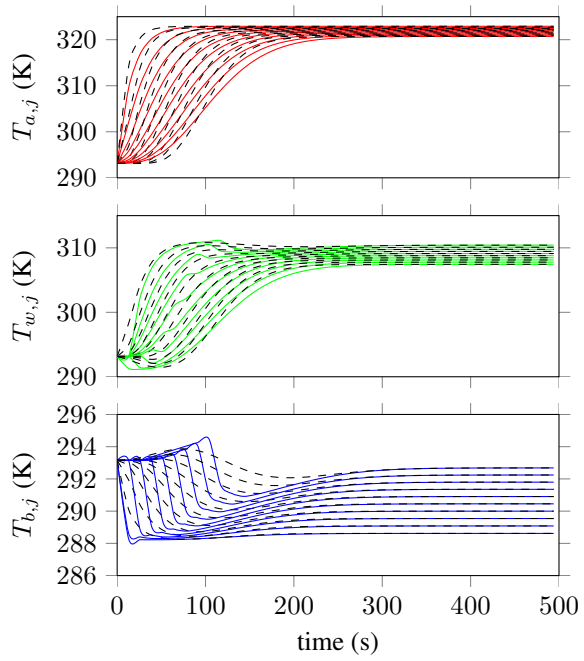


Figure 6. Simulation results of (4) with $\alpha = 1.0$. Dashed lines represent the real trajectories, while the solid lines are the trajectories obtained with DD.

Table 3 shows the simulation statistics for different integration methods. It is worth noticing that the dimension of the system is reduced from 30 to 10 in the mixed-mode method. Also, the EE method needs a smaller step size, hence $h = 10.0$ was chosen, again for numerical stability reasons.

	Mixed-mode	**BDF**	**IE**	**EE**
# Steps	**38**	212	**38**	50
# Function ev.	**114**	241	**114**	–
# Jacobian ev.	**2**	4	**2**	–
# Fun. ev. in Jac. ev.	**22**	120	62	–
# Newton iterations	**76**	237	**76**	–
Accuracy	0.017	–	**0.014**	0.059
Sim time	**0.04s**	0.15s	0.06s	0.08s

Table 3. Simulation statistics for Model (4) ($h = 13.0$).

As can be seen in (2), the integration step depends on the choice of α. Choosing a value of $\alpha = 1.0$ is usually a good choice—in fact the value of α used in the previous examples. This choice is however quite aggressive from the point of view of accuracy, even if the low frequency dynamics are caught (see, for instance, $T_{b,j}$ in Figure 6). Choosing a smaller value of α, e.g., $\alpha = 0.5$, will conversely yields a more conservative partitioning but more accurate a numerical solution. In particular, the output of the cycle analysis changes to

$$\begin{aligned} T_{a,j}: &\quad h \leq 5.191 \\ T_{b,j}: &\quad h \leq 6.664 \\ T_{w,j}: &\quad h \leq 6.829 \end{aligned}$$

and choosing $h = 6.0$ leads to the same partition as before, but producing more accurate solutions (see Figure 7). Apparently enough (see Table 4), the performance of the

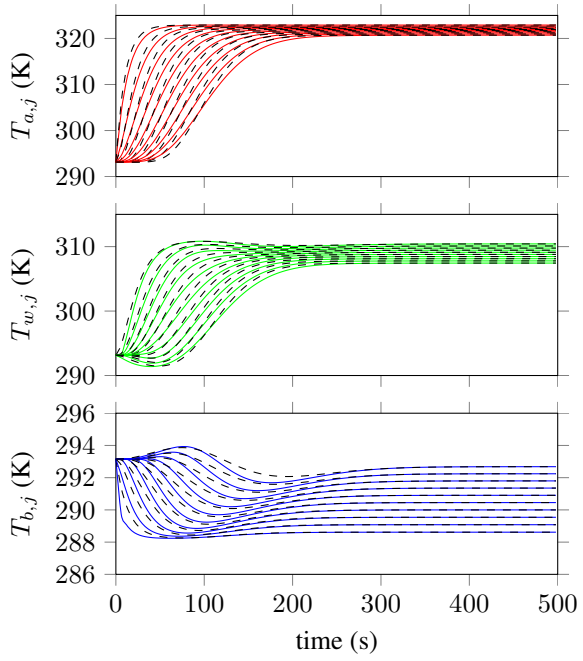


Figure 7. Simulation results of (4) with $\alpha = 0.5$. Dashed lines represent the real trajectories, while the solid lines are the trajectories obtained with DD.

mixed-mode method is still better in terms of simulation speed, and the accuracy is improved.

	Mixed-mode	**BDF**	**IE**	**EE**
# Steps	**83**	213	**83**	100
# Function ev.	**234**	243	243	–
# Jacobian ev.	**4**	**4**	**4**	–
# Fun. ev. in Jac. ev.	**44**	120	124	–
# Newton iterations	**151**	239	160	–
Accuracy	0.011	–	**0.008**	0.018
Sim time	**0.08s**	0.15s	0.16s	0.10s

Table 4. Simulation statistics for Model (4) ($h = 6.0$).

The presented examples have been kept as small as possible in order to improve results readability, but the method can be applied to larger models as well. For example, by changing in (4) the parameter N to 30, the model becomes of order 90, and the obtained simulation results are summarised in Table 5.

	Mixed-mode	BDF	IE	EE
# Steps	**125**	304	**125**	250
# Function ev.	**336**	337	345	–
# Jacobian ev.	**6**	**6**	**6**	–
# Fun. ev. in Jac. ev.	**186**	540	546	–
# Newton iterations	**211**	333	220	–
Accuracy	**0.014**	–	**0.014**	0.084
Sim time	0.21s	0.43s	0.41s	**0.20s**

Table 5. Simulation statistics for Model (4) ($h = 4.0$).

4.3 Remarks

The presented examples prove the usefulness of the approach, and show its potentialities, concluding the presentation of the DD technique. However, a last statement need motivating, i.e., the DD technique can complement existing EOOMT. To this end, we need discussing how to insert the presented technique in a manipulation toolchain of modern EOOMT (Section 5).

5. A Unifying Manipulation Toolchain

As already stated, nowadays MOR techniques do not allow to introduce approximations in the solution of DAE systems, neither acting on equations (e.g., MOR techniques) nor acting on the solution (e.g., DD). In this section we propose a complementing manipulation toolchain that bridges those concepts, without altering the classical manipulation framework, but adding some useful functionalities for the model designer (see Figure 8).

In particular, the idea is that if the analyst wants to perform some approximations in order to improve simulation speed, he/she needs to be able to specify high-level properties, e.g., upper bounds on the approximation error, and which technique must be used for it, e.g., the MOR technique as well as whether or not the use of DD is advisable.

Figure 8 depicts the proposed toolchain of model manipulations, from the EOO description to the simulation algorithm ready for code generation. The decision nodes (the diamond ones in the diagram) show where additional manipulation for simplification can be performed. If, in every decision node, the simplification is not performed, the classical manipulation toolchain comes out. Otherwise, a simpler model is produced at the end of the toolchain. The diagram also reports some coloured dashed boxes on the right side. Red boxes stand for already available methodologies that can be automatically applicable at this level of the manipulation, while green ones stand for potential methodologies which may be introduced as automatic procedures, but to date not exploited in the context of EOO modelling.

5.1 An Example Toolchain Implementation

To prove the feasibility of extending an EOO modelling toolchain as here suggested, the task was actually carried out by using JModelica[1] as the Modelica translator, exporting the model as a Functional Mockup Unit (FMU), and

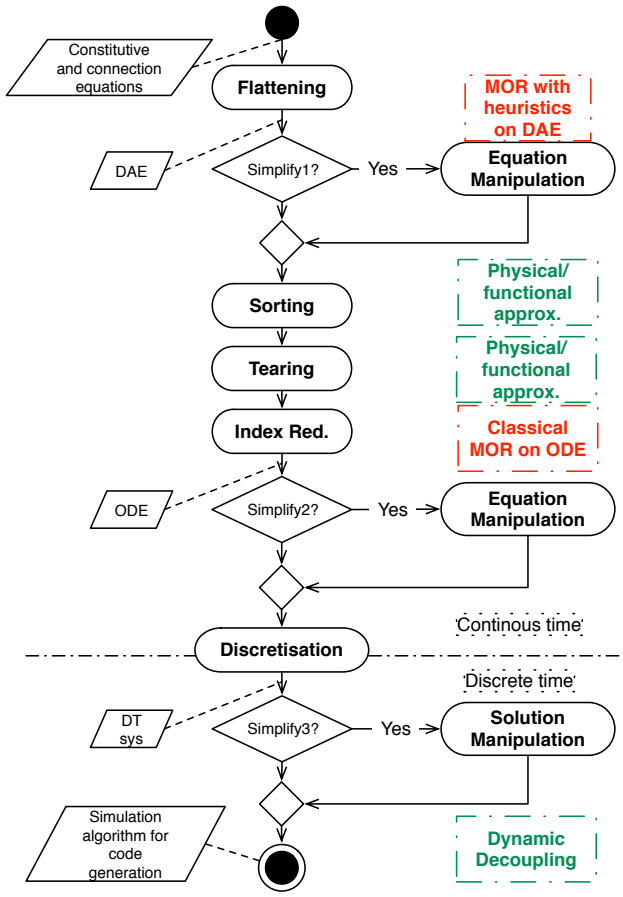


Figure 8. Activity diagram of the modified manipulation toolchain.

employing Assimulo[2] for the numerical integration, having developed the mixed-mode integrator *ad hoc*.

More in detail, the toolchain of Figure 8 was modified – for the case when "simplify" is desired – as shown in Figure 9: the output of the continuous-time part (the manipulated `model.mo`) is exported by means of the Functional Mockup Interface (FMI) to `model.fmu`, elaborated by the external python module `jd2.py` that performs the cycle analysis (i.e., takes care of the "discretisation" and the "solution manipulation" blocks); the partitioned model is then simulated with Assimulo, with the developed mixed-mode method. It is worth noticing that the integration of a new functionality (like DD) into an EOO modelling toolchain was greatly eased by adopting, for the various phases, tools that allow for some common interchange format—a feature of great importance indeed.

The developed code, including the reported examples, is available as free software, within the terms of the Modelica License v2, at the URL `http://home.dei.polimi.it/leva/jd2.html`.

6. Conclusion and Future Work

A technique was presented to allow equation-based EOOMT to take profit of DD, partitioning a complex model into "weakly coupled" submodels in a view to enhancing the

[1] `http://www.jmodelica.org`

[2] `http://www.jmodelica.org/assimulo`

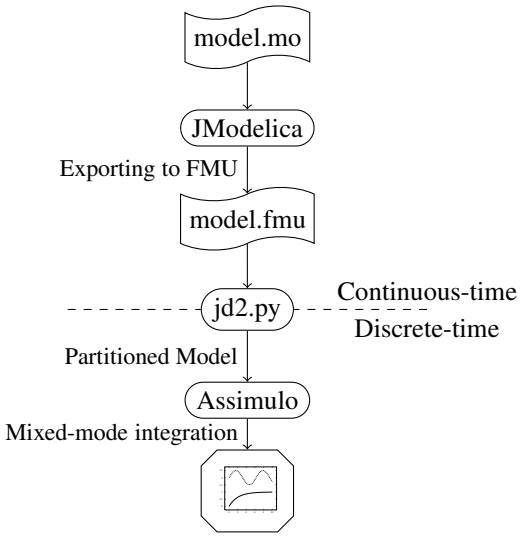


Figure 9. Integration of DD in the toolchain of Figure 8.

obtained simulation efficiency. Also, differently from some alternatives that were comparatively reviewed, the technique is naturally keen to the use of parallel simulation, or co-simulation.

The technique is based on a structural analysis of the system – called here "cycle analysis", and novel – coupled to a convenient use of mixed-mode integration. Some considerations were made on how the presented technique can be integrated in EOOMT, considering a Modelica translator as example. Simulation tests were reported to illustrate the achieved benefits, and the realised implementation was made available as free software to the community.

Future work will concern the exploitation of the mentioned keenness to co-simulation, and a tighter integration into EOOMT by developing a consistent and informative user interface. Also, the analysis will be deepened by defining convenient "separability indices" – on which some preliminary ideas are already available – to form the basis for said informative interfaces, and possibly to further automate the overall decoupling process. Finally, models of higher complexity will be addressed, so as to possibly improve the time performance of the software implementation.

Acknowledgements

The authors are grateful to Prof. F. Casella and Prof. J. Åkesson for the useful discussions and constructive criticisms, and for providing advice and cooperation to the integration of DD in the JModelica framework.

References

[1] A. Antoulas. *Approximation of large-scale dynamical systems*, volume 6 of *Advances in Design And Control*. SIAM, 2005.

[2] A. Bartolini, A. Leva, and C. Maffezzoni. A process simulation environment based on visual programming and dynamic decoupling. *Simulation*, 71(3):183–193, 1998.

[3] F. Casella, A. Leva, and C. Maffezzoni. Dynamic simulation of a condensation plate column by dynamic decoupling. In *Proc. EUROSIM '98, Espoo 1998*, pages 368–374, 1998.

[4] F. Casella and C. Maffezzoni. Exploiting weak interactions in object-oriented modeling. *Simulation News Europe*, 22:8–10, 1998.

[5] F. Cellier and E. Kofman. *Continuous system simulation*. Springer, 2006.

[6] J. Chen and S.-M. Kang. Model-order reduction of nonlinear MEMS devices through arclength-based Karhunen-Loeve decomposition. In *The 2001 IEEE Int. Symp. on Circuits and Systems*, volume 3, pages 457–460, 2001.

[7] J. Chen, S.-M. Kang, J. Zou, C. Liu, and J. Schutt-Aine. Reduced-order modeling of weakly nonlinear MEMS devices with Taylor-series expansion and Arnoldi approach. *J. of Microelectromechanical Systems*, 13(3):441– 451, 2004.

[8] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley, 2003.

[9] L. Goldberg and G. Ann. *Efficient algorithms for listing combinatorial structures*, volume 5. Cambridge Univ Pr, 2009.

[10] M. Innocent, P. Wambacq, S. Donnay, H. Tilmans, W. Sansen, and H. De Man. An analytic volterra-series-based model for a mems variable capacitor. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 22(2):124–131, 2003.

[11] S. Lall, J. Marsden, and S. Glavaski. A subspace approach to balanced truncation for model reduction of nonlinear control systems. *Int. J. of Robust and Nonlinear Control*, 12:519–535, 2002.

[12] L. Mikelsons and T. Brandt. Symbolic model reduction for interval-valued scenarios. In *ASME Conf. Proc.*, volume 49002, pages 263–272. ASME, 2009.

[13] L. Mikelsons and T. Brandt. Generation of continuously adjustable vehicle models using symbolic reduction methods. *Multibody System Dynamics*, 26:153–173, 2011.

[14] A. V. Papadopoulos, J. Åkesson, F. Casella, and A. Leva. Automatic partitioning and simulation of weakly coupled systems. Technical report, Politecnico di Milano, 2013.

[15] J. R. Phillips. Projection frameworks for model reduction of weakly nonlinear systems. In *Proc. of the 37th Annual Design Automation Conf.*, DAC '00, pages 184–189, New York, NY, USA, 2000. ACM.

[16] J. Scherpen. Balancing for nonlinear systems. *Systems & Control Letters*, 21(2):143–153, 1993.

[17] A. Schiela and H. Olsson. Mixed-mode integration for real-time simulation. In *Modelica Workshop 2000 Proc.*, pages 69–75, 2000.

[18] M. Sjölund, R. Braun, P. Fritzson, and P. Krus. Towards efficient distributed simulation in modelica using transmission line modeling. In *3rd Int. workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pages 71–80, 2010.

[19] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. on Computing*, 1(2):146–160, 1971.

[20] R. Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM J. on Computing*, 2(3):211–216, 1972.

A Strategy for Parallel Simulation of Declarative Object-Oriented Models of Generalized Physical Networks

Francesco Casella[1]

[1]Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy,
francesco.casella@polimi.it

Abstract

For several years now, most of the growth of computing power has been made possible by exploiting parallel CPUs on the same chip; unfortunately, state-of-the-art software tools for the simulation of declarative, object-oriented models still generate single-threaded simulation code, showing an increasingly disappointing performance. This paper presents a simple strategy for the efficient computation of the right-hand-side of the ordinary differential equations resulting from the causalization of object-oriented models, which is often the computational bottleneck of the executable simulation code. It is shown how this strategy can be particularly effective in the case of generalized physical networks, i.e., system models built by the connection of components storing certain quantities and of components describing the flow of such quantities between them.

Keywords Parallel simulation, Declarative modelling, Structural analysis

1. Introduction

For several years now, most of the growth of computing power predicted by Moore's law has been made possible by exploiting parallel CPUs on the same chip; this trend is likely to continue for many years in the future. Significant speed-up in the simulation of declarative, object-oriented models will require to exploit the availability of parallel processing units.

With reference to the Modelica community, there are several attempts in this direction reported in the literature, mostly from Linköping University PELAB. One possible approach (see, e.g., [2, 8, 9]) is to analyse the mutual dependencies among the equations and variables of the system by means of graph analysis, eventually providing some kind of optimal scheduling of tasks to solve them in parallel, while avoiding idle times and bottlenecks in the computation.

A completely different approach, also pioneered at PELAB, is based on Transmission Line Modelling (TLM) [11, 13]. The basic idea is that physical interactions can often be modelled by means of components that represent wave propagation in finite time (e.g. pressure waves in hydraulic systems, elastic waves in mechanical systems, electromagnetic waves in electrical systems). It is then possible to split large system models into several smaller subsystems, that only interact through TLM components. This allows to simulate each sub-system in parallel for the duration of the TLM delay, as its behaviour will only depend on the past history of connected sub-systems.

This approach is interesting because it is based on physical first principles, introducing no approximations; however, the values of transmission line delays in most systems are quite small, thus limiting the maximum length of the integration time step allowed for the simulation. Moreover, this approach critically depends on the good judgement of the modeller, that must introduce appropriate TLM components all over the system model in order to obtain a performance benefit.

In spite of the above-mentioned studies, the state-of-the-art software tools for the simulation of declarative, object-oriented models still generate single-threaded simulation code, as of today. This results in an increasingly disappointing performance, as the number of cores available on standard desktop workstations or even laptops roughly doubles every two years, while the simulation speed basically remains the same.

The goal of this paper is to show that, for a fairly large class of object-oriented models of physical systems, a simple strategy for parallel simulation can be envisioned, which is expected to provide large speed-up ratios when using many-cores CPUs, and which does not depend on the accurate estimation of the computation and communication delay to obtain good performance. Since this strategy is very easy to implement and test, it is hoped that it quickly finds its way into mainstream object-oriented simulation tools, thus improving the state of the art in this field.

The paper is organised as follows. Section 2 contains the statement of the problem and a discussion of related work. The algorithm to partition the solution of the model

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. 19 April, 2013, University of Nottingham, UK.

into independent tasks that can be executed in parallel is described in detail in Section 3. In Section 4, models of generalized physical networks are introduced, and the results of the partitioning algorithm are discussed. Section 5 briefly discusses the task scheduling problem, while Section 6 concludes the paper with final remarks and indications for future work.

2. Problem Statement and Related Work

The starting point of this analysis is an equation-based, object-oriented model of a dynamical system, e.g. written in Modelica. For the sake of conciseness, the analysis is limited to continuous-time systems, though it could be easily extended to hybrid systems with event handling and clocked variables.

After flattening, the model is transformed into a set of Differential-Algebraic Equations (DAEs)

$$F(x, \dot{x}, v, t) = 0, \tag{1}$$

where $F(\cdot)$ is a vector-valued function, x is the vector of variables appearing under derivative sign in the model, v is the vector of all other algebraic variables, and t is the time.

A commonly adopted strategy for the numerical simulation of such systems is to first transform the DAEs (1) into Ordinary Differential equations (ODEs), i.e., solving equations (1) for the state derivatives $\dot{x}$ and for the other variables v as a function of the states and of time:

$$\dot{x} = f(x, t) \tag{2}$$

$$v = g(x, t). \tag{3}$$

By defining the vector z of unknowns as

$$z = \begin{bmatrix} \dot{x} \\ v \end{bmatrix}, \tag{4}$$

the ODEs (2)-(3) can be formulated as

$$z = h(x, t) \tag{5}$$

Sophisticated numerical and symbolic manipulation techniques (see [5] for a comprehensive review) are employed to generate efficient code to compute $h(x, t)$, which will then be called by the ODE integration algorithm at each simulation step. Implicit integration algorithms will also require every now and then the computation of the Jacobian $\frac{\partial h(x,t)}{\partial x}$, which might be performed either symbolically or numerically [4, 3]. This code is then linked to standard routines for numerical integration of differential equations, such as DASSL or the routines from the Sundials suite, thus generating an executable simulation code.

In this context, it is in principle possible to exploit parallelism in the computation of $h(x, t)$, in the computation of $\frac{\partial h(x,t)}{\partial x}$, and in the algorithm for numerical integration, which might, e.g., employ parallel algorithms to solve the implicit equations required to compute the next state value at each time step. This paper focuses on those problems in which the computation of $h(x, t)$ takes the lion's share of the simulation time, e.g., because it involves the computation of cumbersome functions to evaluate the properties of a fluid in an energy conversion system model. Of course, it is also possible to combine the approach presented here with the parallel computation of the Jacobian (which is fairly trivial if done numerically) and of the numerical integration algorithms, but this is outside the scope of the present work.

The algorithm presented in the next section follows the same principle that was first put forward in [1], and later on further developed in the Modelica context in [2]: exploiting the dependencies between the different equations (and parts thereof) to determine the order in which the systems has to be solved and which systems that can be solved in parallel. However, it tries to do so in a simpler way, by exploiting the mathematical structure of generalized network models.

More specifically, [2] first represents the algorithm to compute $h(x, t)$ with the finest possible granularity: each node in the dependency graph is a single term in the right-hand-side expressions of those equations that can be solved explicitly for (x, t), or a system of implicit equations for those who can't. Subsequently, these atomic tasks are merged into larger tasks, taking into account execution and communication costs, in order to minimize the overall execution time and maximize the parallel speed-up ratio. The merging algorithms are fairly involved, and their result critically depends on those costs, which are often hard to estimate reliably. Moreover, this kind of analysis seems to fit well the computational model of networked systems (e.g., clusters of workstations, which were popular at the time of that work), where communication delays are significant when compared to execution times, making a clever merging of tasks mandatory for good performance. Results obtained by the application of these techniques to a few representative test models were reported in [2] and subsequent related work, but no analysis was ever attempted to understand what is the typical structure of the dependencies in different classes of physical system models, in order to understand how much they can benefit in general from the application of this parallelization technique. Unfortunately, even though these algorithm were implemented in earlier versions of the OpenModelica compilers, they are currently no longer supported, which prevents trying them on real-life problems that can only be handled by more recent versions of the compiler.

The aim and scope of this paper are somewhat different. First of all, the underlying computational model is that of multiple-core CPUs with shared memory, in which communication delays tend to be small or negligible compared to execution times, at least as long as all the variables of the model can be kept within the on-chip cache memory at all times. This seems a feasible proposition for models of moderately large size: a system with 10000 variables (after optimizations such as alias elimination) in double precision requires only 80 kilobytes of shared cache memory. Second, it is shown that a fairly large class of object-oriented models, namely generalized physical networks, has a dependency structure that can be very well exploited by a sim-

ple parallelization algorithm which does not critically depend on the accurate estimation of execution times, but can guarantee nearly optimal allocation of parallel resources, as long as the number of nodes in the network is much larger than the number of parallel processing units.

3. An Algorithm for Parallel Solution of Equations from Declarative Models

The proposed algorithm is now outlined in detail.

1. Build an Equations-Variables (E-V) digraph, where every E-node corresponds to an equation in (1), every V-node correspond to a scalar unknown variable in z, and an edge exists between an E-node and a V-node if the unknown variable shows up in the equation.
2. Find a complete matching between E and V nodes (see [7] for a review of suitable algorithms); if this is not possible because the DAE system has index greater than one, apply Pantelides' algorithm [12] and the Dummy Derivative algorithm [10] until the system is reduced to index 1 and a complete matching can be found.
3. Transform the E-V digraph into a directed graph by first replacing each non-matching edge with an arc going from the E-node to the V-node, then by collapsing each V-node with its matching E-node.
4. Run Tarjan's algorithm [6] on the directed graph to locate its strongly connected components, corresponding to systems of algebraic equations that need to be solved simultaneously for their matching unknown variables.
5. Collapse each set of nodes making up a strong components into one macro-node.
6. Let $i = 1$
7. Search for all the sinks in the graph and collect them in the set S_i; these correspond to equations (or to systems of implicit equations) that can be solved independently of each other.
8. Delete all nodes in S_i from the directed graph, as well as all arcs connected to them.
9. If there are still nodes in the graph, increase i by one and goto Step 7.

When the algorithm shown above terminates, all the equations and systems of implicit equations of the system will be collected in the sets S_i.

Proof: after executing Step 5, the directed graph has no closed cycles left in it, because each and every strong components has been collapsed into a single macro-node; therefore, there exists at least one sink in the graph. Removing nodes without outgoing arcs does not create cycles, so that at each iteration at least one node is removed from the graph, until there will be none left, QED.

Note that state-of-the-art Modelica tools already perform Steps 1 to 4, so that the addition to the tool code in order to implement the proposed strategy is minimal.

The result of this analysis can also be visualized in terms of the Block Lower Triangular (BLT) representation of the incidence matrix, see Figure 1. Each set S_i corresponds to a block diagonal square matrix in the BLT matrix, marked in red in Figure 1, where every block on the diagonal corresponds to a strong component of the system of equations. As the ordering induced by the directed graph is partial, there exist many different BLT transformations of the original system corresponding to the same graph.

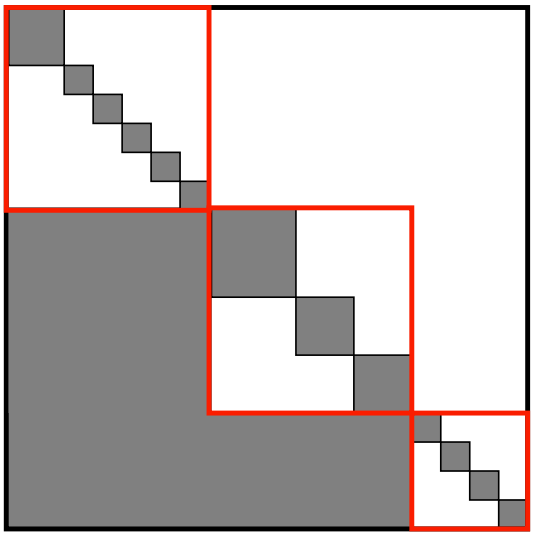


Figure 1. The incidence matrix in BLT form with S_i sets.

All the equations or systems of equations showing up in each set S_i can now be solved independently on parallel cores. Before moving to the solution of set S_{i+1}, it is necessary to wait that all equations belonging to the set S_i have been solved.

The latter requirement can in general create bottlenecks, e.g., $N-1$ cores might stand idle for a long time, waiting for the N^{th} one to complete its task. However, if the number of nodes is much larger than the number of cores and there is no single node whose execution time is disproportionately longer than that of all the others, on average the impact of such situations on the overall execution time will be small. It will be shown in the next Section that this is precisely the case of large generalized physical networks.

4. Application to Generalized Network Models

Many physical models can be built by connecting storage components and flow components, see Figures 2-3. The former ones describe the storage of certain quantities, by means of dynamic balance equations; the latter instead describe the flow of those quantities between different components, which is governed by the difference of some potential variable at the two boundaries. Two examples will be detailed in this section: thermal networks and thermo-hydraulic networks.

4.1 Thermal Networks

Thermal networks describe the flow of heat between bodies having different temperature. In this case the stored quantity is thermal energy, which is conveniently described by temperature state variables, while the flow components

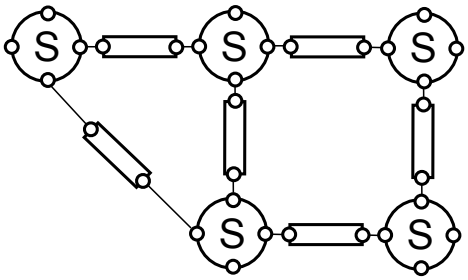


Figure 2. A physical network model with flow components connected to storage components only.

compute the thermal power flow based on the temperature at the two boundaries.

Thermal storage components are described by energy balance equations:

$$C(T_i)\frac{dT_i}{dt} = \sum_j Q_{i,j}, \tag{6}$$

where T_i is the temperature of the i-th storage component, $C(T)$ is the thermal capacitance, and $Q_{i,j}$ are the heat flows entering the i-th component. For simplicity, assume all thermal power flows can be modelled by constant thermal conductances:

$$Q_i = G_i(T_{i,a} - T_{i,b}) \tag{7}$$

where G_i is the thermal conductance of the i-th flow component and $T_{i,a}, T_{i,b}$ are the two boundary temperatures.

Assuming that each thermal flow component is directly connected to two storage components, as in Figure 2, once all alias variables have been eliminated, the equations (6) will be matched to their corresponding temperature derivatives, while the equations (7) will be matched to their corresponding heat flows. There will be no strong components in the E-V graph, corresponding to a strictly lower triangular BLT form of the incidence matrix. If the algorithm presented in Section 3 is now applied, the set S_1 will contain all the flow equations (7), each of which can be solved independently, and the set S_2 will contain all the storage equations (6), each of which can be solved independently once the heat flows have been computed by the equations in S_1.

With reference to the thermal network in Fig. 2, the directed graphs at each algorithm iteration are shown in Fig.

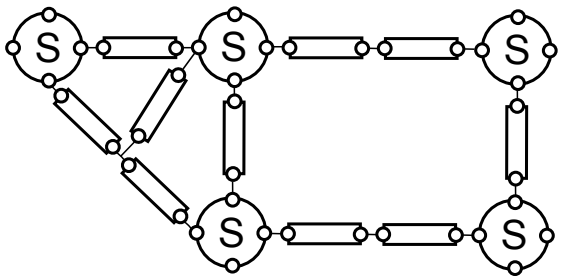


Figure 3. A physical network model with flow components directly connected to each other.

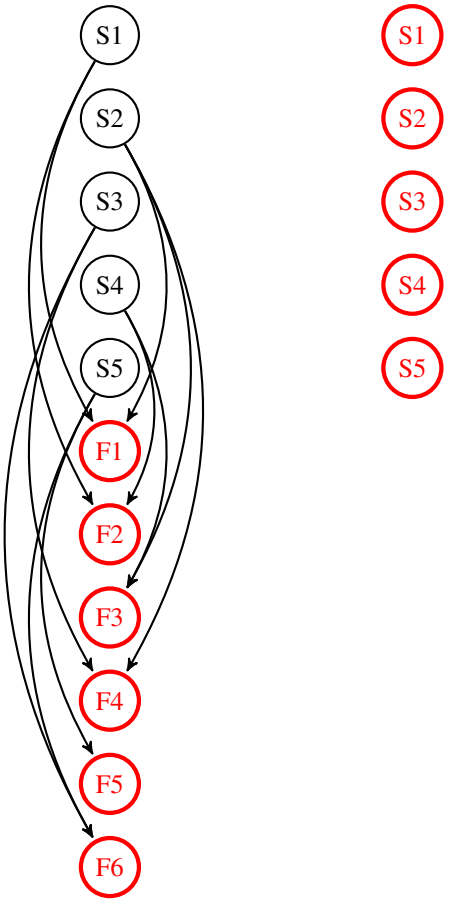


Figure 4. Iterations of the parallelization algorithm: thermal network

4. Nodes marked with the letter S represent storage equations (6), nodes marked with the letter F represent thermal flow equations (7); thick-bordered red nodes correspond to the set S_i at the i-th iteration.

In case of more complex connection topologies, where the heat flow components are directly connected to other heat flow components as in Figure 3, there will be strong components in the E-V graph, corresponding to sets of algebraic equations that must be solved simultaneously to determine the heat flows and the intermediate temperatures, which in this case are not known state variables. Consequently, the set S_1 will also contain the corresponding macro-nodes.

4.2 Thermo-Hydraulic Systems

Thermo-hydraulic networks describe the flow of mass and thermal energy between different components representing the storage of mass and thermal energy in finite volumes of the system, by means of flow components describing the mass flow and the heat flow (e.g., due to convective heat transfer) between different volumes. In this case, the stored quantities are mass and energy, which can be described, e.g., by pressure and temperature state variables. Mass flow rates are determined by the pressure difference between the boundaries of flow components, and also by the upstream

properties of the fluid (e.g., the density). Heat flows are determined by thermal conductances, as in the previous sub-section.

Storage components are described by mass and energy balance equations:

$$\begin{bmatrix} e_i & h_i & \rho_i & \frac{\partial \rho_i}{\partial p} & \frac{\partial \rho_i}{\partial T} & \frac{\partial e_i}{\partial p} & \frac{\partial e_i}{\partial T} \end{bmatrix} = f(p_i, T_i) \quad (8)$$

$$M_i = \rho_i V_i \quad (9)$$

$$\frac{dM_i}{dt} = \sum_j w_{i,j} \quad (10)$$

$$\frac{dE_i}{dt} = \sum_j w_{i,j} h_{i,j} + \sum_j Q_{i,j} \quad (11)$$

$$\frac{dM_i}{dt} = \frac{\partial \rho_i}{\partial p}\frac{dp_i}{dt} + \frac{\partial \rho_i}{\partial T}\frac{dT_i}{dt} \quad (12)$$

$$\frac{dE_i}{dt} = \left(\frac{\partial e_i}{\partial p}\frac{dp_i}{dt} + \frac{\partial e_i}{\partial T}\frac{dT_i}{dt}\right) M_i + e_i \frac{dM_i}{dt} \quad (13)$$

where $e_i, h_i, \rho_i, p_i, T_i$ are the specific internal energy, specific enthalpy, density, pressure, and temperature of the fluid contained in the i-th component, V_i is the volume of the component, M_i is the mass of the fluid contained in the component, $w_{i,j}$ are the mass flow rates entering the component, $h_{i,j}$ the associated upstream specific enthalpies, and $Q_{i,j}$ the heat flows entering the component.

Mass flow components determine the mass flow rate as a function of the boundary pressures and of the upstream density:

$$w_i = w(p_{i,a}, p_{i,b}, \rho_i), \quad (14)$$

while heat flows components are the same as in the previous sub-section.

Assuming that mass flow and heat flow components are always connected between two storage components, once all alias variables have been eliminated, equations (8) will be matched to all the properties on their left-hand-side, equation (9) will be matched to M_i, equation (10) will be matched to $\frac{dM_i}{dt}$, equation (11) will be matched to $\frac{dE_i}{dt}$, the pairs of equations (12)-(13) will be matched to dp_i/dt and dT_i/dt, forming a strong component of two variables and equations for each i, equations (14) will be matched to the mass flow rates w_i, and equations (7) will be matched to the heat flows Q_i.

After the algorithm illustrated in Section 3 has been applied, the set S_1 will contain all the fluid property equations (8) and the thermal flow equations (7), each of which can be computed independently. The set S_2 will contain the equations (9), (14), which can be solved independently. The set S_3 will contain the equations (10) and (11), each of which can be solved independently. Finally, the set S_4 will contain the macro-nodes corresponding to the systems (12)-(13), each of which can be solved independently to compute the state derivatives.

With reference to a simple system composed of three storage components, connected in series by two mass flow components, with the flow direction from the first to the last storage component, the directed graphs shown in Fig. 5 are obtained at each iteration. Equations (8) are marked with P, (9) with M, (14) with F, (10) with MB, (11) with EB, (12)-(13) with D. As before, thick-bordered red nodes correspond to the set S_i at the i-th iteration.

In case there are series-connected heat flow or mass flow components in the system, as in Figure 3, their variables and equations will form strong components in the E-V graph, which will end up in sets S_1 (heat flows) and S_2 (mass flows), respectively.

4.3 Outlook

First of all, it is worth noting how the number of sets of equations S_i, that need to be solved in sequence, remains very low (2 for thermal networks and 4 for thermo-hydraulic networks), regardless of the size of the system. Therefore, as the number of components increases, the possibility of exploiting a large number of parallel CPU also increases, since there will be an increasing number of tasks in each S_i that can be performed in parallel before synchronizing for the transition to the next set S_{i+1}. For instance, if there are 1000 storage components in a thermo-hydraulic network, it is possible to distribute the computation of the corresponding fluid properties over up to 1000 parallel cores.

It is also worth noting that in the case of thermo-hydraulic systems such as steam power plant models, the computation of the fluid properties in each storage component, equation (8), often takes up the lion's share (90% or more) of the CPU time required to solve the DAEs for the state derivatives, and also a large share of the total CPU time required for the entire system simulation, as the time required to compute $h(x,t)$ dominates the time spent by the integration algorithm to find the value of the next state vector. A simple strategy as the one proposed here will be thus very effective in this case, since those computations will all end up in set S_1, and thus will be performed in parallel on all the available CPU cores. In these cases, a speed-up ratio close to the number of cores can be expected.

5. Scheduling Policies

The parallel tasks determined by the algorithm discussed in the previous section need to be run several times at each simulation time step, depending on the chosen integration algorithm, so they will be run hundreds or thousands of times in a typical simulation run. A trivial scheduling policy for the parallel solution of the system equations is to first set up a thread for each (macro) nodes in the graph; subsequently, for every required computation of $h(x,t)$, the threads corresponding to the set S_1 are activated, so they run on the first available core until there are no more threads running, then those corresponding to set S_2, and so on and so forth until S_N is completed. All threads read and write from and to a shared memory; since every node only computes the variables it is matched to, and only reads variables that were computed in previous parallel sequences, it is guaranteed that read/write conflicts cannot take place, thus avoiding the need of mechanisms such as semaphores.

In many cases (e.g., when cumbersome fluid properties computations are involved, as noted in the previous section), such a simple policy could already be highly advan-

Figure 5. Iterations of the parallelization algorithm: thermo-hydraulic network

tageous, compared to a purely sequential solution of the DAEs. However, there are two major potential problems that could arise.

The first problem is the impact of the overhead required to activate a thread for each (macro) node in the equations directed graph. Consider for example the case described in Section 4.1: every instance of equation (7), which only requires a subtraction and a multiplication to be solved, will end up in a separate thread. If the thread activation time is comparable or higher than the time required for the two floating point operations, then the end result of this parallelization strategy could be a code that actually runs slower than its sequential counterpart. This problem can be solved by roughly estimating the order of magnitude of the execution time associated to each (macro) node, and then aggregate many of them until the thread set-up time is negligible compared to the total execution time.

The second problem is how to guarantee that all cores are used as much as possible, and none stays idle for a long time. If a few tasks in S_i take a much longer time than all the other ones, and there is a large number of parallel cores available, activating them as the last ones might result in a waste of time, because most of the cores will eventually stay idle, waiting for those longer tasks to end. A possible solution to this problem is to estimate the execution time of each task, then start the longer-running ones first. Again, a very rough estimate of the order of magnitude of the execution time is enough for this purpose. If the number of tasks in S_i, which corresponds to the number of nodes in the physical networks, is much larger than the number of processing units, the impact of the idle time spent at the end of each parallel section of the algorithm will on average be small, compared to the total execution time. This is the case, for example, if a 16-cores CPU is used to simulate a network of a few hundred nodes (e.g., a thermal power plant model).

6. Conclusions and Future Work

In this paper, a simple algorithm has been presented that allows to distribute over parallel CPU cores the solution of DAEs stemming from object-oriented models. It has been shown how this algorithm can be very effective in partitioning the solution of the system DAEs over many parallel CPU cores, when applied to large models of thermal and thermo-hydraulic networks, which can easily involve hundreds or thousands of storage and flow models.

In the near future, it is planned to implement the algorithm in the OpenModelica compiler, which already offers support for parallel simulation of systems having a decoupled structure (e.g., thanks to the TLM methodology), using the OpenMP framework. This will allow to experiment the proposed strategy on generalized physical network models, but also on different kinds of models, such as mechanical systems. Another interesting perspective could be to couple the strategy presented here for parallel solving of DAEs with parallel ODE solvers for sparse systems, in order to fully exploit parallelism in all the tasks required to simulate an object-oriented declarative model.

References

[1] Niclas Andersson and Peter Fritzson. Generating parallel code from object oriented mathematical models. In *Proceedings 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, USA, Jul 19–21 1995.

[2] P. Aronsson. *Automatic Parallelization of Equation-Based Simulation Programs*. PhD thesis, Linköping University, Department of Computer and Information Science, 2006.

[3] Willi Braun, Stephanie Gallardo Yances, Kilian Link, and Bernhard Bachmann. Fast simulation of fluid models with colored jacobians. In *Proceedings of the 9th International Modelica Conference*, pages 247–252, Munich, Germany, Sep. 3–5 2012. Modelica Association.

[4] Willi Braun, Lennart Ochel, and Bernhard Bachmann. Symbolically derived jacobians using automatic differentiation - Enhancement of the OpenModelica compiler. In *Proceedings 8th International Modelica Conference*, pages 495–501, Dresden, Germany, Mar 20-22 2010. Modelica Association.

[5] F. E. Cellier and E. Kofman. *Continuous System Simulation*. Springer-Verlag, 2006.

[6] I. S. Duff and J. K. Reid. An implementation of Tarjan's algorithm for the block triangularization of a matrix. *ACM Transactions on Mathematical Software*, 4(2):137–147, 1978.

[7] Jens Frenkel, Gunter Künze, and Peter Fritzson. Survey of appropriate matching algorithms for large scale systems of differential algebraic equations. In *Proceedings 9th International Modelica Conference*, pages 433–442, Munich, Germany, Sep. 2012. Modelica Association.

[8] H. Lundvall. Automatic parallelization using pipelining for equation-based simulation languages, 2008. Lic. Thesis.

[9] H. Lundvall, K. Stavåker, P. Fritzson, and C. Kessler. Automatic parallelization of simulation code for equation-based models with software pipelining and measurements on three platforms. *Computer architecture news, Special issue MCC08 - Multicore computing 2008*, 36(5), 2008.

[10] S. E. Mattsson and G. Söderlind. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM Journal on Scientific Computing*, 14(3):677–692, 1993.

[11] Kaj Nyström and Peter Fritzson. Parallel simulation with transmission lines in Modelica. In *Proceedings 5th Modelica Conference*, pages 325–331, Vienna, Austria, Sep 6–8 2006. The Modelica Association.

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

[13] Martin Sjölund, Robert Braun, Peter Fritzson, and Petter Krus. Towards efficient distributed simulation in modelica using transmission line modeling. In *Proceedings 3rd International Workshop on Equation-Based Object-Oriented Languages and Tools*, pages 71–77, Oslo, Norway, Oct 3 2010.

Session III: Diagnosis and Debugging

Functional Debugging of Equation-based Languages

Arquimedes Canedo[1] Ling Shen[1]

[1]Siemens Corporation, Corporate Technology, Princeton, USA,
{arquimedes.canedo, ling.shen}@siemens.com

Abstract

State-of-the-art debugging techniques for equation-based languages follow a low-level approach to interface users with the complex interactions between equations and algorithms that describe cyber-physical processes. Although these techniques are useful for understanding the low-level behaviors, they do not provide the means for creating a system-level understanding that is often necessary during the early concept product design phase. In this paper, we present a novel debugging technique for equation-based languages based on a high-level approach to facilitate the system-level understanding of complex cyber-physical processes. Our debugging interface is based on functional models that describe what the system does in a formal language that uses natural language elements to improve inter-disciplinary communication. Our novel technique, referred to as functional debugging, can be used in the context of the current systems engineering industrial practice in order to identify system-level problems and explore design alternatives during the early concept design phase. We present a working implementation of our functional debugger and we discuss the benefits of our approach using an automotive use-case.

Keywords Functional modeling, debuggers, equation-based languages, simulation, cyber-physical systems, concept design

1. Introduction

Product development, from consumer products to military systems, is a highly competitive area where companies are constantly challenged to meet quality targets, revenue targets, and launch dates for new and innovative products [2]. In order to reduce the product development cycle, companies use systems engineering methodologies that attempt to parallelize and detect errors in the design as early as possible. For example, DARPA's META-II project [59] has the qualitative goal to compress the system design, development, test, and evaluation of mission critical design applications by a factor of 5x or more by identifying system-level and component interaction problems early in the design cycle. Currently, most computer-based design tools are suitable for detail design and it is very difficult or impossible to effectively front-load the detection of system-level design flaws [10, 17].

Products characterized by a blend of multiple disciplines including mechanical, electrical, thermal, software, and control are often referred to as cyber-physical systems (CPS). CPS are often characterized by the use of dynamic architectures (e.g. based upon the availability of elements such as sensors) that produce online, emergent, and on-the-fly unprecedented behavior. Therefore, CPS design, analysis, validation necessitates a new systems science that encompasses both physical and computational aspects [1]. Object oriented equation-based languages are often used to describe CPS because they can be used to model the behavior of both continuous (physical-) and discrete (cyber-) processes. To facilitate physical modeling in terms of energy conservation principles, these languages are implemented as *declarative* programming languages that describe *what* the goal is. Debugging these programs is very challenging because during execution or simulation, these programs are highly optimized and transformed [44] into *imperative* programs that instruct the computer *how* to reach the goal. Unfortunately, these debugging techniques expose the user with the low-level details of the model and therefore, it is difficult to incorporate these techniques in tools for the early concept design phase.

In this paper, we introduce a new debugging technique suitable for the concept design phase. Based on the observation that functional models describe *what* the system is supposed to do, and models in equation-based languages describe *what* the cyber-physical process is, we provide a *functional debugging* interface that helps users understand complex processes in a high-level of abstraction. Our implementation couples a functional model (functionality) with an underlying simulation model (behavior). This enables, for the first time, a dynamic functional representation of the system that serves as a quick validation tool for new design concepts. The functional debugging technique can be integrated into the systems engineering process by reusing functional and simulation components and allowing the identification of system-level problems early in the design. Specifically, the novel contributions of this paper are:

- A model-based debugging methodology, referred to as *functional debugging*, that interprets the results of simulation models written in equation-based languages in a high-level manner and allows the identification of system-level errors and integration problems early in the design cycle.

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. 19 April, 2013, University of Nottingham, UK.
 Proceedings available at:
http://www.ep.liu.se/ecp_home/index.en.aspx?issue=084
EOOLT 2013 website:
http://www.eoolt.org/2013/

- The observation that declarative equation-based languages fundamentally describe *what* the system does and therefore can be naturally mapped to functional models that also describe *what* the system is supposed to do but in a higher-level of abstraction that is suitable for communication and design space exploration of new concepts.
- An implementation of the functional debugging methodology that, for the first time, provides a *dynamic* or *executable* functional model that effectively combines functionality and behavior in the same model.

The rest of the paper is organized as follows. Section 2 puts our work into context with an overview of the state-of-the-art in equation-based languages and their debugging techniques, and functional modeling. Section 3 introduces our new functional debugging approach and provides the details of our implementation. Section 4 presents how the functional debugger can be integrated into a systems engineering process with an automotive use-case. Section 5 summarizes our findings and provides the outlook for future work.

2. Background and Related Work

2.1 Physical Modeling with Equation-based Languages

In recent years, companies from all sectors are designing complex products through *physical modeling* – the combination of components that correspond to physical objects in the real world (e.g. pipes, motors, resistors, software). This approach is very attractive because reusable components encapsulate an associated behavioral description according to the laws of physics and principles of energy conservation. The interconnection of components in a model creates complete mathematical models that effectively combine different disciplines. Thus, by focusing the design on the structure of the system and automatically finding the equations that describe its behavior, physical modeling eliminates the need for manually finding mathematical descriptions of systems [51]. Equation-based languages such as Bond Graphs [12], Modelica [33], Simscape [28] have been developed to provide the syntax and semantics for physical modeling. Most equation-based languages are declarative programming languages that describe *what* the program should accomplish. It is the responsibility of the compilers and optimizers to transform equation-based declarative programs into an imperative program that specifies *how* to accomplish the goal as most numerical solvers require an imperative program to simulate the dynamic behavior of the system. Due to the extensive transformations that a declarative program suffers when converted into its imperative equivalent, *what the user sees (equations in the declarative model) is NOT what the user gets (code in the imperative simulation)*, and therefore it is very challenging to debug these applications.

2.2 State-of-the-art Debugging Techniques for Equation-based Languages

Debugging equation-based languages is a challenging problem that requires a combination of classical debugging techniques and other special techniques. In [44], the authors provide a comprehensive survey of the state-of-the-art in techniques for debugging declarative equation-based languages typically used in physical modeling. These debugging techniques are categorized as static (compile-time) and dynamic (run-time). Static techniques focus on tracing the complex process of symbolically transforming declarative code into highly optimized imperative code to provide explanations regarding problematic code. Novel and innovative static debugging techniques using graph-theoretic methods have been developed [9]. Dynamic techniques, on the other hand, are similar to classical debugging and focus on interactively inspecting the imperative parts of the model that relate to functions and algorithms typically used to describe control code and embedded software. Hybrid approaches [44] that combine static and dynamic methods are the most advanced debugging techniques for equation-based languages.

Although these techniques are invaluable for identifying errors in models and code during the detail design phase, they must focus on the low-level aspects of modeling and simulation. Integrating these debugging techniques to the first iterations of the systems engineering processes is difficult because a high-level of abstraction, rather than a low-level, is preferred during the early concept design phase of modern cyber-physical systems [25]. In this paper, we present a debugging technique that deals with the functional aspects of equation-based languages and presents to the user a high-level interface to complex cyber-physical processes to facilitate the conceptual design space exploration of complex products. In the following Sections we discuss how our high-level debugging approach and state-of-the-art low-level debugging techniques are complementary in a systems engineering context.

2.3 Functional Modeling

Functional modeling is a systems engineering activity where products are described in terms of their functionalities and the functionalities of their subsystems. Fundamentally, a functional model reflects *what* the system does and, therefore, we observe that functional models are strongly related to declarative equation-based languages. Because a Functional Model decouples the design intentions (functions) from behavior and/or structure (logical components[1]), it can be used as the basis for communication among engineers of different disciplines. Functional modeling reflects the design intentions that are typically driven by the product requirements and the human creativity.

Functional modeling is acknowledged by many researchers and practitioners to be a subjective process [17], therefore suitable for concept design. Defining a system in terms of its functionality[2] may seem simplistic and unnecessary but this is exactly what improves the systems engineering process by consolidating multiple engineering paradigms (e.g. electrical, mechanical, software, thermal engineering) into a unified system representation. By *making explicit the implicit knowledge of the engineers*, a functional model exposes the obvious facts about the system that people can easily understand, regardless of their domain of expertise. This improves the communi-

[1] Logical components (and models) are often used as the guidelines for the creation of simulation models.

[2] Functionality of a system is defined as its purpose, intent, or goal.

cation among different disciplines because it brings the minds of the domain experts and designers to a system-level abstraction that is facilitated by natural language. In this paper, we introduce a novel high-level debugging technique suitable for early concept design phases that uses functional modeling as a debugging interface for equation-based languages. Compared to existing research on functional modeling [17, 10, 46, 8, 25, 61], we are the first to demonstrate the use of functional models for debugging simulation models.

3. Functional Debugging

In this paper, we define *functional debugging of equation-based languages* as the mechanism by which states and variables of a running simulation are visualized through a functional model to create an implementation independent understanding of a cyber-physical process. As shown in Figure 1, a functional debugger relies on three components : a functional editor, a simulation model synthesizer, and a simulation runtime[3]. The functional editor is a visual programming environment for users to author functional models that describe *what the system does*. The functional editor is also used as the debugger user interface that allows users to visualize and interact with the simulation in a high-level of abstraction. Our implementation uses Microsoft Visio as the functional editor. The simulation model synthesizer is a computer program (automatic) or a simulation expert (manual) that takes a functional model as an input and generates a corresponding simulation model that *realizes* or *embodies* the system's functionality. This simulation model provides the executable semantics to the functional model. In addition to the simulation model, the synthesizer also generates a mapping model that associates functions to simulation components. Finally, the simulation runtime simulates the simulation model and calculates the dynamic behavior of the system. It is important to note that different simulation runtimes may be used to simulate the same functional model. For example, the thermal-vibration facet of a functional model may be simulated using a finite element analysis solver, and its 1D electro-mechanical facet may be simulated using Modelica or Simscape.

The functional debugger takes a functional model, a simulation model, and a mapping model as inputs. The mapping model specifies how functions and flows in the functional model associate to simulation components and effort/flow variables in the simulation model. This information is used during debugging (dotted lines in Figure 1) to relate the simulation output to visualization in the functional model, and to relate user interaction debugging commands to the running simulation. For interactive debugging, the functional debugger should be capable of controlling a simulation through pausing, stopping, resuming, advancing time to the next integration step, and querying simulation variables. The rest of this Section describes our implementation of the functional debugger architecture.

3.1 Functional Editor

Visual programming languages are suitable for authoring functional models [56, 21, 42, 48, 17, 66, 23] because a

[3] In this paper, we use *simulation runtime* and *simulation engine* interchangeably.

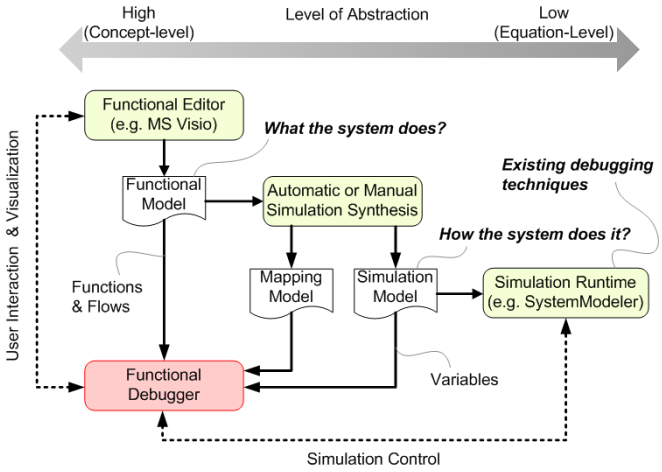


Figure 1. Functional Debugging Architecture consists of three main components: a functional editor, a synthesizer, and a simulation runtime. Different models are necessary for the functional debugger to relate functions to behavior.

Table 1. Functional modeling shapes in Visio stencil.

Visio Shape	Syntax
Function Block	Function
Material Flow	Material
Energy Flow	Energy
Signal Flow	Signal

diagrammatic representation facilitates the understanding of the system as a collection of functionalities interacting through the exchange of material, energy, and signals. Although a functional model can be also expressed textually, or as a design matrix [31], we believe that a visual functional editor improves the productivity of designers and our implementation provides an editor based on Microsoft Visio ActiveX control that can be easily embedded in other systems engineering tools. We have extended Visio with a C# implementation to improve the user-interaction and to manage the communication and data transfer between the displayed interface and the simulation runtime.

The functional modeling types are provided as shapes in a Visio stencil as shown in Table 1. We use the de-facto functional modeling syntax consisting of a block-flow diagram where blocks represent functions (process) that transform inputs into outputs (flows) [21, 42]. Blocks and flows use the Functional Basis syntax [56] to categorize functions into 8 categories and a total of 32 primitive functions, and flows into 3 categories (*material*, *energy*, and *signal* and a total of 18 flow subtypes. Constraining the vocabulary for functional modeling is beneficial for the systems engineering process because it normalizes the understanding and consistency of the models across the computer-aided tools and the organization. Although functional modeling is a highly subjective and creative process [17], the use of a constrained vocabulary does not affect the expressiveness of the functional models.

In the functional editor, a functional model can be refined into more specific descriptions in a process referred to as *functional decomposition*. For example, in the functional model of an automobile shown in Figure 2, the "`transport people`" function can be decomposed into sub-functions such as "`Store Chemical Energy`" and "`Convert Chemical Energy to Rotational Mechanical Energy`" implying the design of an internal combustion engine car. Furthermore, sub-functions can be decomposed to create a *functional decomposition tree* where the root node represents the top-level function and the leaf-nodes represent elementary functions such as "`Transfer Translational Mechanical Energy (TME)`".

Although our implementation uses the Functional Basis vocabulary, we use different semantics and we have added additional function types to facilitate the modeling of modern cyber-physical systems. For example, the original Functional Basis specifies that functional models are executed from left-to-right [56]. This causality rule, unfortunately, prohibits the coupling of functional models to acausal equation-based simulation languages because a change in the direction of energy flow during simulation is not expressible in the original Functional Basis semantics. Moreover, this causality rule does not allow for *feedback loops*, an essential construct for control theory modeling. To overcome these limitations, our functional editor allows acausal (left-to-right and right-to-left) execution semantics and the creation of feedback loops anywhere in the functional model as shown in the Third-level Functions in Figure 2. Moreover, we provide additional elementary functions for "`Control`" (function in black) and "`Sense`" (function in gray) to model cyber-physical control systems. Note that the functional modeling flows are represented by a directed arrow in Table 1. This is simply the syntax of the Functional Basis [56] and during debugging, the simulation semantics will affect the look and feel of these flows and functions. In other words, although the static functional model is constructed with directed flows, the dynamic functional model implies and reflects energy and material transfers in both directions and this also affects the functions' signatures.

3.2 Simulation Model Synthesis

The goal of the simulation model synthesis is to find components that fulfill the functionalities in a functional model. The synthesis can be performed manually by a simulation expert, or automatically by a synthesis tool. Automatic synthesis of functional models to simulation models is challenging because one function may be realized by multiple and different components, and one component may realize multiple functions. In other words, multiple valid simulation models exist for a given functional model, but only a few are useful for modeling the actual system. In our previous work [11], we introduced a *context-sensitive synthesis algorithm* that reliably generates high-quality simulation models from functional models. The synthesizer puts every function within a functional model into a context provided by its input and output flows, and using *engineering rules*[4] it correctly maps functions to the specified simulation components from reusable component libraries. Engineering rules and simulation component libraries are the means for capturing *engineering knowledge*. Due to the easy access to various simulation component libraries [34, 26, 36], our synthesizer currently generates Modelica code as an output. However, the synthesizer can be easily modified to emit and reuse components from other equation-based languages.

A simulation model consists of components with well defined interfaces, and each component may contain equations, variables, and algorithms. In order to create a correct mapping from functions to simulation components, the functional debugger must associate functions and flows in a functional model with components and variables in a simulation model. In the case of automatic synthesis, the output of an engineering rule is the mapping of functions and flows to components and variables. On the other hand, manual mapping requires the designer to make these relations by looking at both the functional and the simulation models and deciding how the two models relate. Either way, the functional debugger needs access to functional models, simulation models, and the mapping model.

3.2.1 Mapping of Functional Models to Simulation Models

It is possible to relate functional models (functions and flows) to simulation models (components and variables) because the concept of physical quantities exist in both models. Functional models specify *material*, *energy*, and *signal* flows and transformation functions operating on these flows. Physical-based equation-based languages, on the other hand, specify complementary *physical domains* such as *electricity*, *mechanics*, *software*, etc., and the physical behavior of components operating and governed by laws on these domains such as a *resistor*, *gearbox*, or *PID controller*. An important observation is that a single functional *energy* flow maps to a pair of conjugate variables in the simulation model that are used to accomplish *acausal* modeling[5]. These conjugate variables are known differently in different equation-based languages but have very similar semantics. For example, Modelica uses *potential-flow* [33] variables, Bond Graphs use *effort-flow* [12] variables, and Simscape uses *across-through* [28] variables.

Table 2 (adapted from [56]) shows the mapping between flow types (e.g. `electrical`, `magnetic`, etc. in Column 2) in a functional model to conjugate variables (Column 3) in equation-based simulation languages. The last two Columns shows some of the system-level equation-based languages (e.g. Modelica) and domain-specific equation-based languages (e.g. CAD/CAE) that are typically used to simulate physical systems. This table shows that functional models can be mapped to both system-level languages and domain-specific languages and therefore, functional debugging can be adapted to various equation-based languages. Notice that a single equation-based language is not sufficient to cover all the functional flow types. Even though it is out of the scope of this paper, we believe it is important to observe that functional debugging can be also used to comprehend multi-tool multi-language co-simulations of complex systems.

[4] Engineering rules are analogous to machine description files in a traditional compiler.

[5] Acausal modeling describes the behavior of components in terms of energy conservation laws [19].

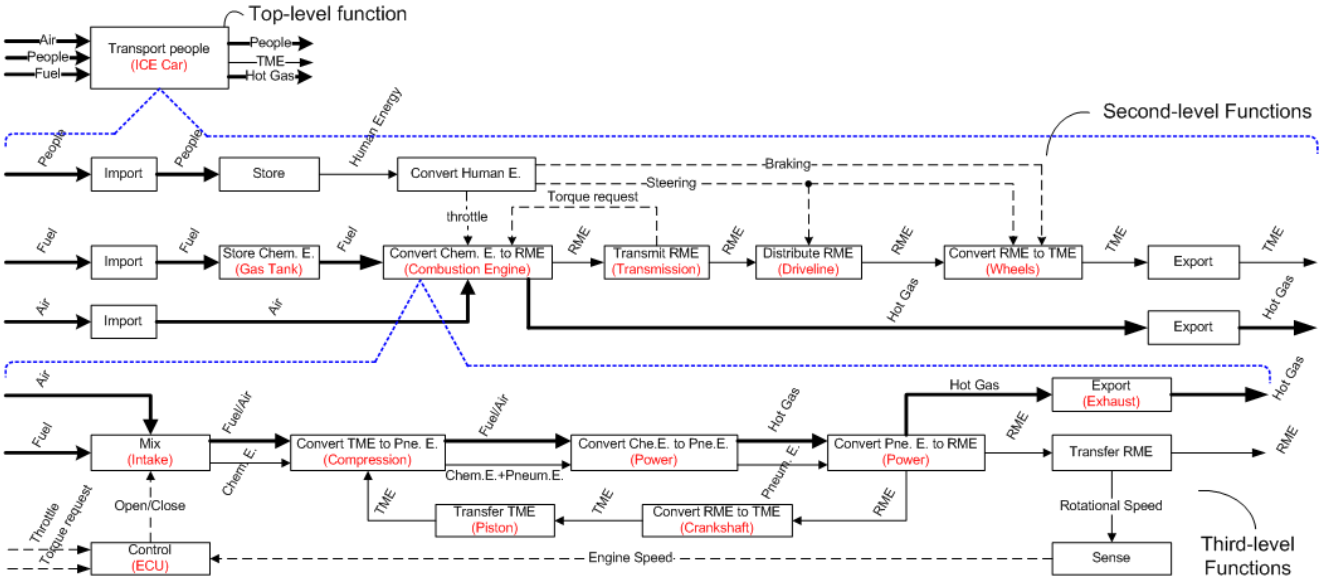


Figure 2. Functional model of an internal combustion engine car showing the functions associated with the main powertrain subsystems (in parentheses). Syntactically and semantically, our functional modeling approach handles feedback loops.

Our functional debugger implementation uses a data structure referred to as the Mapping Model (See Figure 1) to read the mapping information of functions and flows (Functional Model) to components and conjugate variables (Simulation Model). Although it is common that mappings are from *function(s)-to-component(s)* and *flow(s)-to-variable(s)*, other combinations are also possible including *flow(s)-to-component(s)* and *component(s)-to-variable(s)*. For example, a "`pneumatic energy`" flow in a functional model may be mapped to a "`pipe`" component, or to a "`pressure`" variable.

3.3 Simulation Runtime

A simulation runtime responsible for executing the simulation models, is the last component required for functional debugging. Although a simulation model is heavily transformed and optimized into a mathematical model for integration with numerical methods, the variables remain visible *during* simulation. Using the mapping model, the functional debugger can query the variables'status and values during simulation.

From the functional debugging perspective, there are two important requirements for the simulation runtime. First, in order to facilitate a natural human-computer interaction in the functional debugger, the simulation runtime must allow the synchronization of the simulation time with the real (human) time. Whenever the simulation time is faster than the real time, the simulation runtime must delay the execution of the simulation in order to synchronize the two times. In case that the simulation time is slower than the real time, the simulation runtime can adopt execution strategies similar to the ones used in hardware-in-the-loop simulations including fixed-step size solvers, loop tearing, or iterative limits. The second requirement demands the simulation runtime to be programmatically controlled by the functional debugger in order to start, pause, stop, and proceed to the next iteration step during the simulation. Currently, our functional debugger uses Wolfram's SystemModeler [64] as the simulation runtime and the next Section discusses the details of our implementation.

3.4 User Interaction, Visualization, and Simulation Control

The functional debugger consists of three applications as shown in Figure 3. The Functional Editor or Functional Debugger GUI (left) handles the user interaction events such as breakpoints and visualization requests on specific functions and flows. This C# application extends the functionality of Visio through the Visio Object Model [32] and allows the functional debugging specific commands and visualization such as stop, pause, restart, and perform the next iteration step; load functional, simulation, and mapping models; detect user events to debug specific functions and flows and to zoom in/out in the functional model hierarchy; manipulate the look & feel of Visio shapes representing the functional model to convey points of interest during the simulation. These points of interest can be pre-programmed by the user to monitor a range of operation of a subsystem, or built-in into our implementation (e.g. indicate when the energy flow changes direction). In our implementation, the simulation runtime (right) [64] uses an application-specific TCP protocol that allows a client application to control the simulation and set/receive simulation data. After the simulation runtime server has been initialized for control commands and data flow, this application streams data over TCP to the client after every integration time step. Through an initialization file, this application can be configured to maintain the simulation time and the real-time synchronized. The functional debugger application (middle) is the intermediary between the GUI and the simulation runtime. Its main responsibility is to retrieve data from the simulation and map it to the functional model in the GUI, and also to control the simulation according to the user commands.

Although we have developed an in-house implementation, each component of our functional debugger has an analogous technology that could be used to provide

Table 2. Relationship between functional models and equation-based languages based on flow types.

Functional Modeling		Equation-based Languages		
Flow Class	Flow Type	Conjugate Vars. (Effort/Flow)	System-Level Lang.	Domain Specific Lang.
Energy	Electrical	Electromotive Force / Current	[34], [28]	[45], [13]
	Mechanical (Rotational)	Torque / Angular Velocity	[34], [28]	[52], [57], [5]
	Mechanical (Translational)	Force / Linear Velocity	[34], [28]	[52], [57], [5]
	Mechanical (Vibrational)	Amplitude / Frequency		[52], [57], [5]
	Hydraulic	Pressure / Volumetric Flow	[35], [28]	[53]
	Pneumatic	Pressure / Mass Flow	[34]	[53]
	Thermal	Temperature / Heat Flow	[34], [28]	[52], [57]
	Electromagnetic	Intensity / Velocity		[4]
	Magnetic	Mag. Force / Mag. Flux Rate	[34]	
	Chemical	Affinity / Reaction Rate	[38]	
	Biological	Pressure / Volumetric Flow	[38]	
	Human	Force / Motion		
	Acoustic	Pressure / Particle Velocity	[24]	[55]
	Radioactive	Intensity / Decay Rate		
Signal	Status		[41], [29]	[37], [30], [16]
	Control		[41], [29]	[37], [30], [16]
Material	Human			[54], [20] [18]
	Gas			[52], [57], [5]
	Liquid			[52], [57], [5]
	Solid			[54], [20], [18]

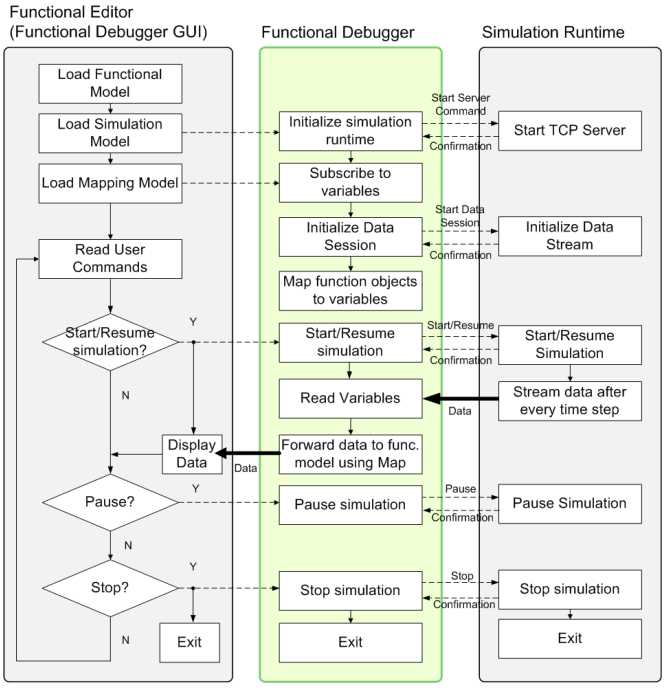


Figure 3. Control and data flow interactions between the functional editor, the functional debugger, and the simulation runtime.

the same functionality. For example, the functional editor could be implemented in SysML [39]. Several published investigations [66, 3] have shown how to create functional models in SysML. Similarly, the simulation model synthesizer could be realized by SysML4Modelica [43] or ModelicaML [50]. Open source Modelica runtimes [40, 22] could be used as the simulation runtime. And FMI [6] could be used as the communication and data transfer mechanism between the functional editor and the simulation runtime.

3.5 Industry Perspective

Concept design, despite being a critical design phase that determines 70-80% of the cost of a product [15], lacks the tool and methodology support that is available for detail design phases [49]. We strongly believe that the development of tools for conceptual design is mandatory to handle the complexity of large-scale cyber-physical systems [63, 47, 27, 58] where system-level optimization plays a critical role. Our functional debugger is a concept design tool that allows product designers to functionally understand the complex underlying cyber-physical processes of a system. Additionally, we see functional debugging as a complementary and orthogonal approach to existing debugging techniques that are employed during detail design. We also believe that functional debugging can be used as a tool to consolidate 3D CAD/kinematics with system-level simulation models. This would allow system designers to have an integrated and dynamic function-behavior-structure [60] view of the system with the capacity for testing and simulating design alternatives while reusing existing components.

4. Case Study: eCar Development

We evaluate our functional debugger with a common scenario in automotive development. In order to reduce risk and cost, automotive companies invest in the development of architectures that can be reused to produce different models of cars within and across brands [14, 7]. Therefore, it is natural that even radical new designs, such as an eCar, attempt to reuse an existing architecture and a set of compatible cyber-physical components. Functional models are used in this type of scenarios to understand the impact of major architectural changes[6] in the overall design. In summary, our objective is to demonstrate how the functional debugger supports a realistic conceptual

[6] An architecture is, after all, the allocation of functions (or functionality) to specific cyber-physical (logical) components (e.g. a gearbox, a wheel, an ECU).

design scenario where an eCar is developed while reusing, as much as possible, components of an existing architecture. Additionally, we show how our functional debugging approach is compatible and orthogonal to the existing debugging techniques for equation-based languages.

4.1 Baseline Architecture

We first created a baseline functional model of an internal combustion engine car shown in Figure 2. This baseline functional model describes the functionality of the automotive driveline industrial example in Modelica language published in [65]. The mapping of functions-to-components is indicated by the parentheses in Figure 2 and this represents the baseline architecture for our scenario. Using the baseline models as the starting point, the next step is to conceptually design an eCar with the help of functional debugging.

4.2 Concept Design Space Exploration

Conceptually, the simplest way to create an eCar from a conventional car is by replacing its internal combustion engine with an electric motor. In terms of functionality, the function "`convert chemical energy to rotational mechanical energy`" must be replaced with "`convert electrical energy to rotational mechanical energy`" as shown in Figure 4. This change also implies new functionality where the "`electrical energy`" flow is "`stored`" (e.g. in a battery).

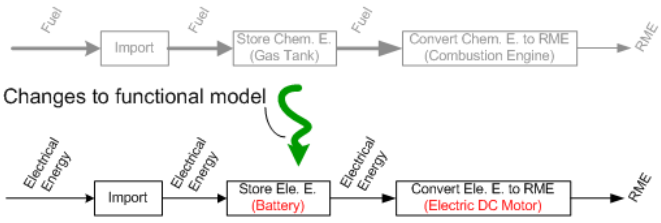


Figure 4. Changes to the functional model in Figure 2 to convey the new design intentions of an e-Car. Fuel containing chemical energy is replaced with electrical energy. This implies the use of a Battery and an Electric DC Motor instead of a Gas Tank and a Combustion Engine.

To mimic the reusability aspect in the current system engineering practice, we created new engineering rules to convert the newly introduced eCar functionality into existing simulation models of a DC motor and a battery developed in-house [62]. As a result, the synthesizer creates an aggregated simulation model that replaces the internal combustion engine component from the baseline simulation model with the battery and DC motor, but reuses the rest of the simulation components in the baseline. The coupling between the DC motor and the baseline drivetrain is possible because the two have a compatible interface and this allows the aggregated simulation model to be correctly generated and compiled using SystemModeler.

This workflow shows that a simple change in the functional model can be used to generate new simulation models that allow the designer to understand and quantify how a change in functionality of an existing architecture has an impact in the overall system-level design. The relation of functions to components, or mapping model, was created by the engineering rules and the analogy between functions and system-level equation-based languages discussed in Table 2. Therefore, at this point in time, the three input models to the functional debugger are available: an eCar functional model (Visio), a simulation model (Modelica), and the mapping model (data structure described in Section 3.2.1). The next step is to run the eCar simulation model under the functional debugger to identify any possible system-level problems created by the architectural change.

4.3 Functional Debugging

We use the New European Driving Cycle to test the eCar simulation model in the functional debugger. Figure 5 shows the functional debugger under four modes of operation: (a) Acceleration, (b) Cruise, (c) Deceleration, and (d) Idle. During acceleration in Figure 5(a), the functional debugger shows that the main energy transfer in the power train, indicated by the direction of the flows, is from left-to-right starting from the "`convert electrical energy to rotational mechanical energy`". While the Modelica simulation explains the physical behaviors, the functional debugger helps a non-expert to understand that rotational mechanical energy (RME) is functionally correct. During cruise in Figure 5(b), the functional debugger shows that there is an equilibrium of energy transfer in the powertrain and this is indicated by the bi-directional flows. During deceleration in Figure 5(c), the functional debugger shows that RME flow is from right-to-left. In addition, this functional debugging snapshot shows that the function being performed by the "Electric Motor" component changed to "Convert RME to electrical energy". This insight is very important for the systems engineer because it shows that the newly introduced electric motor is performing two functions and this can be used to validate the requirements. It is also important to note that this additional functionality can be used to recharge the battery while the eCar decelerates. During the idle mode in Figure 5(d), the functional debugger shows the case when the clutch is disengaged and the electric motor and the transmission are physically decoupled, and the functional debugger eliminates the flow connecting these two components. This causes the functionality of the electric motor to change to "`convert electrical energy to thermal energy`".

The snapshots in Figure 5 illustrate how the functional debugger can help the concept level designer to create a mental high-level picture of the system and conceptually understand how a functional and architectural change affects the rest of the system. It also allows them to visualize potential new innovations such as regenerative breaking, and visualize the energy, material, and signal flows through the system. Functional debugging can be easily integrated to the current systems engineering processes and reuse the existing and legacy simulation, functional, and architectural models. Another important feature is that functional debugging allows any non-technical person to easily understand the cyber-physical process at the functional or conceptual level.

Iterative design is a very important aspect of the systems engineering process. Although the results shown in Figure 5 are functionally correct as the system does what it is supposed to, the systems engineer must verify that

Figure 5. Visualization of an eCar simulation through the functional debugger on different modes: (a) acceleration, (b) cruise, (c) deceleration, (d) idle. Notice the energy flows across the functional model on the different modes, and the mapping of multiple functions to a single component (e.g. Electric Motor).

the eCar is reaching its performance targets by enabling the display of numerical values of the conjugate variables of the simulation in the functional model. Enabling the numerical values in the functional debugger reveals that although the system is functionally correct, the eCar never accelerates to even 5% of the desired speed. Since the eCar concept includes a newly introduced component, the electric motor, the first guess would probably be that the test needs a stronger motor. However, after simulating the eCar with a stronger electric motor, the results are still unfavorable. These quick iterations that use the functional debugger as a visualizer for the underlying simulation provide valuable information to the systems engineer about the system-level integration problems on the new concept at a level of abstraction where they can reason about the possible problem, but without being concerned about the details of the cyber-physical implementation.

Typically, this situation would lead the systems engineer to report and discuss the problem with the multi-disciplinary teams in charge of the transmission, the software, and the electro-mobility. Using the common language of functionality, engineers can communicate at a high-level of abstraction and then translate these insights to their domain of expertise. At this point in time, the domain experts would perform detail design iterations on their subsystems and this is where existing debugging techniques for equation-based languages are very useful for identifying the root cause of the problem. In this example, the problem is in the control gains in the controller software at the transmission control unit, and it required an expert to use the existing debugging techniques to find the solution. Because functional debugging is used early in the concept design phase, and its purpose is to communicate potential problems to the systems engineer using a high-level of abstraction (functionality) rather than a low-level of abstraction (behavior), we argue that it enhances the systems engineering and it is complementary and orthogonal to the existing debugging techniques for equation-based languages.

5. Summary

With the objective of supporting the early concept design phases with computer-based tools, we introduced a new methodology referred to as *functional debugging* that builds a functional view of an underlying cyber-physical process described in equation-based languages. Our implementation couples functions and flows in functional models with conjugate variables in simulation models, and this mapping enables a high-level view of *what* the system does. In a systems engineering context, our functional debugger can be used as a rapid prototyping tool for new concepts to identify system-level integration problems. Through an industrial use-case, we have shown that functional debugging can be a valuable tool for an iterative design process that involves the coordination of multiple disciplines. Additionally, we have shown that functional debugging is compatible with existing low-level debugging techniques for equation-based languages. Our future work will include the implementation of functional debugging for domain-specific equation-based languages.

Acknowledgments

The authors would like to thank Eric Schwarzenbach from Princeton University and Georg Muenzel from Siemens Corporate Technology for their support during the research and development of the functional debugger.

References

[1] Foundations for innovation: Strategic R&D opportunities for the 21st century cyber-physical systems – connecting computer and information systems with the physical world. Technical report, NIST, 2013.

[2] Aberdeen Group. System design: New product development for mechatronics. January 2008.

[3] A. A. Alvarez Cabrera, M. S. Erden, and T. Tomiyama. On the potential of function-behavior-state (FBS) methodology for the integration of modeling tools. In *Proc. of the 19th CIRP Design Conference âĂŞ Competitive Design*, pages 412–419, 2009.

[4] ANSYS. HFSS. `http://www.ansys.com`.

[5] Autodesk. Simulation Software. `http://www.ni.com/labview/`.

[6] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauss, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, and S. Wolf. The functional mockup interface for tool independent exchange of simulation models. In *Proceedings of the 8th Modelica Conference*, pages 105 – 114, 2011.

[7] Manfred Broy, Mario Gleirscher, Peter Kluge, Wolfgang Krenzer, Stefano Merenda, and Doris Wild. Automotive architecture framework: Towards a holistic and standardised system architecture description. Technical report, TUM, 2009.

[8] Cari R. Bryant, Robert B. Stone, Daniel A. Mcadams, Tolga Kurtoglu, and Matthew I. Campbell. Concept generation from the functional basis of design. In *Proc. of International Conference on Engineering Design, ICED 2005*, pages 15–18, 2005.

[9] Peter Bunus. An empirical study on debugging equation-based simulation models. In *Proceedings of the 4th International Modelica Conference*, pages 281 – 288, 2005.

[10] A.A. Alvarez Cabrera, M.J. Foeken, O.A. Tekin, K. Woestenenk, M.S. Erden, B. De Schutter, M.J.L. van Tooren, R. Babuska, F.J.A.M. van Houten, and T. Tomiyama. Towards automation of control software: A review of challenges in mechatronic design. *Mechatronics*, 20(8):876 – 886, 2010.

[11] A. Canedo, E. Schwarzenbach, and M. A. Al-Faruque. Context-sensitive synthesis of executable functional models of cyber-physical systems. In *ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2013.

[12] F. E. Cellier. *Continuous System Modeling*. Springer-Verlag, 1991.

[13] E. Christen and K. Bakalar. Vhdl-ams-a hardware description language for analog and mixed-signal applications. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 46(10):1263 –1272, oct 1999.

[14] Jeffrey B. Dahmus, Javier P. Gonzalez-Zugasti, and Kevin N. Otto. Modular product architecture. *Design Studies*, 22(5):409–424, September 2011.

[15] D. Dumbacher and S. R. Davis. Building operations efficiencies into NASA's Ares I crew launch vehicle design. In *54th Joint JANNAF Propulsion Conference*, 2007.

[16] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, jan 2003.

[17] M.S. Erden, H. Komoto, T.J. Van Beek, V.D'Amelio, E. Echavarria, and T. Tomiyama. A review of funcion modeling: approaches and applications. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 22:147–169, 2008.

[18] G. S. Fishman. *Discrete-Event Simulation - Modeling, Programming, and Analysis*. Springer, 2001.

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

[20] D. Gross, J. F. Shortle, J. M. Thompson, and C. M. Harris. *Fundamentals of Queueing Theory*. Wiley, 4th edition, 2011.

[21] Julie Hirtz, Robert B. Stone, Simon Szykman, Daniel A. McAdams, and Kristin L. Wood. A functional basis for engineering design: Reconciling and evolving previous efforts. Technical report, NIST, 2002.

[22] JModelica. `http://www.jmodelica.org/`.

[23] Hitoshi Komoto and Tetsuo Tomiyama. A framework for computer-aided conceptual design and its application to system architecting of mechatronics products. *Comput. Aided Des.*, 44(10):931–946, October 2012.

[24] H. Kuehnelt, Thomas Baeuml, and Anton Haumer. SoundDuctFlow: a Modelica library for modeling acoustics and flow in duct networks. In *Proc. of the 7th Intl. Modelica Conference*, pages 519–525, 2009.

[25] Tolga Kurtoglu and Matthew I. Campbell. Automated synthesis of electromechanical design configurations from empirical analysis of function to form mapping. *Journal of Engineering Design*, 19, 2008.

[26] Lawrence Berkeley National Laboratory - Modelica Buildings Library. `http://simulationresearch.lbl.gov/modelica`.

[27] Insup Lee, Oleg Sokolsky, Sanjian Chen, John Hatcliff, Eunkyoung Jee, BaekGyu Kim, Andrew L. King, Margaret Mullen-Fortino, Soojin Park, Alex Roederer, and Krishna K. Venkatasubramanian. Challenges and research directions in medical cyber-physical systems. *Proceedings of the IEEE*,

100(1):75–90, 2012.

[28] MathWorks. Simscape. http://www.mathworks.com/products/simscape/.

[29] MathWorks. Simulink. http://www.mathworks.com/products/simulink/.

[30] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[31] Oregon state university, design engineering lab, design repository. http://designengineeringlab.org/.

[32] Microsoft. Visio. http://msdn.microsoft.com.

[33] Modelica Association, Modelica. https://modelica.org/.

[34] Modelica Association, Modelica Standard Library. https://modelica.org/libraries/Modelica/.

[35] Modelon. Hydraulics Library. http://www.modelon.com/products/modelica-libraries/hydraulics-library/.

[36] Modelon - Vehicle Dynamics Library. http://www.modelon.com/.

[37] National Instruments. LabVIEW System Design Software. http://www.ni.com/labview/.

[38] E. L. Nilson and P. Fritzon. BioChem - A Biological and Chemical Library for Modelica. In *Proc. of the 3rd Intl. Modelica Conference*, pages 215–220, 2003.

[39] OMG Systems Modeling Language (SysML). http://www.omgsysml.org/.

[40] OpenModelica. https://www.openmodelica.org/.

[41] Martin Otter, Karl-Erik Årzén, and Isolde Dressler. Stategraph—A Modelica library for hierarchical state machines. In *Modelica 2005 Proceedings*, 2005.

[42] G. Pahl, W. Beitz, J. Feldhusen, and K.H. Grote. *Engineering Design - A Systematic Approach*. Springer, 3rd edition, 2007.

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

[44] Adrian Pop, Martin Sjolund, Adeel Asghar, Peter Fritzson, and Francesco Casella. Static and dynamic debugging of Modelica models. In *Proceedings of the 9th International Modelica Conference*, pages 443 – 454, 2012.

[45] Thomas L. Quarles. *Analysis of Performance and Convergence Issues for Circuit Simulation*. PhD thesis, EECS Department, University of California, Berkeley, 1989.

[46] Venkat Rajagopalan, Cari R. Bryant, Jeremy Johnson, Daniel A. McAdams, Robert B. Stone, Tolga Kurtoglu, and Matthew I. Campbell. Creation of assembly models to support automated concept generation. *ASME Conference Proc.*, 2005(4742Xa):259–266, 2005.

[47] Ragunathan (Raj) Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-physical systems: the next computing revolution. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 731–736, New York, NY, USA, 2010. ACM.

[48] S.D. Rudov-Clark and J. Stecki. The language of FMEA: on the effective use and reuse of FMEA data. In *AIAC-13 Thirteenth Australian International Aerospace Congress*, 2009.

[49] Alberto Sangiovanni-Vincentelli. Quo vadis SLD: Reasoning about trends and challenges of system-level design. *Proceedings of the IEEE*, 95(3):467–506, March 2007.

[50] Wladimir Schamai, Peter Fritzson, Chris Paredis, and Adrian Pop. Towards unified system modeling and simulation with ModelicaML: Modeling of executable behavior using graphical notations. In *Proceedings of the 7th Modelica Conference*, pages 612 – 621, 2009.

[51] Peter Schwarz. Physically oriented modeling of heterogeneous systems. *Math. Comput. Simul.*, 53(4-6):333–344, October 2000.

[52] Siemens. NX. http://www.plm.automation.siemens.com.

[53] Siemens. Simulation & Testing SIMIT. http://www.siemens.com.

[54] Siemens. Technomatix. http://www.plm.automation.siemens.com.

[55] MCS Software. Actran Acoustics. http://www.mscsoftware.com/product/actran-acoustics.

[56] Robert B. Stone and Kristin L. Wood. Development of a functional basis for design. *Journal of Mechanical Design*, 122(4):359–370, 2000.

[57] Dassault Systemes. SolidWorks. http://www.solidworks.com/.

[58] Janos Sztipanovits, Xenofon Koutsoukos, Gabor Karsai, Nicholas Kottenstette, Panos Antsaklis, Vijay Gupta, Bill Goodwine, John Baras, and Shige Wang. Toward a science of cyber-physical system integration. *Proceedings of the IEEE*, 100(1):29–44, 2012.

[59] Serdar Uckun. Meta II: Formal co-verification of correctness of large-scale cyber-physical systems during design. Technical report, Palo Alto Research Center, 2011.

[60] Y. Umeda, H. Takeda, T. Tomiyama, and H. Yoshikawa. Function, behaviour, and structure. *Applications of artificial intelligence in engineering V*, 1:177–194, 1990.

[61] Thom J. van Beek, Mustafa S. Erden, and Tetsuo Tomiyama. Modular design of mechatronic systems with function modeling. *Mechatronics*, 20(8):850 – 863, 2010.

[62] A. Votintseva, P. Witschel, and A. Goedecke. Analysis of a complex system for electrical mobility using a model-based engineering approach focusing on simulation. *Procedia Computer Science*, 6(0):57 – 62, 2011.

[63] Wolfgang Wahlster. Industry 4.0: From smart factories to smart products. In *Forum Business Meets Research BMR 2012*, 2012.

[64] Wolfram. Systemmodeler. http://www.wolfram.com/system-modeler/.

[65] Wolfram. Systemmodeler. http://www.wolfram.com/system-modeler/industry-examples/automotive-transportation/.

[66] Stefan Wölkl and Kristina Shea. A computational product model for conceptual design using SysML. *ASME Conference Proc.*, 2009(48999):635–645, 2009.

Toward an Equation-Oriented Framework for Diagnosis of Complex Systems

Alexander Feldman Gregory Provan

University College Cork, Ireland

a.feldman@ucc.ie, g.provan@cs.ucc.ie

Abstract

Diagnosis of complex systems is a critical area for most real-world systems. Given the wide range of system types, including physical systems, logic circuits, state-machines, control systems, and software, there is no commonly-accepted modeling language or inference algorithms for model-Based Diagnosis (MBD) of such systems. Designing a language that can be used for modeling such a wide class of systems, while being able to efficiently solve the model, is a formidable task. The computational efficiency with which a given model can be solved, although often neglected by designers of modeling languages, is a key to parameter identification and answering MBD challenges. We address this freedom-of-modeling versus model-solving efficiency trade-off challenge by evolving a language for MBD of physical system, called LYDIA. In this paper we report on the abilities of LYDIA to model a class of physical systems, the algorithms that we use for solving MBD problems and the results that we have obtained for several challenging systems.

Keywords model-based diagnosis, model-based testing, automated reasoning, modeling language

1. Introduction

Diagnosing complex systems using a model-based diagnosis (MBD) approach has led to the development of several languages, most of which are extensions of simulation languages. Examples include logic-based diagnosis languages (which extend multi-valued logics), e.g., [20]; bond-graph diagnosis languages (which extend bond-graphs), e.g., [21]; and MODELICA-based diagnosis languages (which extend MODELICA), e.g., [2].

Each of these languages has strong and weak points. The logic-based diagnosis languages have a well-defined diagnosis semantics and cleverly-designed inference algorithms, but are limited in the types of behaviours that they can be described by logic-based constraints.

The bond-graph and MODELICA-based diagnosis languages can describe a wider class of systems, but do not have a well-defined diagnosis semantics or efficient diagnostics inference algorithms. For example, MODELICA focuses on a single nominal mode of a system[1], whereas a diagnosis language must be able to specify behaviours for all the pertinent nominal and faulty modes of the system.

In this article we propose a diagnosis framework that aims to provide the expressive power of a MODELICA-based diagnosis language together with a clear diagnosis semantics and efficient diagnostics inference algorithms. The language, LYDIA, is a component-based, hierarchical language that supports dynamical systems (based on ODEs) as well as logical constraint-based representations. In addition, the framework provides a range of simulation and fault-isolation algorithms. Hence, LYDIA provides many of the simulation-oriented capabilities of a Modelica implementation together with the associated mode-identification and fault-isolation algorithms. Further, LYDIA allows modes to be identified by their likelihood of explaining the observed data, using a variety of likelihood metrics.

Our contributions are as follows:

1. We describe a modeling language, LYDIA, that supports the specification of mode-based behaviours for nominal and faulty models;
2. We describe a framework that can simulate, identify modes and isolate faults for models described in the LYDIA language;

2. Related Work

This section compares our approach to that of some key existing equation-oriented systems/languages, e.g., MODELICA (`http://modelica.org/`), bond graphs (`http://bondgraph.org/`), and Rodelica [5].

2.1 Diagnostic Approaches

MODELICA is a mixed declarative/procedural language; see, e.g., [12]. The declarative aspect involves the dynamical systems equations; the procedural aspect involves the specification of code

[1] Of course, it is possible to have multiple modes in MODELICA, however, the price is often increased modeling and simulation complexity.

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. 19 April, 2013, University of Nottingham, UK.

 Proceedings available at: `http://www.ep.liu.se/ecp_home/index.en.aspx?issue=084`

EOOLT 2013 website: `http://www.eoolt.org/2013/`

fragments within the model itself. It is noted in [12] that the semantics of MODELICA is defined not just by the declarative aspects of the model, but also by the compilation of the model that aims at optimizing simulation efficiency. In other words, semantics is also provided by the process of translation of a hierarchical model, which consists of a hierarchically-nested set of classes, instances and connections, into a flat model, consisting of a set of constants, variables and equations. In the flat model, an optimization/compilation step performs several operations, including sorting and conversion of equations to assignment statements. Next, strongly-connected sets of equations are solved using symbolic and/or numeric solvers.

In contrast to MODELICA, the LYDIA language is fully declarative. For example, whereas MODELICA allows users to define procedural entities (code fragments) within the model itself, LYDIA only allows declarative statements developed from syntactically correct language statements. Allowing in-model algorithm specification is problematic for model-based reasoning, since this interferes with the symbolic manipulation of equations that is crucial to simulation and diagnostics inference.

Bond graphs are an equation-oriented language which bear a close relation to MODELICA; in fact, a bond graph library exists in MODELICA [7]. Bond graphs constrain MODELICA to represent systems in terms of energy and power flows, thus forming a semantic framework for physical systems. Bond graphs model physical systems in terms of four entities: effort e, flow f, generalized momentum p, and generalized displacement q. A graph G enables the specification of energy flows among components, where a node corresponds to a component and an edge to a bond (i.e., a flow/effort interaction) between the joined components. The semantics of a bond graph G is specified through the assignment of differential- algebraic equations to each of the nodes and edges of the graph G, as based on mapping the graph structure (noting the connection semantics of the two basic connection types, parallel and series connection) into equations. These differential-algebraic equations describe the behavior of the four variables $p,\ q,\ e$ and f, for each of the physical components in the system (i.e., the nodes in the graph G).

Standard bond graphs have no notion of mode, or mode-based inference. Extended models, e.g., [21], have been developed, but the approach is quite different to that of LYDIA. For example, LYDIA does not impose any flow/energy restrictions to semantics, and its notion of mode is an inherent part of the language, rather than an extension.

RODELICA is a diagnosis language based on MODELICA, and used as the basis for the diagnostics system RODON [5, 4]. Rodelica is similar in structure to the LYDIA-NG language, in that it specifies component modes along with their associated behaviours. However, Rodelica is strictly more limited than the LYDIA-NG language, in that it allows not full ODSs but point- or interval-valued arithmetic constraints. In addition, a Rodelica model is restricted to atemporal equations (and hence uses data from one time instance), and cannot define the stochastic occurrence of faults in components. On top of this, the Rodon diagnostics system is limited to a single inference engine, as opposed to the ability of LYDIA-NG to use multiple inference engines and residual generators.

MATLAB/SIMULINK models (`http://mathworks.com/`) have a highly procedural semantics associated with simulation of a block-oriented model. Procedural tools for execution of a Simulink block during a given simulation step are governed by a number of factors; these include, among others, whether or not the block (or a subsystem containing the block) has a sample time, or whether or not the block resides in a conditionally executed subsystem. Block execution can also be disabled by *conditional input branch* statements. Matlab/Simulink has no inherent language framework for modes, nor well-established algorithms for diagnosis. As mentioned earlier, the LYDIA language is fully declarative and has associated algorithms for simulation and diagnostics inference.

2.2 LYDIA versus MODELICA

LYDIA is an equation-based language, i.e., a LYDIA model is translated to a system of equations. These equations can be Boolean, linear, or systems of ODEs. One of the major differences between LYDIA and MODELICA is the approach to solving systems of equations. MODELICA can solve DAEs. LYDIA tries to identify the type of the system of equations and invoke appropriate solver (simulation engine). For example, if LYDIA-NG detects a model that contains Boolean variables only, it will not use an ODE solver but a SAT algorithm which is better optimized for this class of systems. The idea is to use specialized solvers for various tasks, for examples trigonometric systems, etc.

LYDIA is in the same category of equation/simulation-based declarative languages but is targeted toward the diagnostic user-group. These are the the major differences between the two languages:

Syntax: LYDIA evolved from several diagnostic projects. The design of the language syntax was probably dictated by the experience of the language designers. LYDIA has syntactical resemblance to C, VERILOG, and ADA.

Type system: Both LYDIA and MODELICA are strongly typed languages. LYDIA optimizes heavily the use of Boolean variables.

Object orientation: MODELICA is an object-oriented language, while LYDIA is not. From all features of object-oriented languages [14], the most important one for equation-oriented approaches is inheritance which may lead to more compact models. LYDIA uses external pre-processors to achieve the same goals, however, in future extensions of the language the authors of LYDIA may bring inheritance into the language. Information hiding is supported by MODELICA and not supported by LYDIA. Information hiding may help modelers avoid mistakes.

Explicit procedures: LYDIA does not support MODELICA-type algorithm sections. The reason for that is that while MODELICA models are typically used from simulation only, LYDIA models are used for multiple simultaneous simulations. This imposes strict requirements on (1) the computational efficiency of the simulation and (2) the side-effects of the simulation. An example of a side-effect would be a MODELICA algorithm using a file on disk. In LYDIA this would create a problem as this file would be overwritten by the multiple simultaneous simulations for the various fault-modes. Further it is very difficult to have automated performance analysis on procedural code.

Units: LYDIA does not support directly units. Units can be specified as string attributes, but there is no unit algebra. Units may be supported as first-class citizens in future versions of LYDIA.

Modes: The most important difference between LYDIA and MODELICA is the use of modes in LYDIA. This is not strictly a language difference but an issue of interpreting the models. LYDIA detects health and user-input variables and identifies components based on these variables. This can be easily achieved in MODELICA by using special variable types,

e.g., according to a naming convention. LYDIA also identifies which equations belong to which component mode.

3. LYDIA Modeling Examples

This section describes the LYDIA language through examples, rather than use a formal approach. Viewed simply, LYDIA enables users to define models in terms of constraints, where constraints may range from logic to differential equations. Further, LYDIA supports component-based definitions of systems, such that for each component we can associate a *mode* that represents the distinct functional modes that drive the component's behaviors. When a system is composed, the system-level modes consist of the cross-product of the component modes, and the system equations consist of the union of the mode-based component equations. We use standard methods for component composition, e.g., [13].

A LYDIA model has four sections: prologue, domains, structure, and components. The prologue describes the main characteristics of the model, i.e., the types of the constraints, fault-modeling, etc. The structure displays the model hierarchy that is essential for many of the MBD algorithms existing today. The domain description specifies symbolic values for all the Finite Domain Integer (FDI) variables (Booleans are treated as a special case of many-valued logic). Finally, for each component a set of constraints and transitions are specified. In particular, LYDIA supports constraints ranging from logic to differential equations.

The models discussed in this section come from three different domains. The first one is an analogue electrical circuit. The example shown in section 3.1 is a logic circuit similar to the ones that are used for benchmarking of Automated Test Patter Generation (ATPG) [3]. Finally, the third example illustrates the use of Ordinary Differential Equations (ODEs) in LYDIA.

Another difference between LYDIA and simulation languages like SIMULINK and MODELICA is that LYDIA splits the model in two: system model and diagnostic scenario. The system model is similar to what we have in other modeling languages. The diagnostic scenario contains sensor data, initial values and other information that is unknown at modeling time. This allows the use of compilation methods—the system model does not change and can be compiled (sometimes in the strict sense of the term [8]) to facilitate reasoning. while the sensor data is supplied at run-time and even includes noise.

3.1 An Analogue Electrical Circuit

Figure 1 shows a small electrical circuit. It consists of a single voltage source (V_1), two switches (SW_1 and SW_2), and a current sensor (I_1). If we disregard the switches and the sensor (take the voltage source and resistors only) we can easily calculate all node voltages and branch currents. This can be done with an electronic calculator or with a simulation program like SPICE [16]. LYDIA-NG implements SPICE simulation.

When modeling a system in LYDIA we start with each component separately. With some luck, these are standard components and are already modeled in a component library. Otherwise the component models have to be created. We next show a resistor model in LYDIA. This is already part of the electrical component library `electrical.lcl`.

```
1  type ResistorHealth = enum { nominal, open, short };
2
3  system Resistor ( float resistance , current , pn, nn)
4  {
5      ResistorHealth h;
6
7      attribute health (h) = (h == ResistorHealth .nominal);
8
9      switch (h) {
10         ResistorHealth .nominal ->
11         {
12             resistor ( resistance , current , pn, nn);
13         }
14         ResistorHealth .open ->
15         {
16             // no constraint
17         }
18         ResistorHealth . short ->
19         {
20             resistor (0, current , pn, nn);
21         }
22     }
23 }
```

Listing 1. Resistor model

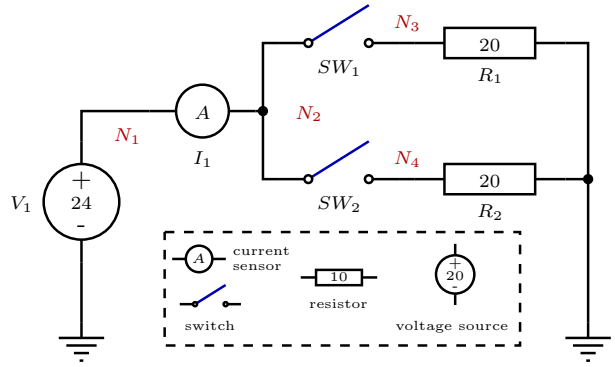


Figure 1. A small power distribution network

Line 1 defines a new discrete LYDIA type that will be used for the health of the resistor. This type defines three modes for the resistor: nominal, open-circuited, and short-circuited. The first mode is a nominal and the other two are fault-modes. The health variable itself is declared in line 5. We specify in the model which mode is the nominal and which modes are the fault modes by adding a variable attribute. This is done in line 7 of the resistor model.

Component models in LYDIA typically follow the same structure. LYDIA simulates for a set of nominal/fault modes. The choice which simulation goes for which fault mode is made in line 9 of the resistor model. When diagnosing, disambiguating, or otherwise reasoning, LYDIA-NG, will pick the relevant equations (constraints) depending on the hypothesized (assumed) value of the health variables (in the case of the resistor model above, the health variable is h).

The actual resistor equations (constraints) are specified in lines 12, 16, and 20. Notice that in the case of an open-circuit, there is no constraint, i.e., an open-circuited resistor is modeled with an empty set of constraints. This is equivalent to specifying a resistor with infinite resistance but eliminates the need of this resistor to be pruned (LYDIA-NG supports infinite resistance and conductance through symbolic preprocessing). For the nominal mode we specify the built-in constraint resistor , parametrized with its nominal resistance. LYDIA-NG will take this and fill-in the proper values in a nodal equation matrix so it can compute the unknown voltages and currents. In the case of a short-circuit

(line 20), we make the resistance parameter zero and LYDIA-NG knows how to deal with this case during simulation.

When diagnosing, LYDIA-NG will choose constraints based on hypothesized fault modes and construct a simulation model. This simulation model will be simulated with a domain-specific solver (in the above case with SPICE). This process will be repeated multiple times until the proper fault mode is identified.

We next describe a component that cannot be fully-simulated with SPICE . This is the current sensor. The current sensor consists of a small-resistor (just like the majority of the electronic current sensors do) and some equation that allows the reading of the sensor to be "stuck-at" some value in the presence of a fault. Of course, the last equation has nothing to do with SPICE and is a very simple algebraic equation. This algebraic equation can be solved by value propagation *after* the SPICE simulation finishes. LYDIA-NG partitions the constraints (equations) and invokes the appropriate solvers (up until now we have mentioned the SPICE solver and a simple algebraic propagation-based solver) automatically. Here is a model of a current sensor:

```
1  type SensorHealth = enum { nominal, failed };
2
3  system CurrentSensor( float pn, nn)
4  {
5      float r;
6
7      attribute observable(r);
8      attribute name(r) = "current␣[A]";
9
10     float current;
11
12     SensorHealth h;
13
14     attribute health(h) = (h == SensorHealth.nominal);
15
16     switch (h) {
17         SensorHealth.nominal ->
18         {
19             resistor (0.01, current, pn, nn);
20             r = current;
21         }
22         SensorHealth.failed ->
23         {
24             resistor (0.01, current, pn, nn);
25             r != current;
26         }
27     }
28 }
```

Listing 2. Current sensor model

Last, we show how to model a switch in LYDIA:

```
1  type SwitchHealth = enum { nominal, stuck };
2  type SwitchCommand = enum { open, closed };
3
4  system Switch( float current, pn, nn)
5  {
6      SwitchHealth h;
7      SwitchCommand cmd;
8
9      attribute health(h) = (h == SwitchHealth.nominal);
10     attribute control(cmd) =
11         (cmd == SwitchCommand.closed);
12
13     switch (cmd) {
14         SwitchCommand.open ->
15         {
16             switch (h) {
17                 SwitchHealth.nominal ->
18                 {
19                     // no constraint
20                 }
21                 SwitchHealth.stuck ->
22                 {
23                     resistor (0, current, pn, nn);
24                 }
25             }
26         }
27         SwitchCommand.closed ->
28         {
29             switch (h) {
30                 SwitchHealth.nominal ->
31                 {
32                     resistor (0, current, pn, nn);
33                 }
34                 SwitchHealth.stuck ->
35                 {
36                     // no constraint
37                 }
38             }
39         }
40     }
41 }
```

Listing 3. Single throw switch model

The new feature in the switch model is that we have a user-command variable—the commanded position of the switch. So, this is an example in which we have two parameters: the commanded position and the health. Remember that, when building component models we have to simulate for a subset of the Cartesian product of each user-command/health variable. In the above example we have two possible switch positions (open and close) and two possible modes (nominal and stuck), hence there is a total of four models for each combination of values of the parameters. Remember that the user-commands have to be specified in the outer switch statement and the health models in the inner switch statements. It is also possible to specify each simulation with a sequence of if-statements but using switches is more elegant.

Of course, there are also models of the voltage sensor and the voltage source but we will not discuss those. Fortunately, there are component libraries that come with LYDIA and the user is not required to model standard components. We have shown the three component models above only to explain some basic LYDIA modeling and to provide information for users to design their own component libraries for non-standard components.

Before we are ready to start simulation/diagnosis in LYDIA-NG we have to connect together all components that are shown in figure 1. This is done in the top-level (or main) LYDIA system:

```
1  #include "electrical.lcl"
2
3  attribute void reference;
4
5  system main()
6  {
7      float ground;
8
9      attribute reference(ground);
10
```

```
11      float N1, N2, N3, N4;
12      float V1, R1, R2, SW1, SW2;
13
14      voltage_source (24.0, V1, N1, ground);
15
16      system CurrentSensor I1(N1, N2);
17      system SimpleSwitch SW1(SW1, N2, N3);
18      system SimpleSwitch SW2(SW2, N2, N4);
19      system Resistor R1(20.0, R1, N3, ground);
20      system Resistor R2(20.0, R2, N4, ground);
21  }
```

Listing 4. Top-level system in a model of a power distribution network

The above top-level system starts with the include directive in line 1 so LYDIA can use the electrical component library. LYDIA-NG uses a C-like preprocessor. The first three models in this section were excerpts from the electrical component library.

Notice that the top-level system in a LYDIA model comes last (i.e., first all component and subsystem models and the last system is the top-level one). The rest of the systems and sub-systems do not have to be in a particular order as far as the top-level system is the last one.

The significant part of the top-level system instantiates and connects components from the component library (see lines 16–20). A system instantiation is done by specifying the keyword **system** followed by the type of the system and then the name of the instantiation. After the name of an instance follows (a left parenthesis and) a list of variables. The number and type of variables should match the system interface, otherwise the LYDIA compiler is going to produce an error.

3.2 A System of Boolean Equations

It is straightforward to enter Boolean equations in a LYDIA model. These systems of equations can be used for modeling of digital integrated circuits, or other combinatorial computational devices. Boolean functions are often represented graphically, by using the same symbols as in a standard computer arithmetic schoolbook [19]. An example Boolean function is shown in figure 2. This function implements a full-adder, a device that computes the sum (and the carry) of two Boolean numbers (and carry). By composing multiple of these one can build, for example, a 32-bit adder.

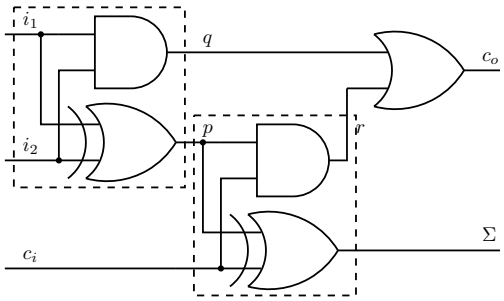


Figure 2. Boolean full adder

The full-adder shown in figure 2 has three types of components, in this case logic-gates: OR-gate, AND-gate, and XOR-gate. This is how a model of an AND-gate looks like:

```
1  system and2(bool o, i1, i2)
2  {
3      bool h;
4
5      attribute health(h) = h;
6
7      h => (o = (i1 and i2));
8      !h => (o = !(i1 and i2));
9  }
```

Listing 5. Model of an AND-gate with two inputs

Let us have a closer look at the above AND-gate model. First, all variables are Boolean. There are one output (o) and two input variables (i1 and i2). These three variables are declared as formals of the system (line 1). The health variable h is declared in line 3. The health attribute in line 5 tells LYDIA that when h equals true, the component is healthy, and otherwise. LYDIA-NG needs this so it can perform diagnosis. When diagnosing, LYDIA-NG performs multiple simulation for different values of the health variables and it needs to know which value means healthy (nominal) and which values means fault.

There are two constraints in the model of the AND-gate: one for when the gate is working nominally (line 7), and one for when the gate is at-fault (line 8). Instead of a **switch**-predicate like in the example in section 3.1, we use conditional expressions to differentiate between the nominal and the fault mode.

We call the model of the AND-gate shown in listing 5, a "stuck-at-opposite" model. This means that when the gate is faulty, the output of the gate has the opposite value of what it is supposed to be for the specified inputs. This is the same as a weak-fault model [10] but allows simulation by simple value propagation for any value of the health variable h.

A full-adder is composed of two half-adders as illustrated in figure 2. Each of the two half-adders in figure 2 is enclosed by a dashed rectangle. Modeling separately the half-adder and composing the full-adder out of the two half-adders and the OR-gate results in non-trivial hierarchy (i.e., a hierarchy where we do not only have component models and a top-level system, but also subsystems). The model of the half-adder simply combines and AND-gate and an XOR-gate as shown next.

```
1  system halfadder2(bool i1, i2, sum, carry)
2  {
3      system and2 A(carry, i1, i2);
4      system xor2 X(sum, i1, i2);
5  }
```

Listing 6. Model of a half-adder

Of course, a large number of Boolean gates, adders and various logic circuits are already modeled in the `std-logic-so.lcl` component library and can be used in any model that includes it.

To conclude the model of the full-adder, we have to compose the two half-adders and an OR-gate into the final top-level design. This is shown in the listing that comes next.

```
1  #include "std-logic-so.lcl"
2
3  attribute void input;
4  attribute void output;
5
6  system fulladder(bool ci, i1, i2, sum, carry)
7  {
8      attribute input(ci, i1, i2);
9      attribute output(sum, carry);
10     attribute observable(ci, i1, i2, sum, carry);
11
12     bool f, p, q;
13
14     system halfadder2 HA1(i1, i2, f, p);
```

```
15      system halfadder2 HA2(ci, f, sum, q);
16      system or2 O(carry, p, q);
17  }
```

Listing 7. Top-level system in a model of a Boolean adder

What is new in the top-level system above, is that in addition to all the subsystems and variables we have two new attributes—input (line 3) and output (line 4). We use these two attributes to denote the primary inputs (ci, i1, and i2), and the primary outputs (sum and carry). LYDIA-NG needs to know what is an input and what is an output so it can simulate, i.e., propagate the values of the Boolean inputs through the circuit to obtain the values of the Boolean outputs.

3.3 Ordinary Differential Equations

One of the first devices for measuring time is a water-filled vessel with a small orifice in it. Measuring time with such a device requires solving a differential equation as the rate of change of the observable quantity (the water height) depends on the amount of water in the vessel. The so called clepsydra problem is a standard problem in schoolbooks on ordinary differential equations [23, p. 108] and can be solved analytically.

In this paper we solve a diagnostic version of the water clock problem. We start with the clepsydra example in *Ordinary Differential Equations: An Elementary Textbook for Students in Mathematics, Engineering, and the Sciences* [22, p. 183] and modify it so it has two holes as illustrated in figure 3. Either of these two holes (or both) can get instantaneously and fully blocked. The goal is, given a measurement of the water height, to determine which of the holes is open and which is stuck. This is how the problem is formulated:

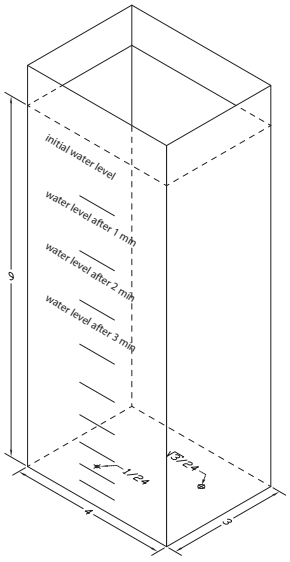


Figure 3. Water clock with two holes that can get blocked

Problem 1. Water flows freely throgh two orificies of a water vessel. It has been established that the rate of flow of water is proportional to the area of the orifices and the square root of the height of the water:

$$Adh = ka\sqrt{h}dt, \tag{1}$$

where

A is the area of the water surface,

h is the height of the water,

r_1 and r_2 are the radii of the orifices.

It has been determined experimentally that $k = -4.8$. In this problem, $A = 12$, $r_1 = \sqrt{3}/24$, $r_2 = 1/24$, and the initial level of the water is $h_0 = 9$.

In addition to all that, there is a discrete parameter $m \in \{0, 1, 2, 3\}$ where $m = 0$ means that both holes are unblocked, $m = 1$ means that the hole with radius r_1 is blocked, $m = 2$ means that the hole with radius r_2 is blocked, and $m = 3$ means that both holes are stuck. Given a measurement of the water α_t at time t and (predicted) levels of the water h_0, h_1, h_2, and h_3 for $m = 0, m = 1, m = 2$, and $m = 3$, respectively, we can compute the value of m from the formula:

$$m = \underset{i \in \{0,1,2,3\}}{\arg\min} |\alpha_t - h_i| \tag{2}$$

Given $h_t = 6.5$ at time $t = 135$, compute the value of the discrete parameter m.

Let us model in LYDIA the clepsydra shown in figure 3.

```
1   type ClepsydraHealth = enum { nominal, s1, s2, sb };
2
3   system Clepsydra()
4   {
5       float A = 12;
6       float r1 = (sqrt(3) / 2) / 12;
7       float r2 = (1 / 2) / 12;
8
9       float height;
10
11      attribute observable(height);
12
13      ClepsydraHealth h;
14
15      attribute health(h) = (h == ClepsydraHealth.nominal);
16
17      switch (h) {
18          ClepsydraHealth.nominal ->
19          {
20              float area = pi * r1 ^ 2 + pi * r2 ^ 2;
21              height' = (-4.8 * area * sqrt(height)) / A;
22          }
23          ClepsydraHealth.s1 ->
24          {
25              float area = pi * r2 ^ 2;
26              height' = (-4.8 * area * sqrt(height)) / A;
27          }
28          ClepsydraHealth.s2 ->
29          {
30              float area = pi * r1 ^ 2;
31              height' = (-4.8 * area * sqrt(height)) / A;
32          }
33          ClepsydraHealth.sb ->
34          {
35              height' = 0;
36          }
37      }
38  }
```

Listing 8. Clepsydra model

The only new feature in the clepsydra model above is the use of derivatives in lines 21, 26, 31, and 35. In this case the dependent variable is height and the independent variable is (implicitly) t.

Notice that t (not used in the clepsydra model) is a keyword in LYDIA.

What remains is to specify the initial value (water height) and the measurement 10 min, after the beginning of the scenario. This is done in a scenario file similar to the one that follows.

```
1  scenario start  @ + 0      {}
2   initial values @ + 0      { height = 9; }
3  observation     @ + 135 s  { height = 6.5; }
4  scenario end    @ + 10 min {}
```

Listing 9. Clepsydra scenario

There are four events in the scenario above. The first and last ones in lines 1 and 4 mark the beginning and the end of the scenario (timestamps are in milliseconds). The initial value for the ODE is given in line 2. In line 3, the scenario supplies the measured water level.

If we now invoke LYDIA-NG to diagnose the above system with the above scenario, we should get that the clepsydra is most likely in state $m = 2$, i.e., the hole with radius r_2 is blocked.

4. LYDIA Concepts and Definitions

We represent a generic linear analogue system in terms of a relation between effort $\vec{x}$ and flow $\vec{z}$ vectors of variables, using $T\vec{x} = \vec{z}$, where T is an $n \times m$ matrix. For example, for circuits $\vec{x} \in \mathbb{R}^n$ is an (unknown) nodal voltage vector, and $\vec{z} \in \mathbb{R}^m$ is a measurable current-source vector.

We will adopt a mode-based representation, i.e., we assume that the system can operate in a set Ω of modes, which can consist of nominal or faulty operating conditions. Given a system that consists of a discrete set of components with a corresponding set of health parameters COMPS, a mode $\omega \in \Omega$ is an assignment to all variables in COMPS.

Further, for each mode we assume that we can specify a distinct set of equations. Hence, for each $\omega \in \Omega$ we specify an equation set SD_ω given by $T_\omega \vec{x_\omega} = \vec{z}_\omega$.

4.1 Models and Residuals

Definition 1 (Diagnostic Model)**.** Given a system that consists of a discrete set of components with a corresponding set of health parameters COMPS, a diagnostic model $M = \langle \mathrm{SD}, \mathrm{COMPS} \rangle$ is specified using a function $\mathrm{SD} = \bigcup_{\omega \in \Omega} \mathrm{SD}_\omega$.

In this article, we are typically given the flow vector $\vec{z}$ and must compute the effort vector via $\vec{x_\omega} = T_\omega^{-1} \vec{z}_\omega$, a process we call *simulation* of $\vec{x_\omega}$. Since SD is linear, we can simulate efficiently.

Given an observation $\vec{\alpha}$, we estimate the mode (i.e., solve a diagnostic problem) by computing an optimal solution of a parameter estimation problem where the parameters are discrete and the problem is split in two parts: simulation and residual analysis.

In real-world applications, straightforward simulation function (from definition 1) is not sufficient to adequately solve the diagnostic problem. This is because models are imprecise, there is sensor noise, health parameters are discrete, etc. Instead, we compute a difference between $\vec{\alpha}$ and a simulation $\hat{\vec{z}}$ (using the residual function of Definition 2), and then identify the mode that minimises this function.

Definition 2 (Residual Function)**.** Given two m-dimensional real vectors $\hat{\vec{z}}, \vec{\alpha} \in \mathbb{R}^m$, a residual function $R : \{\hat{\vec{z}}, \vec{\alpha}\} \mapsto R(\hat{\vec{z}}, \vec{\alpha})$ maps $\hat{\vec{z}}$ and $\vec{\alpha}$ into the real interval $[0; \infty)$.

Definition 3 (Health Estimation Problem)**.** Given a diagnostic model SD and a residual function R, the health estimation problem is to compute an assignment $\omega_{\min}$ to all variables in COMPS such that:

$$\omega_{\min} = \arg\min_{\omega} R(\mathrm{SD}_\omega, \vec{\alpha})$$

Solving the above health estimation problem, while maintaining computational efficiency is the main goal of our framework.

4.2 Control

LYDIA-NG supports control specifications in a straightforward manner. In particular, LYDIA-NG can specify mode-based controls (e.g., as occurs in Finite State Automata [6] and Hybrid Systems [15] models) using the notion of *mode* inherent in the LYDIA-NG language. This is possible since LYDIA-NG associates a set of dynamical equations with a mode (together with constraints on mode transitions), just as in hybrid systems. As such, a LYDIA-NG can specify a declarative control model and use the simulation infrastructure to run closed-loop feedback-control simulations.

Furthermore, the LYDIA-NG framework can support the diagnosis of hybrid systems, e.g., [1]. It is beyond the scope of this article to describe the control and diagnosis functionality of which LYDIA-NG is capable. It is important to note that LYDIA-NG currently has a simple temporal representation, and hence is limited to discrete-event models. In future work we intend to significantly extend the temporal modeling capabilities, and hence be able to analyse a wider range of hybrid models, e.g., as done in [18].

5. Framework for Model-Based Diagnosis

The basic idea of the LYDIA-NG diagnostic library (shown in Fig. 4) is to perform multiple simulations for various hypothesized health states of the plant. The output of these multiple simulations is then processed and combined into single diagnostic output.

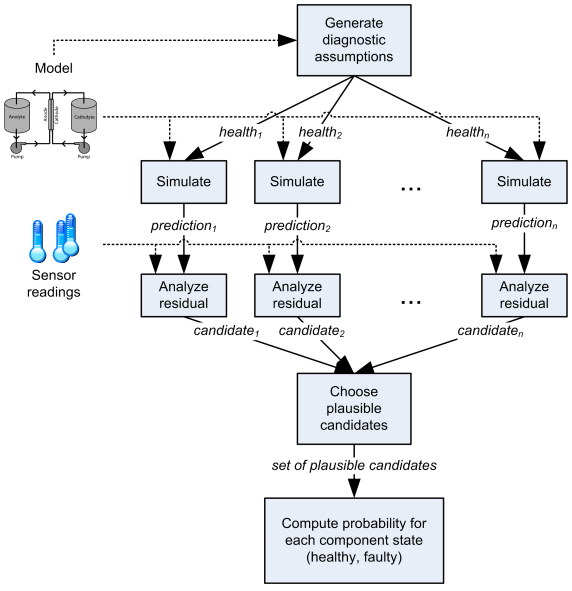


Figure 4. Overview of the LYDIA-NG diagnostic method

The LYDIA-NG diagnostic library consists of the following building blocks:

Generator of Diagnostic Assumptions: A diagnostic assumption is a set of hypothetical assignments for the health or fault state of each component in the system. The "all nominal"

diagnostic assumption assigns healthy status to each component. LYDIA-NG allows one nominal and one or more faulty states per component.

Simulation Engine: Given a diagnostic assumption, LYDIA-NG can construct a simulation model of the system. This simulation model consists of equations. By solving this system of equations LYDIA-NG computes values for one or more *observable* variables. The values of these observable variables is also referred to as a *prediction*.

Residual Analysis Engine: A prediction is compared to the sensor data by a residual analysis engine. This engine combines the individual discrepancies in each sensor data/predicted variable pair to produce a single real value that indicates how close is the prediction of the simulation engine to the sensor data obtained from the plant. A simulation that results in all predicted values coincide with the measured ones will result in the residual being zero. The data structure containing predictions, their corresponding sensor data and the computed residual is called a *diagnostic candidate* or simply *candidate*.

Candidate Selection Algorithm: Not all candidates generated by the residual analysis engine are used for computing the final system health. The candidate selection algorithm discards each candidate whose residual is larger than the residual of the "all nominal" candidate.

System State Estimation Algorithm: LYDIA-NG uses the set of candidates that is computed by the candidate selection algorithm to compute an estimate for the health of each component. This is done by the system state estimation algorithm. Finally, LYDIA-NG computes RCoF by choosing the components with highest probability of failure.

5.1 Search Algorithm

Algorithm 1 shows the top-level diagnostic process. The inputs to algorithm 1 are a model and a scenario, and the result is a diagnosis.

At the heart of algorithm 1 is the use of simulation. Algorithm 1 supports a large variety of simulation methods that may or may not use time as an independent variable. In the setup described in this paper we have used SPICE in combination with a constraint propagation solver. The latter we have used for sensor values, complex components such as mixed analog-digital electronics and other parts of the model where it is difficult or inappropriate to model with SPICE. The only requirement toward the simulation engine is to predict a number of variables whose types can be mapped to LYDIA-NG and to be relatively fast (the computational performance of LYDIA-NG will not be thoroughly discussed in this paper which emphasizes the application of LYDIA-NG to a space model).

The basic idea of algorithm 1 is to simulate for various health assignments and to compare the predictions with the observed sensor data (i.e., telemetry). There are several important aspects of this algorithms that ultimately affect the diagnostic accuracy as measured by various performance metrics.

The first algorithmic property that determines many of the diagnostic performances is the order in which health-assignments are generated. In algorithm 1 this is implemented by the function named NEXTHEALTHASSIGNMENT. The latter subroutine also determines when to stop the search and should be properly parametrized depending on the model and the user requirements. In the standard LYDIA-NG diagnostic library we provide the following diagnostic search policies:

Algorithm 1 Diagnosis framework

```
1: function DIAGNOSE(SCN) returns a diagnosis
       inputs: SCN, diagnostic scenario
       local variables: h, FDI vector, health assignment
                        p, real vector, prediction
                        Ω, a set of diagnostic candidates
                        DIAG, diagnosis, result
2:     while h ← NEXTHEALTHASSIGNMENT() do
3:         p ← SIMULATE(M, γ, h)
4:         r ← COMPUTERESIDUAL(p, α)
5:         Ω ← Ω ∪ ⟨h, r⟩
6:     end while
7:     DIAG ← COMBINECANDIDATES(Ω)
8:     return DIAG
9: end function
```

Breadth-First Search (BFS): This policy first generates the nominal health assignment, then single-faults, double-faults, etc.

Depth-First Search (DFS): This search policy starts with the nominal health assignment, then adds a single-fault, continues with a double fault including the first, and so on, until all components are failed. After the all-faulty assignment is generated, the algorithm backtracks one step and generates a sibling assignment and continues traversing down and backtracking in the same manner until no more backtracking is possible.

Backwards Greedy Stochastic Search (BGSS): In this mode, the search start from the all-faulty assignment. A random health variable is then flipped and the flip is retained iff the flip leads to a decrease in the residual. The order of health variables is arbitrary. As the whole search process is stochastic, it needs to be run multiple iterations in order to achieve the desired completeness. A formal description of this method for Boolean circuit models can be found in [11].

Each simulation produces what we call a *candidate*: a set of predicted values for a given health-assignment. The second important property of algorithm 1 is the comparison and ordering of the diagnostic candidates. This is done by mapping the predicted and observed variables into a single real-number, called *residual*. The residual computation is discussed in what follows.

5.2 Residual Generation

Our mode estimation task requires a method to identify the mode ω whose simulation z_ω most closely matches the observation vector α. We use a residual function $R(z_\omega, \alpha)$ to measure this difference. The specification of $R(z_\omega, \alpha)$ is intentionally generic, since we aim to enable users to specify domain-specific residual generators that are best suited to their application domain.

The area of residual analysis has received significant attention in the literature, and it is not possible to provide a comprehensive set of residual methods within LYDIA-NG.

We currently have implemented a small set of residual generators, the size of which will increase over time.

We provide below examples of two straightforward residual generation functions, together with their advantages and disadvantages. These two residual generation functions bear resemblance to loss functions in decision theory.

Squared Residuals:

$$R_{\text{sq}}(\text{OBS}, \vec{z}, \alpha) = \sum_{v \in \text{OBS}} W(v) \left[\vec{z}(v) - \alpha(v)\right]^2 \quad (3)$$

where $W(v)$ is a weight-value associated with sensor v, $\vec{z}(v)$ is the value of variable v in the prediction assignment $\vec{z}$ and $\alpha(v)$ is the value of the observable variable v.

Absolute Residuals:

$$R_{\text{abs}}(\text{OBS}, \vec{z}, \alpha) = \sum_{v \in \text{OBS}} W(v) \left|\vec{z}(v) - \alpha(v)\right| \quad (4)$$

where $W(v)$, $\vec{z}(v)$ and $\alpha(v)$ are used in the same way as in Eq. 3.

A disadvantage of the squared residuals function R_{sq} is that it adds a lot weight to outliers. In decision theory, the absolute loss function that corresponds to the R_{abs} function is discontinuous. The latter, however, is not a problem for the algorithms described in this paper and we prefer R_{abs} over R_{sq}.

The above two residual functions may lead to relatively bad diagnostic results, especially in the presence of noise. One of the properties of R_{sq} and R_{abs} is that they are memoryless. An alternative would be to use some historical predictions and sensor data. Of course, the use of history would increase the isolation time, but has the potential to also increase the diagnostic accuracy. Another approach for more advanced residual analysis function would be to use methods from machine learning, for example neural networks. Particle filters [9] or Bayesian networks [17] can be also used for residual analysis.

5.3 Computation of Component Failure Probabilities

Consider the circuit shown in Fig. 1 and a scenario $\alpha = \{I_1 = 1.19\}$. This scenario corresponds to one of the resistors being open-circuited or one of the switches being stuck-open. Table 1 shows applying Eq. 4 for the predictions simulated from the nominal and all single-fault health assignments. The rows of Table 1 are sorted in order of an increasing residual value. In this table (and below) we abbreviate a stuck switch as S and an open-circuit resistor mode as OC.

V_1	I_1	SW_1	SW_2	R_1	R_2	faults	R_{abs}
–	–	S	–	–	–	1	0.0006
–	–	–	S	–	–	1	0.0006
–	–	–	–	OC	–	1	0.0006
–	–	–	–	–	OC	1	0.0006
–	–	–	–	–	–	0	1.1758
F	–	–	–	–	–	1	1.1888
–	F	–	–	–	–	1	1.1888
–	–	–	–	SC	–	1	79.3402
–	–	–	–	–	SC	1	79.3402

Table 1. Single-fault residuals for the circuit shown in Fig. 1 and an observation simulated from a single open-circuited resistor

The COMBINECANDIDATES subroutine from algorithm 1 uses a table similar to the one shown in Table 1. It retains only the predictions with residuals smaller than the residual of the nominal prediction. The reason for that is that the nominal prediction is the only one that has a special meaning in LYDIA-NG and leads to a "landmark" residual, i.e., LYDIA-NG does not attempt to differentiate amongst the various fault-mode predictions. As a result, in our running example, only the first four rows of Table 1 are considered when calculating the final fault-probabilities.

The second step of COMBINECANDIDATES is to convert R_{abs} in the interval $[0; 1]$ where $R_{\text{norm}} = 0$ for the nominal prediction and $R_{\text{norm}} = 1$ for a fault prediction that gives $R_{\text{abs}} = 0$. Applying this on Table 1 gives us Table 2.

SW_1	SW_2	R_1	R_2	R_{norm}
S	–	–	–	1
–	S	–	–	1
–	–	OC	–	1
–	–	–	OC	1

Table 2. Normalized single-fault residuals from Table 1 that are smaller than the nominal residual

Finally, what remains to be done is to normalize the rightmost column of Table 2 so it sums up to one and marginalize the probability of failure in each column. For the small circuit we are analyzing this results in $\{\Pr(\text{SW}_1 = \text{S}) = 0.25, \Pr(\text{SW}_2 = \text{S}) = 0.25, \Pr(R_1 = \text{OC}) = 0.25, \Pr(R_2 = \text{OC}) = 0.25\}$. The fact that all probabilities are 0.25 means that algorithm 1 cannot determine unambiguously which component is the faulty one. In this case this is due to the fact that there is only one sensor, i.e., the unambiguity is due to sensor placement and circuit design.

One way to reduce this ambiguity is to change the position of SW_1 and/or SW_2. In the next section we devise an algorithmic framework that works for *any* circuit or model that can be diagnosed in the LYDIA-NG framework.

6. Conclusion and Future Work

This article has described a model-based framework for modeling and diagnostics of complex systems. The framework has several important characteristics, of which we have focused on the modeling language. This LYDIA language is a constraint-based system that enables modelers to specify systems according to a discrete set of system modes, such that each mode is associated with a behaviour. The language allows behaviour specifications based on a wide range of constraints, of which two important constraint representations are ODEs and first-order logic.

We have described several model types that can be developed in LYDIA. In addition, we have proposed a model benchmark for evaluating the capabilities of LYDIA and other MBD languages. By comparing our approach to that of well-known frameworks, such as MATLAB/SIMULINK and MODELICA, we have shown properties of the LYDIA framework that are specific to MBD.

Future work includes extending the range of simulation and diagnosis solvers within our framework, and extending the residual analysis engine.

Acknowledgments

The publishing of this article is supported by Enterprise Ireland grant CC-2011-4005A.

References

[1] Shai A Arogeti, Danwei Wang, and Chang Boon Low. Mode identification of hybrid systems in the presence of fault. *Industrial Electronics, IEEE Transactions on*, 57(4):1452–1467, 2010.

[2] Olof Bäck. *Modelling for diagnosis in Modelica: implementation and analysis*. PhD thesis, University of Linköping, 2008.

[3] Franc Brglez and Hideo Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran. In *Proc. ISCAS'85*, pages 695–698, 1985.

[4] Peter Bunus, Olle Isaksson, Beate Frey, and Burkhard Münker. Model-based diagnostics techniques for avionics applications with rodon. In *2nd Workshop on Aviation System Technology*. Citeseer, 2009.

[5] Peter Bunus, Olle Isaksson, Beate Frey, and Burkhard Münker. Rodon-a model-based diagnosis approach for the dx diagnostic competition. *Proc. DX'09*, pages 423–430, 2009.

[6] Christos G Cassandras and Stephane Lafortune. *Introduction to discrete event systems*, volume 11. Kluwer academic publishers, 1999.

[7] François E Cellier and Àngela Nebot. The modelica bond graph library. In *4th International Modelica Conference*, 2005.

[8] Adnan Darwiche and Pierre Marquis. A knowlege compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[9] Nando de Freitas. Rao-blackwellised particle filtering for fault diagnosis. In *Proc. AEROCONF'02*, volume 4, pages 1767–1772, 2002.

[10] Johan de Kleer, Alan Mackworth, and Raymond Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56(2-3):197–222, 1992.

[11] Alexander Feldman, Gregory Provan, and Arjan van Gemund. Approximate model-based diagnosis using greedy stochastic search. *Journal of Artificial Intelligence Research*, 38:371–413, 2010.

[12] Peter Fritzson and Vadim Engelson. Modelica—a unified object-oriented language for system modeling and simulation. *ECOOP'98—Object-Oriented Programming*, pages 67–90, 1998.

[13] Gregor Gössler and Joseph Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55(1):161–183, 2005.

[14] I. Graham, A. O'Callaghan, and A.C. Wills. *Object-Oriented Methods: Principles & Practice*. Addison-Wesley Object Technology Series. Addison-Wesley, 2000.

[15] Thomas A Henzinger. The theory of hybrid automata. In *Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 278–292. IEEE, 1996.

[16] Ron M. Kielkowski. *Inside SPICE*. Electronic packaging and interconnection series. McGraw-Hill, 1998.

[17] Uri Lerner, Ronald Parr, Daphne Koleer, and Gautam Biswas. Bayesian fault detection and diagnosis in dynamic systems. In *Proc. AAAI'00*, pages 531–537, 2000.

[18] Pieter J Mosterman and Gautam Biswas. A comprehensive methodology for building hybrid models of physical systems. *Artificial Intelligence*, 121(1):171–209, 2000.

[19] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, Inc., New York, NY, USA, 2nd edition, 2009.

[20] Raymond Reiter. A theory of diagnosis from first principles. *Artificial intelligence*, 32(1):57–95, 1987.

[21] AK Samantaray, K. Medjaher, B. Ould Bouamama, M. Staroswiecki, and G. Dauphin-Tanguy. Diagnostic bond graphs for online fault detection and isolation. *Simulation Modelling Practice and Theory*, 14(3):237–262, 2006.

[22] Morris Tenenbaum and Harry Pollard. *Ordinary Differential Equations: An Elementary Textbook for Students of Mathematics, Engineering, and the Sciences*. Dover Books on Mathematics. Dover Publications, 1963.

[23] D.G. Zill. *Differential Equations With Computer Lab Experiments*. Brooks/Cole, 1998.

Session IV: Simulation Methods

Using Artificial States in Modeling Dynamic Systems: Turning Malpractice into Good Practice

Dirk Zimmer

German Aerospace Center (DLR), Institute of System Dynamics and Control, Germany
dirk.zimmer@dlr.de

Abstract

This paper analyzes the current use of artificial states in modeling practice and proposes a new form of equations for the purpose of modeling dynamic systems. These balance dynamics equations are used to formulate dynamic processes that help to find the solution of non-linear systems of equations.

Keywords: artificial states, continuation methods, language design.

1. Introduction

Any kind of formal modeling involves abstraction. The modeler has to study the given system and decide which parts are relevant and which are not. Typically a system contains many dynamic processes where only a small subset is of interest. For instance, in rigid body dynamics, the modeler chooses to ignore the elasticity of the applied material. In power-electronics with ideal switches, the modeler chooses to ignore the complicated switching behavior.

In an equation-based modeling language, the modeler will then provide equations for both parts. The dynamic processes that are regarded as relevant will be represented by differential equations. For other processes idealizations are provided in form of algebraic equation systems. Optimally, the resulting set of differential-algebraic equations has a set of state variables that precisely matches the dynamics of interest. In real modeling practice, this is infeasible for many cases.

In many applications, the modeler is forced to extend the dynamics of the system significantly beyond his area of interest. The reason for this aggravation is that otherwise the systems of non-linear algebraic equations resulting from the idealization of dynamic processes get too complex to be reliably solved by a general simulation engine. In order to avoid this, the modeler counteracts by including more state-variables in his system than he actually intends and thereby breaking the algebraic equation systems down. Consequently, these state variables are denoted as artificial since the dynamics of them are actually of no interest. They have been artificially introduced in order to enable a better computational realization of the simulation code.

This method of artificial states represents common modeling practice. It is applied in many different ways and comes along in many disguises. In mechanics, rigid detents get replaced by stiff spring-damper constructs. In electrics, micro capacitances or leakage currents are used without original intent. In bondgraphs, small-valued C or I elements are being added. And in this paper, we present two further examples that belong to the domains of thermodynamics and microeconomics.

Although the use of artificial states is common practice, it is not regarded as good practice. Instead it is often denounced as malpractice or as method of last resort that shall only be applied if all other potential remedies have failed. This is because of the significant disadvantages this method typically incorporates.

Since the artificial states mostly express dynamic processes whose time scale is orders of magnitudes lower than the time scale of actual interest, the system becomes very stiff. This requires the use of complex ODE-solvers for stiff systems, reduces simulation speed, and often prevents real-time capability of the simulation code. Furthermore, modeling the processes attached to artificial states requires parameters that are mostly of no interest or that cannot be retrieved in a meaningful way. This results in so-called fudge parameters whose values are arbitrarily stipulated but not based on any real data. Instead, the determination of these parameter values represents mostly a trade-off between the unwanted degree of stiffness and the unwanted loss of precision: a true choice between the devil and the deep blue sea.

Hence it is easy to understand why the use of artificial states seems strongly objectionable. The more rewarding question is to ask why this method is still being so frequently applied and why the recent progress in general M&S frameworks has not eradicated the need for this method. Why do modelers use a method from that they know it is bad? What forces them to use a method of last resort? And what is to say about all the other resorts?

This paper examines these questions and it will show that the method of artificial states is not bad per se. It is actually quite clever, a smart thing to do in many occasions, and, when conducted carefully, provides valuable insight into the modeled system. What is wrong

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, 19 April, 2013, University of Nottingham, UK.

EOOLT 2013 website:
http://www.eoolt.org/2013/

about it is the way modelers are forced to apply this method in today´s M&S frameworks. Hence, we will suggest new constructs for modeling languages and new computational processing schemes for simulation engines. With these new tools at hand, the malpractice of artificial states will be turned into good practice.

But first let us look at some examples to expose the current dilemma.

2. Using Artificial States in Modeling Practice

In this section, we demonstrate the practical use of artificial states by the means of two examples. Both are realistic examples in the sense that they demonstrate the kind of problems that a modeler is typically confronted with in equation-based modeling languages. Both examples demonstrate the problems that forced the modeler to use artificial states although being initially reluctant.

2.1 Example 1: Energy Market Model

In the first example, principles from microeconomics are used for the management of energy flows [10]. The idea is the following: based on a market price each generator produces a certain amount of power and each load consumes a certain amount of power. The corresponding cost curves of generators and consumers are continuous monotonic increasing (Figure 1). The market price is then simple determined as the intersection between the two cost curves (Figure 2) for generators and consumers. In this way, a market model can be used to compute the power flow in an energy network.

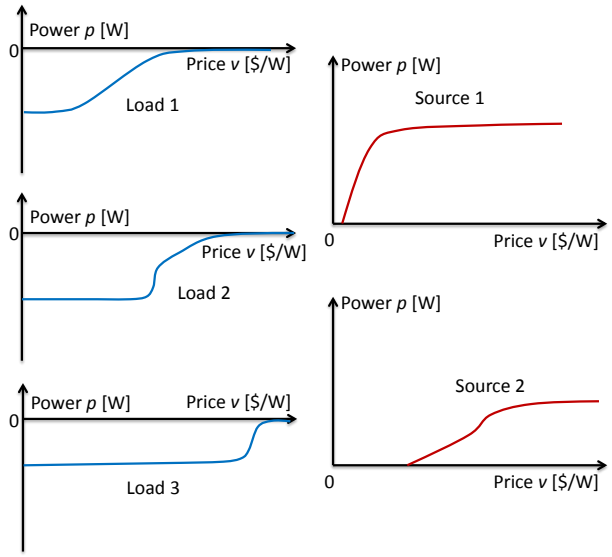


Figure 1: Cost curves

So far, so simple – but when we approach more sophisticated applications, things become a bit more difficult. Figure 3 presents the model diagram of a combined power generator whose outputs are electric and thermal energy. Up to 60% of the thermal energy can be converted into electricity.

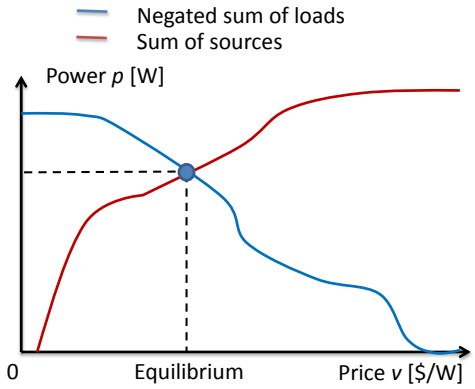


Figure 2: Equilibrium price

To this end, the power is split from the source (red component) into two sub-markets by the fixed split: 40% - 60%. Connected to this market are the consumers (blue). The thermal market, however, can take energy from the electric market but not vice versa. This is modeled by a one-way component that acts like an electric diode. The energy needs to be converted before reaching its consumer, hence the conversion element. The thermal energy can be wasted if inevitable, hence the waste element.

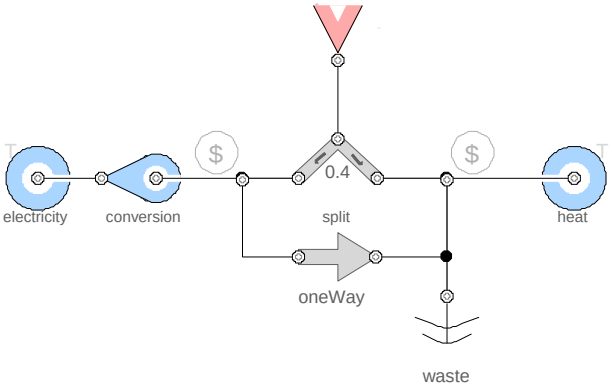


Figure 3: Model diagram of a combined power generator

The problem we get here is that we have two different market prices: one for electric energy and one for thermal energy. However these markets are not independent but coupled by algebraic equations. For instance, the model of the split component states that price at the generator is the weighted mean of the two consumer prices.

$$v_{in} = v_{out1} \cdot R + v_{out2} \cdot (1\text{-}R)$$

Hence we have a non-linear system with two iteration variables, namely the two prices of the electric and thermal market. This is certainly not exceptional and poses often no problems at all. However, in this particular case, it does. The cost-curves for the generator, the consumers, and the one-way limiter as well as the waste element all contain very flat and very steep gradients. This makes iterative, gradient-based solvers (such as Newton's method) difficult to apply since the convergence area is often very small. Finding the initial

solution requires a very good guess and steps of time-integration have to be small in order to stay within the area of convergence.

In order to approach a market solution in a more robust way, we provide a price controller. With this element, it is possible to find the solution in robust way by approaching steady state. Instead of having to determine the market price v directly such that the balance equation of power

$$p_1 + p_2 + p_3 = 0$$

holds, we make the more relaxed statement:

$$p_1 + p_2 + p_3 + p_c = 0$$

and control the price v by the lack or excess of power represented in p_c.

The corresponding controller is a very simple model that introduces an artificial state. It may compensate for any lack or excess of power p_c. The controller increases the market price in case of a power outflow ($p_c > 0$) due to a lack of power and decreases the price in case of a power inflow ($p_c < 0$) due to excess of power.

$$dv/dt \cdot T = p_c$$

where T is an arbitrary time constant. In the diagram of Figure 3, it is depicted as grey "$" placed in a circle. We can use such a price controller, because we know that the cost-functions are monotonic increasing. Any price advance will lower the demand and increase the provision of power and vice versa. This knowledge is not available to a non-linear solver but can be incorporated into the model in this way.

The incorporated disadvantages are a stiff system and that the simulation results are polluted by the dynamics of the price controller.

2.2 Exampe 2: Environmental Control System

The second example represents the modeling of a three-wheel bootstrap circuit from the environmental control system of classic aircraft architectures [7]. Here, air that is tapped from the aircrafts turbine (bleed air) is used to pressurize the cabin. Since the bleed air is hot (ca. 220°C) and at high pressure (ca. 2.5bar) [6], it needs to be cooled down and expanded before it enters the cabin. The idea is to use the energy gained in this expansion process to power a compressor and a fan for the ram air channel that is being used as cooling element. With those two devices joining the drive shaft of the expansion turbine, a more efficient cooling device can be designed.

Let us trace the path of the bleed air in the corresponding model diagram of Figure 4. The bleed air first passes the primary heat exchanger (PHX) for cooling and is then compressed before passing the main heat exchanger (MHX). Before entering the turbine for expansion, the water content needs to be extracted. Hence the bleed air passes a condenser and later on a reheater. These are both heat exchangers where the bleed air is actually interacting with itself at different stages in the circuit. Finally, after expansion, the air is sent to the mixer where it is being used to pressurize the cabin.

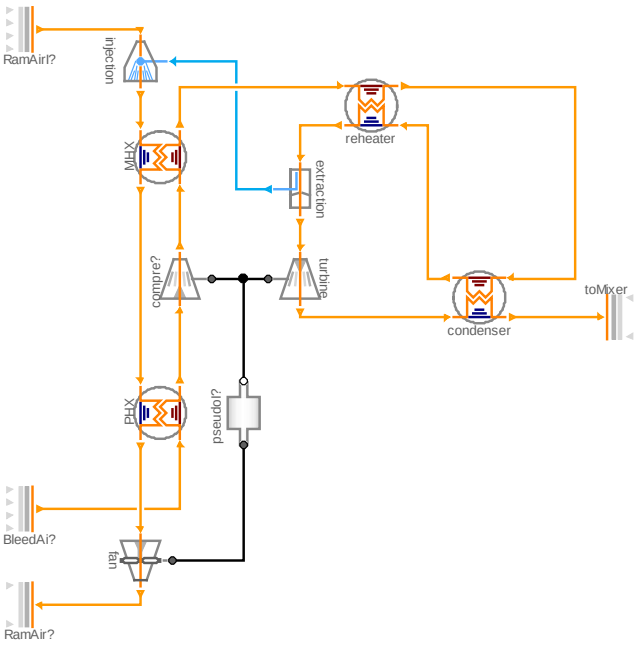


Figure 4: Model diagram of an environmental control system

In this model, we are only interested in the equilibrium point and not in any dynamics of the system at all. In the equilibrium point, the energy consumed by compressor and fan will balance the energy gain of the turbine. Furthermore all losses and gains of thermal energy in the heat exchangers cancel each other out. In the model, this equilibrium point is described by a set of pure algebraic equations. Due to the nature of thermal processes many of these equations are non-linear. The connections between the components in Figure 4 form many loops. This indicates that many of these algebraic equations are tightly coupled with each other[1]. And indeed when we have implemented the model in the modeling language Modelica, there results a very difficult non-linear system with more than 200 equations. Corresponding M&S frameworks like Dymola [3] are able to compress the system but even then a non-linear system remains with more than 40 iteration variables.

Solving such a complex system of equations in a robust manner is a very difficult task. But even when possible, a large system with more than 40 iteration variables significantly slows down the simulation engine as soon as the ECS becomes part of other dynamic processes.

For these reasons, artificial states have been used to tear the algebraic equations system apart. In total 5 state variables were sufficient to break down the non-linear equation system into individual non-linear equations that can be solved one after another.

[1] more technically: they represent a large block in the block lower triangular form of the equation system.

One of the state variables represents the velocity of the drive-shaft. A small inertia has been assigned to this shaft and hence any difference between turbine and compressor power does not need to be immediately balanced. Instead the difference can be used to accelerate or decelerate the drive shaft, as this happens in reality too. The precise value of the inertia I is not important here since we are not interested in the corresponding dynamics.

The inertia of the drive shaft introduces the following differential equation:

$$\tau = der(\omega) \cdot I$$

The variable ω represents the angular velocity of the drive shaft and is now an artificial state of the system. Its product with the torque τ determines the lack or excess of power that is (de-)accelerating the drive shaft.

The other four states are not mechanical inertias but thermal inertias. Although the physical domain is different, the applied methodology is identical.

By using artificial states, the model can be solved robustly and is open for further extension as for instance its inclusion into a complete aircraft energy system model. The amount of stiffness that is added to the system depends on the fudge parameters. However, for many practical applications, solving the stiff system is still faster than solving the original system simply because there is no complicated non-linear system with over 40 iteration variables to be solved.

3. Review of the method of artificial states

Let us review the methodology that can be extracted from the two examples. In both cases, the modeler generated a non-linear system of equations that turned out to be very difficult to solve. It is inappropriate to blame the numerical solvers for this. Without any further information no one can guarantee that any potential solver will find the correct solution[2]. Demanding for a better solver method to solve all of your problems is a pie in the sky.

It is important to understand that these non-linear system of equations result from a process of idealization. In example 1, we requested for a balance between power consumption and generation. The price had to be determined in such a way that the balance is met. Closer to reality is to regard the price determined by continuously ongoing negotiation. A lack of power leads to a higher prices and an excess of demand leads to lower prices. The balance equation simply idealizes this negotiation process by reducing it to an instant and letting it take immediate effect.

Also in example 2, balance equations are a source of idealization. The balance of power along the drive shaft ignores the inertia of the shaft and that it takes time to establish this balance. In many, many cases non-linear systems of equations result from the idealization of such balance dynamics.

[2] Presuming that there is exactly one solution or that there are multiple solutions of which any of them can be regarded as correct.

What happens now is particularly interesting. After the modeler has realized that he has gone too far and that his idealizations have created non-linear systems too difficult to solve, he reverts some of his idealization against his original intent. In example 1, the continuous process of negotiation has been reintroduced by a price controller. In example 2, mechanic and thermal inertia have been added to the system although the corresponding dynamics are of no interest.

The modeler understands that the system cannot be solved without some background knowledge that is inaccessible to the solver. It is inaccessible because it got lost in the process of idealization. For instance, the modeler knows the effects of price advance and price reduction and how to use that knowledge to derive a market solution. He also knows that inertias in physical systems help to balance the system.

But how can a modeler convey such valuable background knowledge into a general M&S framework? He sees no other way than to introduce artificial dynamics in his system and hence the method of artificial states becomes the weapon of choice. In this way, he abuses the time-integration of the simulator as a solver for his non-linear systems of equations.

When using artificial states, the modeler evidently makes a distinction between

- Dynamic processes that are relevant of the system under study.
- Dynamic processes that describe how to solve a non-linear system of equations.

Once we have become aware of this distinction, the problematic point about the use of artificial states becomes evident: The modeler makes this distinction but the M&S framework does not. It is not the modeler who wants to mix up things. He is forced to mix up things because the M&S frameworks do not provide adequate means to make a proper distinction between these two descriptions of dynamic processes.

The aim of a good modeling language should be to grasp the modeler's knowledge in a formal, clear and unambiguous way. So when the modeler knows which dynamics lead to the solution of a non-linear system of equations, any modeling language should encourage him to include this knowledge into his models in a proper form. After all, this represents valuable knowledge that can only be beneficial for the subsequent processes of code generation and simulation.

Hence the next chapter suggests a way, how such knowledge can be conveniently incorporated in a modeling language. It turns out to be surprisingly simple and intuitive.

4. Balance dynamics equations: Turning implicit idealization into explicit idealization

In the previous section, we stated that the idealization of balance dynamics is a very frequent source of non-linear system of equations. Let us therefore review the equations of the price controller from Example 1 that represent exactly one such example. First we had the desired ideal form for the balance of power flows:

$$p_1 + p_2 + p_3 = 0$$

In this case, the market price v has to be determined by a non-linear system of equations. Because of this, we relaxed the balance equation by introducing p_c and determined (or controlled) the price by means of a differential equation:

$$p_1 + p_2 + p_3 + p_c = 0$$
$$dv/dt \cdot T = p_c$$

We can derive the ideal form out of the balance dynamics by assuming a steady-state scenario and setting the derivative to zero. Furthermore, we assume that this steady state is continuously maintained and hence that the corresponding balance dynamics take instantaneous effect. This is like stating that the time constant is approaching zero.

This helps to understand the process of idealization and we can see that there are two major implications behind this idealization process:

1. The balance dynamics take place in no time; so they are regarded as infinitely fast.
2. The balance dynamics finally reach a stable steady state.

Since this pattern of idealization is so common in so many applications, it seems meaningful to enable its explicit formulation in a modeling language. To this end, a simple operator suffices.

Most modeling languages feature an operator for the time-derivative such as:

$$der(v) \cdot T = p_c$$

In strong resemblance to this operator, we can define and use a balance operator instead:

$$balance(v) = p_c$$

The operator balance(v) simply replaces the term der(v) · T and implies the two idealizations der(v) = 0 and T → 0. The fudge parameter T has consequently gone lost.

With this operator we have introduced a new kind of equation. We call them balance dynamics equations. They enable us to state the implicit assumption of the idealized algebraic equation in explicit form. In this way, you get the best of both worlds: you can interpret them as algebraic constraints in the simulation context but you can also interpret them as dynamic process in the solver context. How to precisely do that is content of the next section.

5. Handling balance dynamics equations in a simulation environment

In the following small example, we find an algebraic equation, a differential equation, and a balance dynamics equation:

$$der(x) = z$$
$$balance(y) = p(y)-x$$
$$z = \sin(y)$$

The balance equation now states two things: a non-linear equations system (0 = p(y)-x) and a dynamic process how to solve this system (der(y) = p(y)-x) based on the modeler's knowledge that p is a monotonic increasing function.

Consequently, these model equations can now be transformed in two different ways:

$der(x) = z$	$x = const$
$0 = p(y)-x$	$der(y) = p(y)-x$
$z = \sin(y)$	$z = \sin(y)$

The left version represents the *simulation dynamics*, the dynamics of relevance for simulation; the right version represents *the solver dynamics*, the dynamics needed to solve the non-linear system of equations. This solver dynamics is formulated as sub-simulation (a simulation nested within the main simulation) hence the states of the actual simulation (here: x) are held constant. There remains the question how to take use of such a sub-simulation.

The idea is of course that in case we fail to solve the non-linear equation (here: 0 = p(y)-x) directly, we use the sub-simulation on the differential equation (here: der(y) = p(y) − x) to get to the area of local convergence. But before we pursue this idea any further, let us highlight another benefit of balance dynamics equations. Balance dynamics equations do not only help solving non-linear equations by getting to the area of local convergence but they also provide information that helps to solve the actual system more efficiently once you are in this area.

Whenever an iterative numerical solver is applied to a non-linear system of equations, we need to determine a set of suitable iteration variables. These iteration variables are also often denoted as tearing variables since they are used to tear the algebraic loops apart and generate residual values instead. Choosing such iteration variables is a difficult task where many constraints have to be regarded [11]. This led to the choice of over 40 iteration variables for the ECS system in example 2.

Balance dynamics equations provide an excellent indication which variables to choose as iteration variables. Since they are assigned to a state-variable in the corresponding sub-simulation, this state variable must also be a suitable iteration variable. In this way, the number of iteration variables (and thereby the size) can be significantly reduced. For instance, in Example 2, the number of iteration variables can be reduced from over 40 to 5, leading to a much more efficient simulation.

This coincidence of iteration variables for the direct solution with the state variables for sub-simulation

indicates that these two tasks are actually closely related. Remember, the balance equation

$$\text{balance}(x) = f(x)$$

offers us two different ways to get to a solution. Either we solve the equation $0 = f(x)$ directly or we approach the steady state by a sub-simulation on the differential equation $\text{der}(x) = f(x)$. On the first look, this looks like two separate tasks. However, let us analyze how we would perform such a sub-simulation in practice.

After all, this is a special case: we do not perform a usual simulation; we want to perform a simulation for the sole purpose to approach the steady state with the time[3] $t \to \infty$. Which integration method would we choose for that?

Since the differential equation $\text{der}(x) = f(x)$ is supposed to describe a stable system, an implicit method is a strong favorite. Any explicit integration method would be limited in its step-size in order to maintain stability and approaching infinity with steps of finite width is an unpromising endeavor.

Since we do not care about the precise trajectory leading to the steady state and since the steady state solution itself is insensitive to the local integration error, there is no reason to choose any higher-order method. Order 1 is completely sufficient.

So our method of choice would be to perform Backward Euler with as large steps as possible. Hence, let us look at one integration step of this method going from t to $t+h$:

$$x_{t+h} = x_t + h \cdot \text{der}(x_{t+h})$$

or in our current example:

$$x_{t+h} = x_t + h \cdot f(x_{t+h})$$

Being an implicit method, we have to solve the system of equations: $0 = g(x_{t+h})$ with $g(x_{t+h})$ being defined as:

$$g(x_{t+h}) = x_t - x_{t+h} + h \cdot f(x_{t+h})$$

Evidently for $h \to \infty$, solving $g(x_{t+h})$ becomes equivalent to solving $f(x)$ directly. Now it becomes clear how the solver dynamics can support us to find the solution of $f(x)$. Instead of solving the system $f(x)$, we can solve $g(x)$ and in this way, we have won one important degree of freedom: we can choose h.

In this way, we have transformed the problem into a numerical continuation problem [1]. In general, a continuation problem results from transforming a function $F(x)$ to $F'(x,\lambda)$ with $\lambda \in [0,1]$ where $F'(x,1) = F(x)$ and $F'(x,0)$ is easy to solve. Many solutions methods have been developed for this kind of problem and they are already applied by many M&S Frameworks, mostly to solve initialization problems in a more robust way [9] for instance by using homotopy [8].

[3] Please note, the time t does not represent the main simulation time here but the time of the nested sub-simulation. The complete time-span of the sub-simulation represents only one instant in the main simulation.

To use numerical continuation solvers not only for initialization problems but also during simulation is also not a completely new idea. Artificial time integration is not uncommon to find solutions for PDEs [2]. The main difference to classic continuation problem in our case is that is not bounded by 1 but is free to go to infinity. Hence we have to adapt the continuation solver. The following paragraph sketches an algorithm that is a variant of the simplest kind of numerical continuation: the natural parameter continuation where h is our continuation parameter.

In case h is too large and our guess value for x_{t+h} is outside the convergence area, we can choose h small enough to be located in the convergence area again. And with each solution of $g(x)$, we step a little closer to the final solution of $f(x)$. In this way, we have found a robust way to solve our non-linear system of equations. Figure 5 depicts the corresponding algorithm of the balance dynamics solver.

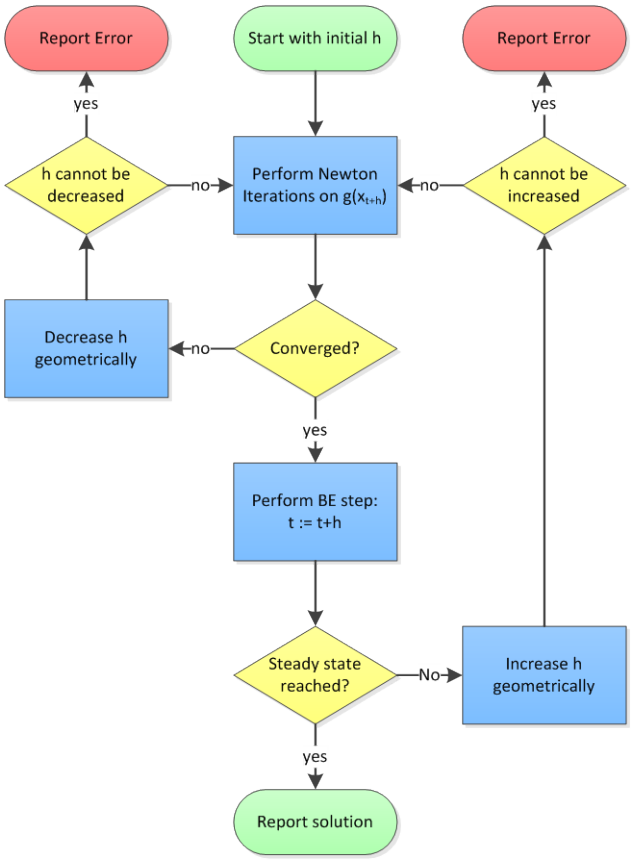


Figure 5: Algorithm for the balance dynamics solver

This algorithm becomes part of the main simulation loop and replaces the former direct solver for $0 = f(x)$. It is hence performed at each integration step of the main simulation task. It is not necessarily slower than the direct solver for $f(x)$. Having a high initial value for the sub-simulation step-size h and a good guess value for x_{t+h}, not many more iterations would be required than for a direct solution of $0 = f(x)$. A call to the direct solver is thus not required.

The difference occurs when good guess values for x_{t+h} are not available. In a normal setup, the integration step-size of the main simulation loop would be reduced in order to reobtain a good guess. Using our balance dynamics solver, this is unlikely to be necessary. More iterations would be needed in the solver to get the solution but the step-size of the main simulation loop can be maintained. And of course, finding the initial solution is also much simpler.

Without balance dynamics equations, the modeler has the choice of either creating a stiff system or a difficult non-linear system of equations. In both cases, he imposes severe limitations on the main integration step size and thereby creates a global damage even when only a small subsystem is actually concerned. With balance dynamics equations and a corresponding solver, the damage is kept local.

A final remark with respect to the algorithm in Figure 5: please note that the step-size control of h is not equivalent to classic step-size control in ODE solvers. It is based solely on the matter of convergence not on the matter of local integration error and hence can be performed much more aggressively.

6. Small application example

To prove the feasibility of this approach, we provide a small example. The following DAE

$$\begin{aligned} dx/dt &= y \\ dy/dt &= -0.1 \cdot a - 0.4 \cdot y \\ s(a) &= 10 \cdot x \end{aligned}$$

requires the solution of an expression containing the non-linear function s(a) displayed in Figure 6:

$$s(a) = \begin{cases} a/4 - 3/4 & \text{if } a < -1 \text{ then} \\ a/4 + 3/4 & \text{else if } a > 1 \text{ then} \\ a & \text{else} \end{cases}$$

with its derivative to be defined as

$$s(a) = \begin{cases} 1/4 & \text{if } a < -1 \text{ then} \\ 1/4 & \text{else if } a > 1 \text{ then} \\ 1 & \text{else} \end{cases}$$

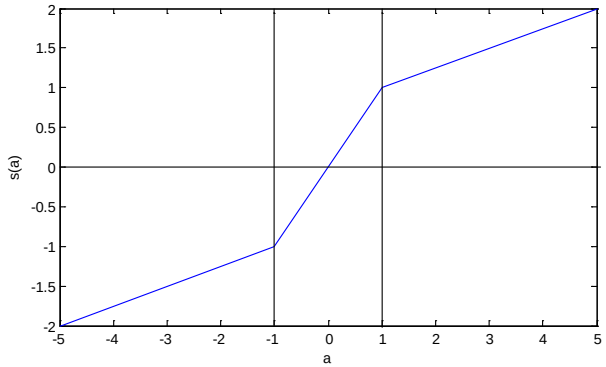


Figure 6: The piecewise linear function s(a)

The convergence area of solving s(a)=0 with respect to Newton's method is exactly [-1,1]. Although the convergence area is strictly limited, the solution can easily be found if one knows that s(a) is strictly monotonic increasing. We can incorporate this knowledge in form of a balance dynamics equation:

$$\begin{aligned} dx/dt &= y \\ dy/dt &= -0.1 \cdot a - 0.4 \cdot y \\ balance(a) &= 10 \cdot x - s(a) \end{aligned}$$

This DAE is now transformed into two forms for numerical ODE solvers.

- For the main ODE solver:

$$\begin{aligned} dx/dt &= y \\ dy/dt &= -0.1 \cdot a - 0.4 \cdot y \\ 0 &= 10 \cdot x - s(a) \end{aligned}$$

- For the continuation solver:

$$\begin{aligned} x &= const \\ der(a) &= 10 \cdot x - s(a) \end{aligned}$$

The main simulation is performed with Forward Euler and a step width of 0.1s for 100s. Without the continuation solver, the non-linear system of equations cannot be solved when the state variable x enters the range of [-0.1,0.1]. A step-size of smaller than 0.01s has to be taken in order to practically ensure the solvability of the system.

The continuation solver has been realized according to the algorithm sketched in Figure 5 and can robustly solve this system of equations. Figure 7 shows the simulation result. The step-size of the main-simulation loop is not impaired by the non-linear system anymore.

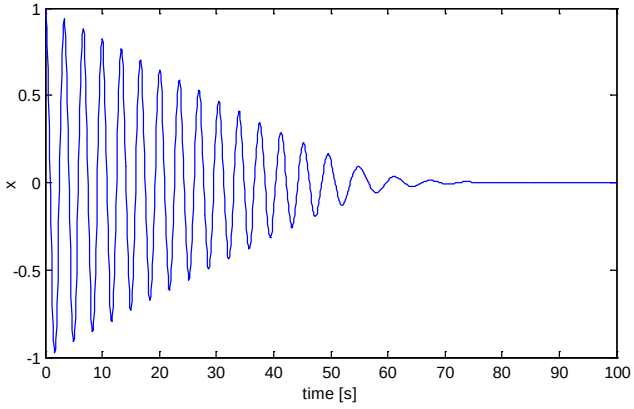


Figure 7: Simulation result showing the state x

Of course, this is a very small and simple example but it demonstrates that the basic idea works. For more mature implementation, we need to examine a number of interesting questions:

- How to optimally control the step-width h of the sub-simulation?
- How to provide suitable initial guesses of this step-width?
- When to stop sub-simulation when convergence is not reached?
- How should the step-size control of the main-simulation loop be controlled w.r.t to the convergence speed of the continuation solver?
- etc…

In this test-implementation, for instance, the initial step width h was chosen to be three times the minimum step-width that was required for the solution of the last step from the main simulation. This ensures that the initial value of h is relatively close to a prior successful continuation step while enabling a geometric increase of h for the case the continuation solver is actually not needed.

Figure 8 plots the number of times the function s(a) is being evaluated by the continuation solver in order to compute a new step or to check for convergence. Each value in the plot represents one time-step of the main simulation loop.

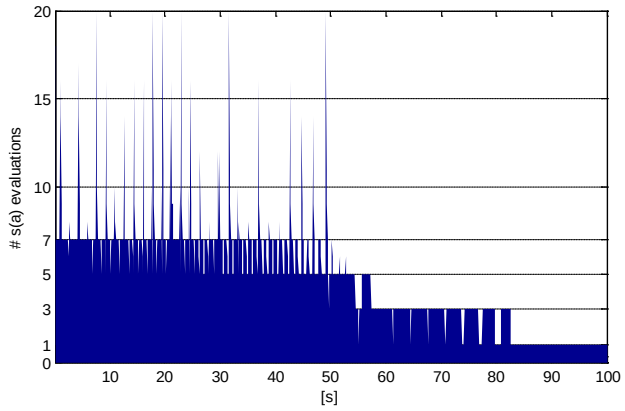


Figure 8: Number of function evaluations

During the first half of the simulation, the system oscillates with high amplitude. There are several peaks caused by the continuation method when several sub-steps had to be taken before convergence could be reached. This is where a direct solution with a gradient-based solver would have failed.

In the second half, there is no need for the continuation method anymore and a large initial step-size and an increasing quality of the guess value significantly reduce the computational effort. With the exception for one extra evaluation to check for convergence, the continuation solver hardly generates any additional burden anymore.

Even this rudimentary test implementation shows that a continuation solver is affordable for each time-step of the main simulation while the robustness of the solution can be improved. Solving at certain points might be expensive but when not needed the overhead is small. Because of the robust solution method, a large step-size of the main simulation can be afforded.

7. Prospective Limits of this Solution Method

The proposed variant for the natural parameter continuation is of course very simple and may not be able to solve all forms of balance dynamics. More elaborated continuation solvers support to deal with more complex continuation paths such as bifurcations of turning points. (There are complete libraries for continuation solvers such PyCont [5]). However, the need for such complex solvers is a warning and we shall rather question ourselves about the origin of this need.

After all, balance dynamics should be simple but practically oriented modelers will tell us that they can become pretty complex even stating self-contradictory sentences such as: “the value of fudge parameters is very significant”. Evidently, “fudge parameter tuning” can become an obsession. Why does this happen?

One important point is that in many complex applications, balance dynamics are layered. For instance, in the model of an environmental control system, there are balance dynamics resulting from physical inertia. There might also be balance dynamics resulting from sub-controllers. The modeler is actually interested in none of these processes but for the working of the sub-controller, it is important that the physical balance dynamics take place in a shorter time span. The resulting layering is illustrated in Figure 9:

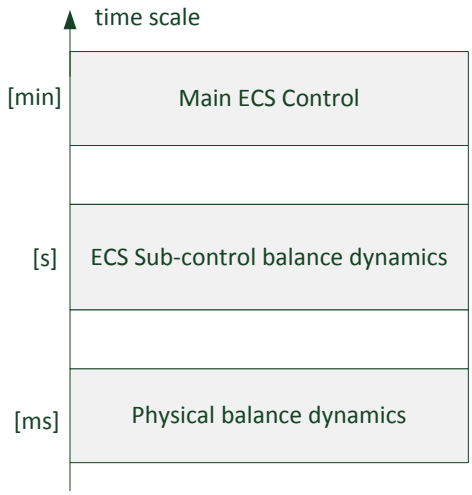


Figure 9: Layering of different balance dynamics

In classic modeling, the modeler is now using the fudge parameters to impose this layering. During this process, he typically makes a trade-off: on the one hand, he wants to separate the different balance dynamics and keep them in right order. On the other hand, he wants to reduce the stiffness of the overall system. Optimizing this trade-off is what is typically described as “fudge parameter tuning”.

Having this idea in mind, it seems now smart that instead of having one continuation solver to solve complex balance dynamics, we prefer nested continuation solvers, each of them solving one layer of simple balance dynamics. This requires that the modeler has means to separate different balance dynamics and to layer them. The currently proposed operator does not offer a sufficient solution for this.

8. Realization within a Modeling Language

The proposal as presented here is of course very easy to realize in most equation-based modeling languages. It is sufficient to add one single operator for the formulation of balance equations just as this had been done for the homotopy operator [8] in Modelica [4].

However, as temptingly simple as this seems, the balance operator as presented here will prove to be only partly sufficient. There are two major flaws involved with this solution:

1. The last section outlined the need to layer balance dynamics and to nest the corresponding continuation solvers. This is not possible with this operator notation.
2. Having available only this operator, it is not possible to reuse existing models (or components) of dynamic processes formulated with derivatives for the balance dynamics. The modeler is forced to remodel all relevant equations using the balance operator instead.

The second point of critique is valid also for the homotopy operator in Modelica. Typically, a modeler first builds a stiff system that includes the balance (or initialization) dynamics and then, in a second stage, he separates the two dynamics from each other. It would be favorable if the modeling of the second stage could reuse the components of the first stage.

For these reasons, we will finally need a better solution than the proposed operator but we can regard this proposal as intermediate solution in order to conduct the heavily needed research on this topic.

9. Conclusions

Since decades modelers use artificial states. Since decades they are being told that this is bad. Since decades they do it anyway. This is because formulating the dynamics that lead to the balance point of a sub-system is often the only way a modeler can explain how to solve his non-linear system of equations. It is unfortunate that M&S frameworks have not recognized this and provided better means for the modeler that enable him to distinguish between simulation dynamics and solver dynamics. This would prevent the rightfully criticized abuse of simulation dynamics to solve non-linear systems of equations.

For this purpose, we have proposed the concept of balance dynamics equations. It turns out that adding a simple operator is at least partly sufficient and a first step to explore the concept further. Balance dynamics equations can then be used to create code for a corresponding solver that is much more robust. Furthermore, the information contained in them can be used to make a better choice of iteration (or tearing) variables. Unwanted stiffness can be avoided and the integration step size of the main-integration loop is not needlessly limited. A difficult non-linear system of equations in a subcomponent will still increase the computational burden but the damage can be kept local.

This work so far is essentially based on theoretical thoughts and analysis of modeling experience. It needs to be put into practice and properly tested. Also balance dynamics equations do not provide never-ending salvation. They won't solve all modeling problems, but they have the potential to solve a big chunk of them. We hope for the future that this or similar methodologies are adapted by M&S Frameworks of industrial maturity.

Acknowledgements

I would like to thank Andreas Pfeiffer, Martin Otter and Michael Sielemann from DLR for suggesting several improvements.

References

[1] Eugene L. Allgower and Kurt Georg. *Introduction to Numerical Continuation Methods*, SIAM Classics in Applied Mathematics 45. 2003.

[2] U. Ascher, H. Huang, and K. van den Doel. Artificial Time Integration. *BIT Numerical Mathematics*, 47(1): 3-25, 2007.

[3] Dymola: available at www.dymola.com

[4] The Modelica Association. *Modelica® A Unified Object-Oriented Language for Systems Modeling - Language Specification Version 3.3*, Available at www.modelica.org, 2012

[5] PyCont available at: www2.gsu.edu/~matrhc/PyCont.html

[6] Rolls Royce. *The Jet Engine*. Rolls Royce Plc. Derby England. 278p. 1996.

[7] M. Sielemann, T. Giese, B. Oehler, M. Gräber, Optimization of an Unconventional Environmental Control System Architecture. In: *SAE International Journal of Aerospace*, 4(2):1263-1275. 2011

[8] M. Sielemann et. al., Robust Initialization of Differential-Algebraic Equations Using Homotopy. In: *Proceedings of 8th International Modelica Conference*. Dresden, Germany, 2011

[9] M. Sielemann and G. Schmitz, A quantitative metric for robustness of nonlinear algebraic equation solvers. In: *Mathematics and Computers in Simulation*, 81 (12), pp 2673-2687. Elsevier, 2011.

[10] D. Zimmer and D. Schlabe, Implementation of a Modelica Library for Energy Management based on Economic Models. *Proceedings of the 9th International Modelica Conference* , Munich, Germany (2012)

[11] D. Zimmer, *Equation-Based Modeling of Variable Structure Systems*. PhD Thesis, ETH Zürich, 219 p. 2010

Biography

Dr. Dirk Zimmer received his PhD degree from the Department of Computer Science at the Swiss Federal Institute of Technology (ETH Zurich). He is currently pursuing his research work at the Institute of System Dynamics and Control belonging to the German Aerospace Center (DLR). Also, he is lecturer at the Institute of Computer Science at the Technical University of Munich (TUM).

Simplification of Differential Algebraic Equations by the Projection Method[1]

Elena Shmoylova[2] Jürgen Gerhard[2] Erik Postma[2] Austin Roche[2]

[2]Maplesoft, Canada, {eshmoylova, jgerhard, epostma, aroche}@maplesoft.com

Abstract

Reduction of a differential algebraic equation (DAE) system to an ordinary differential equation system (ODE) is an important step in solving the DAE numerically. When the ODE is obtained, an ODE solution technique can be used to obtain the final solution. In this paper we consider combining index reduction with projection onto the constraint manifold. We show that the reduction benefits from the projection for DAEs of certain form. We demonstrate that one of the applications where DAEs of this form appear is optimization under constraints. We emphasize the importance of optimization problems in physical systems and provide an example application of the projection method to an electric circuit formulated as an optimization problem where Kirchhoff's laws are acting as constraints.

Keywords differential algebraic equations, index reduction, projection method

1. Introduction

The idea of using projection in the treatment of DAEs stems from the projection method introduced by Scott [16] as an alternative to Lagrange's equations for obtaining equations of motion of a constrained mechanical system. Scott proposed to consider constraints as a manifold and to project the equations of unconstrained motion onto the space tangent to the manifold. The idea came from intuitive observations of a mechanical system subject to holonomic constraints and was later justified by Blajer [6]. Blajer [7] and Arczewski and Blajer [1] extended the applications of the projection method to mechanical systems subject to affine linear nonholonomic constraints. From Arczewski and Blajer's [1] derivations it can be seen that the projection method is not an alternative to Lagrange's equations but rather a technique applied to the Lagrange equations in order to simplify them.

To describe the behavior of a constrained mechanical system correctly, the equations of the unconstrained system and the constraints must be combined into one system of equations that takes into account the contributions of the forces induced by the constraints. One of the possible ways to achieve this goal is to introduce the Lagrange multipliers as it is done in Lagrange's equations. The system of DAEs obtained by introducing the Lagrange multipliers is the system to which the projection method is applied. Simply put, the projection method is a method to eliminate the necessity of computing the Lagrange multipliers and to satisfy the constraints in one simple action – projection onto the constraint manifold. At the same time, it is still possible to find the Lagrange multipliers after the solution is obtained [1], if necessary.

From [1] we see that the projection method consists of the index reduction performed by differentiation followed by projection. We want to investigate what we can gain in the case of an arbitrary DAE by adding projection to the index reduction.

Consider a DAE system in a general form given by

$$\mathbf{f}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t), \mathbf{z}(t)) = 0, \tag{1}$$

where $\mathbf{f} = (f_1, \ldots, f_{n+k})^T$, the superscript T denotes the transpose, t is time, $\mathbf{x}(t) = (x_1(t), \ldots, x_n(t))^T$ is called the vector of differential variables, due to the presence of its derivatives $\dot{\mathbf{x}}(t) = (dx_1(t)/dt, \ldots, dx_n(t)/dt)^T$, and $\mathbf{z}(t) = (z_1(t), \ldots z_k(t))^T$ is called the vector of algebraic variables.

System (1) can be reduced to a system of ODEs by differentiating with respect to t. The number of differentiations needed to obtain equations for $\dot{\mathbf{z}}(t)$ is called the *index* of system (1) [2]. The higher the index of a DAE, the more complicated the problem of converting it to an ODE. For example, a mechanical system with holonomic constraints typically is an index-3 DAE system.

In this paper, we consider DAEs of various indices and investigate the properties of the DAEs that make simplification by projection of the reduced system possible. One application where DAEs with these properties are common is optimization under constraints.

We consider optimization problems that result in DAEs and show how addition of projection to the index reduction

[1] This work has been submitted as US Patent Application 20120179437 on July 12, 2012.

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. 19 April, 2013, University of Nottingham, UK.

EOOLT 2013 website: http://www.eoolt.org/2013/

simplifies the equations. As an example, we formulate an optimization problem for an electric circuit consisting of a resistor, an inductor, and a capacitor connected in series and demonstrate the application of the projection method.

2. Preliminaries

In what follows, to make the formulae easily readable, we do not write explicitly what variables a function depends on, unless this information is crucial for understanding the formulae. Moreover, by a linear function or equation we always mean one that is affine linear.

We want to consider the general form of a DAE and investigate how projection can assist us in the reduction of the DAE to an ODE. We want to consider DAEs of different indices. The general form (1) is not suited for this purpose, for it does not reflect the index information. Luckily, most of the problems encountered in practice can be expressed as a combination of more restrictive structures of ODEs coupled with constraints [2]. These are called Hessenberg forms and are given below.

Hessenberg Index-1

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}, \mathbf{z}), \tag{2a}$$

$$0 = \mathbf{h}(t, \mathbf{x}, \mathbf{z}), \tag{2b}$$

where $\mathbf{x} = \mathbf{x}(t) = (x_1(t), \ldots, x_n(t))^T$ is the vector of differential variables, $\mathbf{f} = (f_1, \ldots, f_n)^T$, $\mathbf{z} = \mathbf{z}(t) = (z_1(t), \ldots, z_k(t))^T$ is the vector of algebraic variables, $\mathbf{h} = (h_1, \ldots, h_k)^T$, and the Jacobian

$$\partial\mathbf{h}/\partial\mathbf{z} \text{ is assumed to be nonsingular for all } t. \tag{3}$$

Equations (2a) are differential, and equations (2b) are algebraic. If the Jacobian is nonsingular, by the implicit function theorem [12], it is possible to solve the algebraic equations (2b) for $\mathbf{z}$ uniquely in a numerical fashion.

We note that the Jacobian in condition (3) may become singular for some particular solution of the system (2). Consequently, the index of the DAE will be higher for this particular solution. Therefore, in what follows, when we speak of index, this is to be understood as the generic index meaning that the corresponding Jacobian is nonsingular for almost all t and for almost all solutions.

Hessenberg Index-2

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}, \mathbf{z}), \tag{4a}$$

$$0 = \mathbf{h}(t, \mathbf{x}), \tag{4b}$$

where $\mathbf{x}$, $\mathbf{z}$, $\mathbf{f}$, $\mathbf{h}$ are defined as above for the index-1 form (2), and the product of Jacobians

$$\frac{\partial \mathbf{h}}{\partial \mathbf{x}} \frac{\partial \mathbf{f}}{\partial \mathbf{z}} \text{ is nonsingular for all } t. \tag{5}$$

For example, a mechanical system subject only to nonholonomic constraints is a Hessenberg index-2 system, where $\mathbf{x}$ is the vector of velocities, and $\mathbf{z}$ is the vector of Lagrange multipliers.

Hessenberg Index-3

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{x}, \mathbf{y}, \mathbf{z}), \tag{6a}$$

$$\dot{\mathbf{x}} = \mathbf{g}(t, \mathbf{x}, \mathbf{y}), \tag{6b}$$

$$0 = \mathbf{h}(t, \mathbf{x}), \tag{6c}$$

where $\mathbf{y} = \mathbf{y}(t) = (y_1(t), \ldots y_m(t))^T$ and $\mathbf{x} = \mathbf{x}(t) = (x_1(t), \ldots x_n(t))^T$ are the vectors of differential variables, $\mathbf{f} = (f_1, \ldots, f_m)^T$, $\mathbf{g} = (g, \ldots, g_n)^T$, $\mathbf{z} = \mathbf{z}(t) = (z_1(t), \ldots, z_k(t))^T$ is the vector of algebraic variables, $\mathbf{h} = (h_1, \ldots, h_k)^T$, and the product of the Jacobians

$$\frac{\partial \mathbf{h}}{\partial \mathbf{x}} \frac{\partial \mathbf{g}}{\partial \mathbf{y}} \frac{\partial \mathbf{f}}{\partial \mathbf{z}} \text{ is nonsingular for all } t. \tag{7}$$

There are two different kinds of differential equations, (6a) and (6b), and a set of algebraic equations (6c). An example of a DAE system in Hessenberg index-3 form is a mechanical system subject only to holonomic constraints, whose displacements, velocities and Lagrange multipliers correspond to $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$, respectively.

Hessenberg forms of higher index are defined similarly.

Given a Hessenberg index-1 equation of the form (2), by differentiating the algebraic equations (2b) once and replacing $\dot{\mathbf{x}}$ according to equation (2a), equations (2) are reduced to ODEs

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}, \mathbf{z}), \tag{8a}$$

$$\dot{\mathbf{z}} = -\left(\frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right)^{-1} \left[\frac{\partial \mathbf{h}}{\partial t} + \frac{\partial \mathbf{h}}{\partial \mathbf{x}} \mathbf{f}(t, \mathbf{x}, \mathbf{z})\right]. \tag{8b}$$

Note that we do not solve the algebraic equations themselves. We use only the derivatives of the algebraic equations in the reduced system (8). In exact computations, we use the initial conditions to satisfy the original algebraic equations. Numerical integration, however, always involves some small errors, which can accumulate into a large error. Consequently, it may happen that a numerical solution of system (8) does not solve the original problem (2) [8]. This phenomenon is known as *constraint drift*. In order to avoid it, the integration procedure must be modified. Therefore, special methods for index-1, index-2, and even Hessenberg index-3 DAEs have been developed [2, 10]. For example, for index-1 DAEs, solving the original algebraic equations by the Newton-Raphson method could be added to every integration step.

In what follows, we take the index-1 form as the target form when performing index reduction, since most numerical integration software can easily handle index-1 DAEs.

3. Reduction of Hessenberg Index-3 DAEs and the Projection Method

As it turns out, not much simplification is possible for Hessenberg index-2 DAEs, and we omit this case here.

Consider now the Hessenberg index-3 system (6). From a geometrical point of view, the algebraic equations (6c) define a time-varying manifold, which we denote by $\mathcal{M}(t)$, in the n-dimensional vector space of $\mathbf{x}$. For $\mathbf{x}$ to stay on $\mathcal{M}(t)$ for every t, $\dot{\mathbf{x}}$ must be tangent to $\mathcal{M}(t)$ for every t. Since $\frac{\partial \mathbf{h}}{\partial \mathbf{x}} \frac{\partial \mathbf{g}}{\partial \mathbf{y}} \frac{\partial \mathbf{f}}{\partial \mathbf{z}}$ is nonsingular,

$$\mathbf{C} = \frac{\partial \mathbf{h}}{\partial \mathbf{x}}$$

is of maximal row-rank, and the rows of the matrix $\mathbf{C}$ span the space normal to $\mathcal{M}(t)$ at $\mathbf{x}(t)$ for every $\mathbf{x}$ and t.

Differentiating the algebraic equations (6c) with respect to time, we get

$$\mathbf{C}\dot{\mathbf{x}} + \frac{\partial \mathbf{h}}{\partial t} = 0. \tag{9}$$

Equation (9) means that the projection of $\dot{\mathbf{x}}$ onto the space normal to $\mathcal{M}(t)$ is determined only by the change of the manifold $\mathcal{M}(t)$ with time. If the manifold is time-invariant, i.e. the algebraic equations are time-invariant, then $\dot{\mathbf{x}}$ is tangent to $\mathcal{M}(t)$ for any t.

To describe the projection of $\dot{\mathbf{x}}$ onto the tangent space, we choose a matrix $\mathbf{D}$ such that $\mathbf{D}$ is of maximal column-rank and

$$\mathbf{CD} = 0, \tag{10}$$

i.e. the columns of $\mathbf{D}$ span the space tangent to $\mathcal{M}(t)$ at $\mathbf{x}(t)$ for every $\mathbf{x}$ and t. Therefore, for any t we have obtained two subspaces of $\mathbb{R}^n$, spanned by the rows of $\mathbf{C}$ and by the columns of $\mathbf{D}$, that are orthogonal complements of each other. By the theorem on orthogonal decomposition [11], any element in $\mathbb{R}^n$ can be represented as a sum of projections onto these subspaces. Consequently, for any t,

$$\dot{\mathbf{x}} = \mathbf{Du} + \mathbf{C}^T\mathbf{v}, \tag{11}$$

for certain vectors $\mathbf{u} \in \mathbb{R}^{n-k}$ and $\mathbf{v} \in \mathbb{R}^k$. Note that both $\mathbf{C}$ and $\mathbf{D}$ depend on $\mathbf{x}$ and t only, but not on $\mathbf{y}$ or $\mathbf{z}$.

Substituting representation (11) into equation (9), we find the tangential component of $\dot{\mathbf{x}}$ as

$$\mathbf{v} = -\left(\mathbf{CC}^T\right)^{-1}\frac{\partial \mathbf{h}}{\partial t}, \tag{12}$$

vanishing identically if $\mathbf{h}$ does not depend on t explicitly.

We start the reduction of system (6) to an index-1 system by differentiating the algebraic equations (6c), as in (9), and then use equation (6b) to obtain:

$$\mathbf{Cg} + \frac{\partial \mathbf{h}}{\partial t} = 0. \tag{13}$$

We differentiate equation (13) once more:

$$\mathbf{C}\left[\frac{\partial \mathbf{g}}{\partial t} + \frac{\partial \mathbf{g}}{\partial \mathbf{x}}\dot{\mathbf{x}} + \frac{\partial \mathbf{g}}{\partial \mathbf{y}}\dot{\mathbf{y}}\right] + \dot{\mathbf{C}}\mathbf{g} + \frac{\partial^2 \mathbf{h}}{\partial t^2} + \frac{\partial^2 \mathbf{h}}{\partial \mathbf{x}\partial t}\dot{\mathbf{x}} = 0. \tag{14}$$

Finally, using equations (6a) and (6b) in equation (14), we can re-write equations (6) as:

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{x}, \mathbf{y}, \mathbf{z}), \tag{15a}$$

$$\dot{\mathbf{x}} = \mathbf{g}(t, \mathbf{x}, \mathbf{y}), \tag{15b}$$

$$0 = \dot{\mathbf{C}}\mathbf{g} + \mathbf{C}\left[\frac{\partial \mathbf{g}}{\partial t} + \frac{\partial \mathbf{g}}{\partial \mathbf{x}}\mathbf{g} + \frac{\partial \mathbf{g}}{\partial \mathbf{y}}\mathbf{f}\right] + \frac{\partial^2 \mathbf{h}}{\partial t^2} + \frac{\partial^2 \mathbf{h}}{\partial \mathbf{x}\partial t}\mathbf{g}. \tag{15c}$$

The system (15) is an index-1 system of DAEs. In order to see this, we notice that $\mathbf{z}$ appears in the equations (15c) only inside $\mathbf{f}$, and the Jacobian in condition (3) for the equations (15c) is equal to the product of Jacobians in condition (7). Then condition (7) yields that the algebraic equations (15c) are solvable for $\mathbf{z}$.

We now want to apply the projection method to simplify the system (6) and compare the result with system (15). We substitute the representation (11) of $\dot{\mathbf{x}}$ into equation (6b):

$$\mathbf{Du} + \mathbf{C}^T\mathbf{v} = \mathbf{g}(t, \mathbf{x}, \mathbf{y}). \tag{16}$$

Instead of differentiating equation (13), we differentiate equation (16) and obtain

$$\dot{\mathbf{D}}\mathbf{u} + \mathbf{D}\dot{\mathbf{u}} + \dot{\mathbf{C}}^T\mathbf{v} + \mathbf{C}^T\dot{\mathbf{v}} = \frac{\partial \mathbf{g}}{\partial t} + \frac{\partial \mathbf{g}}{\partial \mathbf{x}}\mathbf{g} + \frac{\partial \mathbf{g}}{\partial \mathbf{y}}\mathbf{f}, \tag{17}$$

where $\mathbf{v}$ is given by equation (12). Projecting equation (17) onto the normal space by multiplying it by $\mathbf{C}$ on the left, we obtain equations for $\mathbf{z}$:

$$\mathbf{C}\dot{\mathbf{D}}\mathbf{u} + \mathbf{C}\dot{\mathbf{C}}^T\mathbf{v} + \mathbf{C}\mathbf{C}^T\dot{\mathbf{v}} = \mathbf{C}\frac{\partial \mathbf{g}}{\partial t} + \mathbf{C}\frac{\partial \mathbf{g}}{\partial \mathbf{x}}\mathbf{g} + \mathbf{C}\frac{\partial \mathbf{g}}{\partial \mathbf{y}}\mathbf{f}, \tag{18}$$

where we used relation (10) between $\mathbf{C}$ and $\mathbf{D}$. Due to condition (7), equation (18) is solvable for $\mathbf{z}$.

Projecting (17) onto the tangential space by multiplying it by $\mathbf{D}^T$ on the left, we obtain equations for $\dot{\mathbf{u}}$:

$$\mathbf{D}^T\dot{\mathbf{D}}\mathbf{u} + \mathbf{D}^T\mathbf{D}\dot{\mathbf{u}} + \mathbf{D}^T\dot{\mathbf{C}}^T\mathbf{v} = \mathbf{D}^T\frac{\partial \mathbf{g}}{\partial t} + \mathbf{D}^T\frac{\partial \mathbf{g}}{\partial \mathbf{x}}\mathbf{g} + \mathbf{D}^T\frac{\partial \mathbf{g}}{\partial \mathbf{y}}\mathbf{f}, \tag{19}$$

where we used relation (10) between $\mathbf{C}$ and $\mathbf{D}$ again. Since $\mathbf{D}$ is of maximal column-rank, from (19) we can explicitly solve for $\dot{\mathbf{u}}$ in terms of t, $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{u}$. Note that from equation (12), it follows that $\mathbf{v}$ is a known function of t and $\mathbf{x}$. However, to complete the change of variables from $\mathbf{y}$ to $\mathbf{u}$, we also need a representation of $\mathbf{y}$ in terms of t, $\mathbf{x}$ and $\mathbf{u}$. To find the relation between $\mathbf{y}$ and $\mathbf{u}$, we use (16).

In order to determine whether the equations (16) are solvable for $\mathbf{y}$, we take a closer look at condition (7). From condition (7), it follows that $\operatorname{rank} \frac{\partial \mathbf{h}}{\partial \mathbf{x}}\frac{\partial \mathbf{g}}{\partial \mathbf{y}}\frac{\partial \mathbf{f}}{\partial \mathbf{z}} = k$, and, hence,

$$\operatorname{rank} \frac{\partial \mathbf{h}}{\partial \mathbf{x}} \geq k, \tag{20}$$

$$\operatorname{rank} \frac{\partial \mathbf{g}}{\partial \mathbf{y}} \geq k, \tag{21}$$

$$\operatorname{rank} \frac{\partial \mathbf{f}}{\partial \mathbf{z}} \geq k. \tag{22}$$

Moreover, we can conclude that

$$n \geq k \text{ and } m \geq k. \tag{23}$$

To verify relations (23), assume otherwise: Let $n < k$. Then the rank of $\partial \mathbf{h}/\partial \mathbf{x}$ cannot be higher than n, and, therefore, not higher than k, either, which contradicts relation (20). Similarly, if $m < k$, the rank of $\partial \mathbf{f}/\partial \mathbf{z}$ cannot be higher than m, and, therefore, not higher than k, either, which contradicts relation (22). Thus, relations (23) hold.

Let the rank of $\partial \mathbf{g}/\partial \mathbf{y}$ be equal to j. In what follows, for any vector or matrix, the superscript a denotes the first j rows, e.g. $\mathbf{y}^a = (y_1, \ldots, y_j)^T$, and the superscript d denotes the last rows starting from $j+1$, e.g. $\mathbf{y}^d = (y_{j+1}, \ldots, y_m)^T$. For simplicity, we re-order the components of $\mathbf{y}$ so that $\operatorname{rank} \frac{\partial \mathbf{g}}{\partial \mathbf{y}} = \operatorname{rank} \frac{\partial \mathbf{g}}{\partial \mathbf{y}^a} = j$. Then equations (16) can be considered as algebraic equations with respect to the $\mathbf{y}^a$. The variables $\mathbf{y}^d$ are differential variables whose differential equations are given by

$$\dot{\mathbf{y}}^d = \mathbf{f}^d. \tag{24}$$

The projected index-1 DAE obtained from (11), (16), (18), (19), and (24) is given by

$$\dot{\mathbf{x}} = \mathbf{D}\mathbf{u} + \mathbf{C}^T\mathbf{v}, \tag{25a}$$

$$\dot{\mathbf{u}} = \left(\mathbf{D}^T\mathbf{D}\right)^{-1}\mathbf{D}^T\mathbf{w}, \tag{25b}$$

$$\dot{\mathbf{y}}^d = \mathbf{f}^d, \tag{25c}$$

$$0 = \mathbf{D}\mathbf{u} + \mathbf{C}^T\mathbf{v} - \mathbf{g}, \tag{25d}$$

$$0 = \mathbf{C}\left[\mathbf{w} - \mathbf{C}^T\dot{\mathbf{v}}\right], \tag{25e}$$

where $\mathbf{v}$ is given by equations (12) and

$$\mathbf{w} = \left[\frac{\partial \mathbf{g}}{\partial t} + \frac{\partial \mathbf{g}}{\partial \mathbf{x}}\mathbf{g} + \frac{\partial \mathbf{g}}{\partial \mathbf{y}^a}\mathbf{f}^a + \frac{\partial \mathbf{g}}{\partial \mathbf{y}^d}\mathbf{f}^d - \dot{\mathbf{D}}\mathbf{u} - \dot{\mathbf{C}}^T\mathbf{v}\right]. \tag{26}$$

System (25) consists of $n + (n-k) + (m-j)$ differential equations (25a)-(25c), with respect to $n+(n-k)+(m-j)$ differential variables, $\mathbf{x}$, $\mathbf{u}$, and $\mathbf{y}^d$, respectively, and $n+k$ algebraic equations (25d) and (25e) with respect to $j+k$ algebraic variables, $\mathbf{y}^a$ and $\mathbf{z}$, respectively.

Note that equation (25e) is equation (15c) written in terms of $\mathbf{v}$, $\dot{\mathbf{C}}$, and $\dot{\mathbf{D}}$ and not in terms of derivatives of $\mathbf{h}$ explicitly.

The number of differential equations in system (15) obtained by index reduction is $n+m$, whereas the number of differential equations in system (25) obtained by the projection method is $n+m+(n-k-j)$. If $n > k+j$, an application of the projection method is not more beneficial than an application of the index reduction alone. Otherwise, the projection method performs not only index reduction but also decreases the number of differential variables. In addition, there is a number of special cases that allow to simplify system (25) further.

Special case 1. *Some of the algebraic constraints* (6c) *are linear with respect to* $\mathbf{x}$.

We are given a Hessenberg index-3 system of the form

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{x}, \mathbf{y}, \mathbf{z}), \tag{27a}$$

$$\dot{\mathbf{x}} = \mathbf{g}(t, \mathbf{x}, \mathbf{y}), \tag{27b}$$

$$0 = \mathbf{C}_1(t)\mathbf{x} + \alpha(t), \tag{27c}$$

$$0 = \mathbf{h}_2(t, \mathbf{x}), \tag{27d}$$

where $\mathbf{C}_1$ is an $l \times n$ matrix of maximal row rank, such that $l \le n$, and $\mathbf{h}_2$ is a $(k-l)$-dimensional vector function which is nonlinear with respect to $\mathbf{x}$.

First, we consider equations (27b) and (27c). Similarly to the projection method, we choose a matrix $\mathbf{D}_1$ of maximal column-rank and such that

$$\mathbf{C}_1\mathbf{D}_1 = 0. \tag{28}$$

Then, since for every t the rows of $\mathbf{C}_1$ span a subspace in $\mathbb{R}^n$ and the columns of $\mathbf{D}_1$ span the orthogonal complement to that subspace, any $\mathbf{x} \in \mathbb{R}^n$ can be represented as

$$\mathbf{x} = \mathbf{D}_1\chi + \mathbf{C}_1^T\psi, \tag{29}$$

where $\chi = (\chi_1, \ldots, \chi_{n-l})^T$ and $\psi = (\psi_1, \ldots, \psi_l)^T$. Substituting decomposition (29) into equation (27c) and making use of relation (28), we find

$$\psi = -\left(\mathbf{C}_1\mathbf{C}_1^T\right)^{-1}\alpha. \tag{30}$$

From equation (29) we find $\dot{\mathbf{x}}$:

$$\dot{\mathbf{x}} = \dot{\mathbf{D}}_1\chi + \mathbf{D}_1\dot{\chi} + \dot{\mathbf{C}}_1^T\psi + \mathbf{C}_1^T\dot{\psi}.$$

We substitute this equation into equation (27b) and multiply the result by $\mathbf{D}_1^T$ on the left, obtaining a differential equation for $\dot{\chi}$:

$$\dot{\chi} = \left(\mathbf{D}_1^T\mathbf{D}_1\right)^{-1}\mathbf{D}_1^T\left[\mathbf{g}(t, \mathbf{D}_1\chi + \mathbf{C}_1^T\psi, \mathbf{y}) - \dot{\mathbf{D}}_1\chi - \dot{\mathbf{C}}_1^T\psi\right]. \tag{31}$$

On the other hand, we can apply the projection method and derive equations (25), where

$$\mathbf{C} = \begin{pmatrix} \mathbf{C}_1 \\ \frac{\partial \mathbf{h}_2}{\partial \mathbf{x}} \end{pmatrix}.$$

We see that equations (25a) for $\dot{\mathbf{x}}$ can be replaced with equations (31). Thus, we reduce the DAEs (15) to the following DAEs:

$$\dot{\chi} = \left(\mathbf{D}_1^T\mathbf{D}_1\right)^{-1}\mathbf{D}_1^T\left[\mathbf{g} - \dot{\mathbf{D}}_1\chi - \dot{\mathbf{C}}_1^T\psi\right], \tag{32a}$$

$$\dot{\mathbf{u}} = \left(\mathbf{D}^T\mathbf{D}\right)^{-1}\mathbf{D}^T w, \tag{32b}$$

$$\dot{\mathbf{y}}^d = \mathbf{f}^d, \tag{32c}$$

$$0 = \mathbf{D}\mathbf{u} + \mathbf{C}^T\mathbf{v} - \mathbf{g}, \tag{32d}$$

$$0 = \mathbf{C}\left[w - \mathbf{C}^T\dot{\mathbf{v}}\right], \tag{32e}$$

where w is as in (26), we substituted $\mathbf{x} = \mathbf{D}_1\chi + \mathbf{C}_1^T\psi$, ψ is given by equation (30), and $\mathbf{v}$ is given by equation (12).

System (32) has $(n-l)+(n-k)+(m-j)$ differential variables χ, $\mathbf{u}$, $\mathbf{y}^d$ and $j+k$ algebraic variables $\mathbf{y}^a$, $\mathbf{z}$. □

Special case 2. *Equation* (25d) *can be solved symbolically for* $\mathbf{y}^a$*:* $\mathbf{y}^a = \widetilde{\mathbf{y}}^a(t, \mathbf{x}, \mathbf{u}, \mathbf{y}^d)$.

Then we can consider the system of equations (25a), (25b), (25c), and (25e) with $\mathbf{y}^a = \widetilde{\mathbf{y}}^a(t, \mathbf{x}, \mathbf{u}, \mathbf{y}^d)$ substituted, which we solve for $\mathbf{x}$, $\mathbf{u}$, $\mathbf{y}^d$, and $\mathbf{z}$. Consequently, j algebraic equations are removed from the system.

One possible situation where we can express $\mathbf{y}^a$ symbolically is the case where $\mathbf{g}$ is linear with respect to $\mathbf{y}^a$:

$$\mathbf{g}(t, \mathbf{x}, \mathbf{y}^a, \mathbf{y}^d) = \mathbf{A}(t, \mathbf{x}, \mathbf{y}^d)\mathbf{y}^a + \gamma(t, \mathbf{x}, \mathbf{y}^d), \tag{33}$$

and $\mathbf{A}(t, \mathbf{x}, \mathbf{y}^d)$ is of maximal column-rank. □

Special case 3. *Equation* (25e) *can be solved symbolically for* $\mathbf{z}$*:* $\mathbf{z} = \widetilde{\mathbf{z}}(t, \mathbf{x}, \mathbf{u}, \mathbf{y}^a, \mathbf{y}^d)$.

Then we can consider the system of equations (25a), (25b), (25c), and (25d) where $\mathbf{z} = \widetilde{\mathbf{z}}(t, \mathbf{x}, \mathbf{u}, \mathbf{y}^a, \mathbf{y}^d)$ was substituted, which we solve for $\mathbf{x}$, $\mathbf{u}$, $\mathbf{y}^a$, and $\mathbf{y}^d$, and k algebraic equations are removed from the system.

One possible situation where we can express $\mathbf{z}$ symbolically is the case where $\mathbf{f}$ is linear with respect to $\mathbf{z}$:

$$\mathbf{f}(t, \mathbf{x}, \mathbf{y}^a, \mathbf{y}^d, \mathbf{z}) = \varphi(t, \mathbf{x}, \mathbf{y}^a, \mathbf{y}^d) + \mathbf{B}(t, \mathbf{x}, \mathbf{y}^a, \mathbf{y}^d)\mathbf{z}. \tag{34}$$

In order to satisfy condition (7), $\mathbf{B}$ must be such that $\dfrac{\partial \mathbf{h}}{\partial \mathbf{x}}\dfrac{\partial \mathbf{g}}{\partial \mathbf{y}}\mathbf{B}$ is nonsingular for all t. □

When we consider Special cases 2 and 3 together, we can obtain an ODE system with respect to $n + (n-k) + (m-j)$ variables $\mathbf{x}$, $\mathbf{u}$, and $\mathbf{y}^d$. Let us consider what we obtain in the case where $\mathbf{f}$ and $\mathbf{g}$ are given by equations (34) and (33), respectively:

$$\dot{\mathbf{x}} = \mathbf{D}\mathbf{u} + \mathbf{C}^T\mathbf{v}, \tag{35a}$$

$$\dot{\mathbf{u}} = \left(\mathbf{D}^T\mathbf{D}\right)^{-1}\mathbf{D}^T\left[\mathbf{P} + \mathbf{Q}\widetilde{\mathbf{z}} - \dot{\mathbf{D}}\mathbf{u} - \dot{\mathbf{C}}^T\mathbf{v}\right], \tag{35b}$$

$$\dot{\mathbf{y}}^d = \widetilde{\varphi}_d + \widetilde{\mathbf{B}}_d\widetilde{\mathbf{z}}, \tag{35c}$$

where

$$\mathbf{P} = \frac{\partial \mathbf{A}}{\partial t}\widetilde{\mathbf{y}}^a + \frac{\partial \gamma}{\partial t} + \left(\frac{\partial \mathbf{A}}{\partial \mathbf{x}}\widetilde{\mathbf{y}}^a) + \frac{\partial \gamma}{\partial \mathbf{x}}\right)(\mathbf{A}\widetilde{\mathbf{y}}^a + \gamma) + \mathbf{A}\widetilde{\varphi}^a + \left(\frac{\partial \mathbf{A}}{\partial \mathbf{y}^d}\widetilde{\mathbf{y}}^a + \frac{\partial \gamma}{\partial \mathbf{y}^d}\right)\widetilde{\varphi}^d, \tag{36a}$$

$$\mathbf{Q} = \mathbf{A}\widetilde{\mathbf{B}}_a + \left(\frac{\partial \mathbf{A}}{\partial \mathbf{y}^d}\widetilde{\mathbf{y}}^a + \frac{\partial \gamma}{\partial \mathbf{y}^d}\right)\widetilde{\mathbf{B}}^d, \tag{36b}$$

$$\widetilde{\mathbf{B}} = \mathbf{B}(t, \mathbf{x}, \widetilde{\mathbf{y}}^a(t, \mathbf{x}, \mathbf{u}, \mathbf{y}^d), \mathbf{y}^d), \tag{36c}$$

$$\widetilde{\varphi} = \varphi(t, \mathbf{x}, \widetilde{\mathbf{y}}^a(t, \mathbf{x}, \mathbf{u}, \mathbf{y}^d), \mathbf{y}^d), \tag{36d}$$

$$\widetilde{\mathbf{y}}^a = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\left[\mathbf{D}\mathbf{u} + \mathbf{C}^T\mathbf{v} - \gamma\right], \tag{36e}$$

$$\widetilde{\mathbf{z}} = (\mathbf{C}\mathbf{Q})^{-1}\mathbf{C}\left[\dot{\mathbf{D}}\mathbf{u} + \dot{\mathbf{C}}^T\mathbf{v} + \mathbf{C}^T\dot{\mathbf{v}} - \mathbf{P}\right]. \tag{36f}$$

When system (35) is solved, $\mathbf{y}^a$ is given by $\widetilde{\mathbf{y}}^a$ from equation (36e), and $\mathbf{z}$ is given by $\widetilde{\mathbf{z}}$ from equation (36f). Equations (35) have the simplest form we can get by applying projection to the reduced system (15).

REMARK 1. *The original projection method [16, 6, 7, 1] was developed for mechanical systems under holonomic constraints:*

$$\mathbf{M}\ddot{\mathbf{x}} = \varphi(t, \mathbf{x}, \dot{\mathbf{x}}) + \mathbf{C}^T\mathbf{z} \tag{37a}$$

$$0 = \mathbf{h}(t, \mathbf{x}), \tag{37b}$$

where $\mathbf{x}$ *denotes the coordinates,* $\mathbf{z}$ *denote the Lagrange multipliers,* $\mathbf{M}$ *is a symmetric positive-definite generalized mass matrix,* φ *is independent of* $\mathbf{z}$ *and represents forces acting on the system. Denoting the velocities by* $\mathbf{y}$*, we can re-write the system* (37) *in the form described by Special cases 2 and 3 with* $\mathbf{y} = \mathbf{y}^a$*,* $\mathbf{g} = \mathbf{y}$*, and* $\mathbf{f}$ *linear with respect to* $\mathbf{z}$*. However, in this case the computations simplify if we first multiply both sides of equation* (17) *by* $\mathbf{M}$ *on the left and then project the result onto the tangential space, and use equation* (17) *as is when projecting onto the normal space. Equations* (37) *are simplified to:*

$$\dot{\mathbf{x}} = \mathbf{D}\mathbf{u} + \mathbf{C}^T\mathbf{v},$$

$$\dot{\mathbf{u}} = \left(\mathbf{D}^T\mathbf{M}\,\mathbf{D}\right)^{-1}\mathbf{D}^T\left[\widetilde{\varphi} - \mathbf{M}\left(\dot{\mathbf{D}}\mathbf{u} + \dot{\mathbf{C}}^T\mathbf{v} + \mathbf{C}^T\dot{\mathbf{v}}\right)\right],$$

where we used $\mathbf{C}\mathbf{D} = 0$ *and* $\widetilde{\varphi} = \varphi(t, \mathbf{x}, \widetilde{\mathbf{y}})$*. Equations* (36e) *and* (36f)*, respectively, become:*

$$\widetilde{\mathbf{y}} = \mathbf{D}\mathbf{u} + \mathbf{C}^T\mathbf{v},$$

$$\widetilde{\mathbf{z}} = \left(\mathbf{C}\,\mathbf{M}^{-1}\mathbf{C}^T\right)^{-1}\mathbf{C}\left[\dot{\mathbf{D}}\mathbf{u} + \dot{\mathbf{C}}^T\mathbf{v} + \mathbf{C}^T\,\dot{\mathbf{v}} - \mathbf{M}^{-1}\widetilde{\varphi}\right].$$

The last equation is not needed to obtain the motion of the mechanical system but can be used to find the constraint forces, since the vector $\mathbf{C}^T\mathbf{z}$ *represents the generalized forces induced by the constraints.*

Lagrange multipliers $\mathbf{z}$ appear in the equations for a constrained mechanical system (37) as a result of application of the *Principle of Least Action* [4, 13]. For a constrained mechanical system, this principle results in a problem of optimization under constraints. We can show that any optimization under constraints with an objective function similar to the mechanical action gives a DAE system that can be simplified to an ODE system by the projection method. We demonstrate this in Section 6.

4. The Projection Method and Higher Hessenberg Index DAEs

In this section we generalize the approach from the previous section to DAEs of arbitrary Hessenberg index.

We consider a Hessenberg index-$(r+2)$ DAE

$$\dot{\mathbf{y}}_r = \mathbf{f}_r(t, \mathbf{x}, \mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_r, \mathbf{z}), \tag{38a}$$

$$\vdots$$

$$\dot{\mathbf{y}}_2 = \mathbf{f}_2(t, \mathbf{x}, \mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3), \tag{38b}$$

$$\dot{\mathbf{y}}_1 = \mathbf{f}_1(t, \mathbf{x}, \mathbf{y}_1, \mathbf{y}_2), \tag{38c}$$

$$\dot{\mathbf{x}} = \mathbf{g}(t, \mathbf{x}, \mathbf{y}_1), \tag{38d}$$

$$0 = \mathbf{h}(t, \mathbf{x}), \tag{38e}$$

where

$$\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\frac{\partial \mathbf{g}}{\partial \mathbf{y}_1}\frac{\partial \mathbf{f}_1}{\partial \mathbf{y}_2}\cdots\frac{\partial \mathbf{f}_{r-1}}{\partial \mathbf{y}_r}\frac{\partial \mathbf{f}_r}{\partial \mathbf{z}} \text{ is nonsingular for all } t. \tag{39}$$

The vectors $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_r$ are of size $m_1, m_2, \ldots, m_r$, respectively. Due to condition (39), the ranks of the Jacobians $\frac{\partial \mathbf{g}}{\partial \mathbf{y}_1}, \frac{\partial \mathbf{f}_1}{\partial \mathbf{y}_2}, \ldots, \frac{\partial \mathbf{f}_{r-1}}{\partial \mathbf{y}_r}$ and their products forming subproducts in condition (39) are at least k at any t. We reorder the components of the vectors $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_r$ so that the first $j_1, j_2, \ldots, j_r$ components, respectively, denoted by superscript a, are such that:

$$\operatorname{rank}\frac{\partial \mathbf{g}}{\partial \mathbf{y}_1} = \operatorname{rank}\frac{\partial \mathbf{g}}{\partial \mathbf{y}_1^a} = j_1, \tag{40a}$$

$$\operatorname{rank}\frac{\partial \mathbf{g}}{\partial \mathbf{y}_1}\frac{\partial \mathbf{f}_1}{\partial \mathbf{y}_2} = \operatorname{rank}\frac{\partial \mathbf{g}}{\partial \mathbf{y}_1}\frac{\partial \mathbf{f}_1}{\partial \mathbf{y}_2^a} = j_2, \tag{40b}$$

$$\vdots$$

$$\operatorname{rank}\frac{\partial \mathbf{g}}{\partial \mathbf{y}_1}\frac{\partial \mathbf{f}_1}{\partial \mathbf{y}_2}\cdots\frac{\partial \mathbf{f}_{r-1}}{\partial \mathbf{y}_r} = \operatorname{rank}\frac{\partial \mathbf{g}}{\partial \mathbf{y}_1}\frac{\partial \mathbf{f}_1}{\partial \mathbf{y}_2}\cdots\frac{\partial \mathbf{f}_{r-1}}{\partial \mathbf{y}_r^a} = j_r. \tag{40c}$$

As in the previous cases the projection method leads to the representation

$$\dot{\mathbf{x}} = \mathbf{D}\mathbf{u}_1 + \mathbf{C}^T\mathbf{v}, \tag{41}$$

where $\mathbf{C} = \frac{\partial \mathbf{h}}{\partial \mathbf{x}}$ is of maximal row-rank, $\mathbf{D}$ is of maximal column-rank and satisfies (10), $\mathbf{v} = -\left(\mathbf{C}\mathbf{C}^T\right)^{-1}\frac{\partial \mathbf{h}}{\partial t}$, and $\mathbf{u}_1$ is an $(n-k)$-dimensional vector. Then we introduce $(n-k)$-dimensional vectors $\mathbf{u}_2, \ldots, \mathbf{u}_r$ such that

$$\dot{\mathbf{u}}_1 = \mathbf{u}_2, \tag{42a}$$

$$\dot{\mathbf{u}}_2 = \mathbf{u}_3, \tag{42b}$$

$$\vdots$$

$$\dot{\mathbf{u}}_{r-1} = \mathbf{u}_r, \tag{42c}$$

and we want to re-write equations (38) in terms of the projection of $\dot{\mathbf{x}}$ onto the tangential space, $\mathbf{u}_1$, and its derivatives. We substitute (41) into equation (38d), obtaining

$$\mathbf{D}\mathbf{u}_1 + \mathbf{C}^T\mathbf{v} = \mathbf{g}(t, \mathbf{x}, \mathbf{y}_1). \tag{43}$$

From (40a), it follows that (43) can be solved for $\mathbf{y}_1^a$.

Differentiating (43) with respect to time, we obtain

$$\dot{\mathbf{D}}\mathbf{u}_1+\mathbf{D}\mathbf{u}_2+\dot{\mathbf{C}}^T\mathbf{v}+\mathbf{C}^T\dot{\mathbf{v}}=\frac{\partial\mathbf{g}}{\partial t}+\frac{\partial\mathbf{g}}{\partial\mathbf{x}}\mathbf{g}+\frac{\partial\mathbf{g}}{\partial\mathbf{y}_1}\mathbf{f}_1, \quad (44)$$

where we made use of (38d), (38c), and (42a). From condition (40b), it follows that (44) can be solved for $\mathbf{y}_2^a$.

Differentiating equation (43) with respect to time a second time, i.e. differentiating equation (44), and replacing derivatives of variables that occur in the left-hand side with their representation according to equations (41) and (42) and derivatives that occur in the right-hand side according to equations (38), we obtain algebraic equations that can be solved for $\mathbf{y}_3^a$. We continue this process until we differentiate equation (43) r times. After each differentiation we make use of equations (41) and (42) in the left-hand side and of equations (38) on the right-hand side. The i-th differentiation gives us an algebraic equation, which due to conditions (40) can be solved for $\mathbf{y}_{i+1}^a$, for $i=1,\ldots,r-1$.

The r-th differentiation gives us an equation, from which we derive a differential equation for $\dot{\mathbf{u}}_r$ and an algebraic equation for $\mathbf{z}$. Let us consider the result of r-th differentiation. We are interested only in the terms that define equations for $\dot{\mathbf{u}}_r$ and $\mathbf{z}$, and, therefore, for simplicity, we represent the result of r-th differentiation as follows:

$$\begin{aligned}&\mathbf{D}\dot{\mathbf{u}}_r+\varphi_1\left(t,\mathbf{x},\mathbf{u}_1,\mathbf{u}_2,\ldots,\mathbf{u}_r\right) \quad (45)\\ &=\frac{\partial\mathbf{g}}{\partial\mathbf{y}_1}\frac{\partial\mathbf{f}_1}{\partial\mathbf{y}_2}\frac{\partial\mathbf{f}_2}{\partial\mathbf{y}_3}\cdots\frac{\partial\mathbf{f}_{r-1}}{\partial\mathbf{y}_r}\mathbf{f}_r+\varphi_2\left(t,\mathbf{x},\mathbf{y}_1,\mathbf{y}_2,\ldots,\mathbf{y}_r\right),\end{aligned}$$

where functions φ_1 and φ_2 represent the remaining terms, in which we are not interested.

If we project equation (45) onto the tangential space of the constraint manifold by multiplying both sides of the equation by $\mathbf{D}^T$ on the left, we obtain an equation for $\dot{\mathbf{u}}_r$.

$$\dot{\mathbf{u}}_r=\left(\mathbf{D}^T\mathbf{D}\right)^{-1}\mathbf{D}^T\mathbf{e}, \quad (46)$$

where

$$\mathbf{e}=\left[\frac{\partial\mathbf{g}}{\partial\mathbf{y}_1}\frac{\partial\mathbf{f}_1}{\partial\mathbf{y}_2}\frac{\partial\mathbf{f}_2}{\partial\mathbf{y}_3}\cdots\frac{\partial\mathbf{f}_{r-1}}{\partial\mathbf{y}_r}\mathbf{f}_r+\varphi_2-\varphi_1\right]. \quad (47)$$

If we project equation (45) onto the normal space of the constraint manifold by multiplying both sides of the equation by $\mathbf{C}$ on the left, we obtain an equation for $\mathbf{z}$, which is solvable due to condition (39) and is independent of $\dot{\mathbf{u}}_r$ due to relation (10).

$$0=\left(\mathbf{C}\mathbf{C}^T\right)^{-1}\mathbf{C}\,\mathbf{e} \quad (48)$$

The DAE system simplified by the projection method is then given by equations (41), (42), (43), (44), (46), (48), algebraic equations for $\mathbf{y}_3^a,\ldots,\mathbf{y}_r^a$ and differential equations for $\mathbf{y}_1^d,\ldots,\mathbf{y}_r^d$, where we represent each $\mathbf{y}_i$ in terms of $\mathbf{y}_i^a$ and $\mathbf{y}_i^d$, for $i=1,\ldots,r$.

$$\dot{\mathbf{x}}=\mathbf{D}\mathbf{u}_1+\mathbf{C}^T\mathbf{v}, \quad (49a)$$

$$\dot{\mathbf{u}}_1=\mathbf{u}_2, \quad (49b)$$

$$\dot{\mathbf{u}}_2=\mathbf{u}_3, \quad (49c)$$

$$\vdots$$

$$\dot{\mathbf{u}}_{r-1}=\mathbf{u}_r, \quad (49d)$$

$$\dot{\mathbf{u}}_r=\left(\mathbf{D}^T\mathbf{D}\right)^{-1}\mathbf{D}^T\mathbf{e}, \quad (49e)$$

$$\dot{\mathbf{y}}_1^d=\mathbf{f}_1^d, \quad (49f)$$

$$\dot{\mathbf{y}}_2^d=\mathbf{f}_2^d, \quad (49g)$$

$$\vdots$$

$$\dot{\mathbf{y}}_r^d=\mathbf{f}_r^d, \quad (49h)$$

$$0=\mathbf{D}\mathbf{u}_1+\mathbf{C}^T\mathbf{v}-\mathbf{g}, \quad (49i)$$

$$0=\dot{\mathbf{D}}\mathbf{u}_1+\mathbf{D}\mathbf{u}_2+\dot{\mathbf{C}}^T\mathbf{v}+\mathbf{C}^T\dot{\mathbf{v}}-\frac{\partial\mathbf{g}}{\partial t}-\frac{\partial\mathbf{g}}{\partial\mathbf{x}}\mathbf{g}-\frac{\partial\mathbf{g}}{\partial\mathbf{y}_1}\mathbf{f}_1, \quad (49j)$$

$$0=\left.\frac{\mathrm{d}^3}{\mathrm{d}t^3}\left(\mathbf{D}\mathbf{u}_1+\mathbf{C}^T\mathbf{v}\right)\right|_{(41),(42)}-\left.\frac{\mathrm{d}^3}{\mathrm{d}t^3}\left(\mathbf{g}\right)\right|_{(38)}, \quad (49k)$$

$$\vdots$$

$$0=\left.\frac{\mathrm{d}^{r-1}}{\mathrm{d}t^{r-1}}\left(\mathbf{D}\mathbf{u}_1+\mathbf{C}^T\mathbf{v}\right)\right|_{(41),(42)}-\left.\frac{\mathrm{d}^{r-1}}{\mathrm{d}t^{r-1}}\left(\mathbf{g}\right)\right|_{(38)}, \quad (49l)$$

$$0=\left(\mathbf{C}\mathbf{C}^T\right)^{-1}\mathbf{C}\,\mathbf{e}. \quad (49m)$$

System (49) consists of differential equations (49a)-(49h) w.r.t. $n+r(n-k)+(m_1-j_1)+\ldots+(m_r-j_r)$ variables $\mathbf{x},\mathbf{u}_1,\ldots,\mathbf{u}_r,\mathbf{y}_1^d,\ldots,\mathbf{y}_r^d$ and of algebraic equations (49i)-(49m) w.r.t. $j_1+j_2+\ldots+j_r+k$ variables $\mathbf{y}_1^a,\ldots,\mathbf{y}_r^a,\mathbf{z}$. Index reduction alone yields an index-1 DAE w.r.t. $n+m_1+\ldots+m_r$ differential and k algebraic variables.

Special case 4. *Equations* (49i)-(49m) *can be solved symbolically.*

We can decouple these equations from the system, removing $j_1+j_2+\ldots+j_r+k$ algebraic variables and obtaining an ODE system given by equations (49a)-(49h). □

An example of a Hessenberg system of higher index falling under the Special case 4 is a high-order differential equation subject to algebraic constraints.

The analog of Special case 1 can be also considered for higher Hessenberg index DAEs. Then, following the guidelines given in the Special case 1, we can additionally eliminate l differential variables, where l is the number of linear constraints.

5. A Particular Case of DAEs of More General Form

In this section we consider a case when DAEs are not of the form (2), (4), or (6) and show how they can be reduced to one of these forms, in order to apply the projection method. We call this case *mixed Hessenberg index-1,3 form*.

We define the mixed Hessenberg index-1,3 form as a system of the form

$$\dot{\mathbf{y}}=\mathbf{f}(\mathbf{x},\mathbf{y},\mathbf{z}), \quad (50a)$$

$$\dot{\mathbf{x}}=\mathbf{g}(\mathbf{x},\mathbf{y}), \quad (50b)$$

$$0=\mathbf{h}(\mathbf{x})+\mathbf{A}(\mathbf{x})\mathbf{z}, \quad (50c)$$

where the matrix $\mathbf{A}$ is not invertible. We also define a solvability condition guaranteeing that system (50) has index 3.

In order to formulate the solvability condition, we introduce additional matrices. If the matrix $\mathbf{A}$ has rank $k_1<k$,

re-ordering its rows and columns if necessary, we can represent it as

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix}, \tag{51}$$

where

$$\text{rank } \mathbf{A} = \text{ rank } \mathbf{A}_{11} = k_1, \tag{52}$$

and $\mathbf{A}_{11}$, $\mathbf{A}_{12}$, $\mathbf{A}_{21}$, and $\mathbf{A}_{22}$ are, respectively, $k_1 \times k_1$, $k_1 \times (k-k_1)$, $(k-k_1) \times k_1$, and $(k-k_1) \times (k-k_1)$-dimensional matrices. The condition (52) yields that $\mathbf{A}_{11}$ is invertible.

We introduce matrices $\mathbf{L}$ and $\mathbf{R}$:

$$\mathbf{L} = \begin{pmatrix} \mathbf{L}_1 \\ \mathbf{L}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{I}_{k_1} & \mathbf{0} \\ -\mathbf{A}_{21}\mathbf{A}_{11}^{-1} & \mathbf{I}_{k-k_1} \end{pmatrix}, \tag{53a}$$

$$\mathbf{R} = \begin{pmatrix} \mathbf{R}_1 \mid \mathbf{R}_2 \end{pmatrix} = \left(\begin{array}{c|c} \mathbf{I}_{k_1} & -\mathbf{A}_{11}^{-1}\mathbf{A}_{12} \\ \mathbf{0} & \mathbf{I}_{k-k_1} \end{array}\right). \tag{53b}$$

Now we can formulate the solvability condition as follows:

$$\frac{\partial(\mathbf{L}_2\mathbf{h})}{\partial \mathbf{x}} \frac{\partial \mathbf{g}}{\partial \mathbf{y}} \frac{\partial \mathbf{f}}{\partial \mathbf{z}} \mathbf{R}_2 \text{ is invertible.} \tag{54}$$

In order to apply the projection method, we introduce a change of variables:

$$\mathbf{z} = \mathbf{R}\zeta. \tag{55}$$

Substituting transformation (55) into equation (50c) and multiplying it by $\mathbf{L}$ on the left, we obtain:

$$\mathbf{L}\mathbf{h} + \mathbf{L}\mathbf{A}\mathbf{R}\zeta = 0. \tag{56}$$

From definitions (53) it follows that

$$\mathbf{L}\mathbf{A}\mathbf{R} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}.$$

Let us denote the first k_1 rows of ζ by ζ_1 and the last $k-k_1$ by ζ_2 and consider the first k_1 rows of equation (56):

$$\mathbf{L}_1\mathbf{h} + \mathbf{A}_{11}\zeta_1 = 0. \tag{57}$$

From condition (52) it follows that equation (57) can be solved for ζ_1, obtaining:

$$\zeta_1 = -\mathbf{A}_{11}^{-1}\mathbf{L}_1\mathbf{h}. \tag{58}$$

The last $k-k_1$ rows of equation (56) yield the following.

$$\mathbf{L}_2\mathbf{h} = 0, \tag{59}$$

which we consider as a new algebraic constraint. Substituting transformation (55) and representation (58) into the right-hand side of equation (50a), we obtain:

$$\begin{aligned} \mathbf{f}(\mathbf{x}, \mathbf{y}, \mathbf{R}\zeta) &= \mathbf{f}(\mathbf{x}, \mathbf{y}, \mathbf{R}_1\zeta_1 + \mathbf{R}_2\zeta_2) \\ &= \mathbf{f}(\mathbf{x}, \mathbf{y}, \mathbf{R}_2\zeta_2 - \mathbf{R}_1\mathbf{A}_{11}^{-1}\mathbf{L}_1\mathbf{h}). \end{aligned}$$

We obtain a Hessenberg index-3 DAE system

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{x}, \mathbf{y}, \mathbf{R}_2\zeta_2 - \mathbf{R}_1\mathbf{A}_{11}^{-1}\mathbf{L}_1\mathbf{h}), \tag{60a}$$

$$\dot{\mathbf{x}} = \mathbf{g}(\mathbf{x}, \mathbf{y}), \tag{60b}$$

$$0 = \mathbf{L}_2(\mathbf{x})\mathbf{h}(\mathbf{x}). \tag{60c}$$

From the solvability condition (54) it follows that condition (7) on the product of the Jacobians is satisfied, and the system (60) is indeed of a Hessenberg index-3 DAE form. The projection method can now be applied to system (60).

We also consider a special case of mixed Hessenberg index-1,3 form, where the matrix $\mathbf{A}$ can be represented, by re-ordering columns and rows if necessary, in the form

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}, \tag{61}$$

where $\mathbf{A}_{11}$ is invertible. Equations (50) now take the form

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{x}, \mathbf{y}, \mathbf{z}_1, \mathbf{z}_2), \tag{62a}$$

$$\dot{\mathbf{x}} = \mathbf{g}(\mathbf{x}, \mathbf{y}), \tag{62b}$$

$$0 = \mathbf{h}_1(\mathbf{x}) + \mathbf{A}_{11}(\mathbf{x})\mathbf{z}_1 + \mathbf{A}_{12}(\mathbf{x})\mathbf{z}_2, \tag{62c}$$

$$0 = \mathbf{h}_2(\mathbf{x}), \tag{62d}$$

assuming the solvability condition is satisfied:

$$\frac{\partial \mathbf{h}_2}{\partial \mathbf{x}} \frac{\partial \mathbf{g}}{\partial \mathbf{y}} \frac{\partial \mathbf{f}}{\partial \mathbf{z_2}} \text{ is nonsingular for all } t.$$

In this case we do not need to define matrices $\mathbf{L}$ and $\mathbf{R}$. Our goal is to obtain index-1 DAEs, and the equations (62c) are already in index-1 form. We then apply the projection method to the system (62a), (62b), and (62d), treating the variables $\mathbf{z}_1$ as known, to obtain index-1 equations for $\mathbf{y}$ and $\mathbf{z}_2$ and obtain an index-1 system of DAEs consisting of the result of application of the projection method and (62c).

6. Optimization Under Constraints

We now show how a general optimization problem can be represented as a higher-index Hessenberg system, and how the projection method can then be applied.

Consider the problem of optimizing the functional

$$F(\mathbf{x}) = \int_{t_1}^{t_2} \mathcal{L}\left(t, \mathbf{x}, \dot{\mathbf{x}}, \ldots, \delta^k\mathbf{x}\right) dt, \tag{63}$$

where $\delta^k\mathbf{x} = \frac{\partial^k \mathbf{x}}{\partial t^k}$, subject to the constraints

$$\phi(\mathbf{x}) = \int_{t_1}^{t_2} \mathbf{h}(t, \mathbf{x}) dt = 0. \tag{64}$$

$\mathcal{L}$ is called a *Lagrangian*, $\mathbf{h} = (h_1, \ldots, h_m)^T$, $\phi = (\phi_1, \ldots, \phi_m)^T$, and $\phi_j = \int_{t_1}^{t_2} h_j(\mathbf{x}(t))dt$ for $j = 1, \ldots m$.

For example, a Lagrangian for a particle of charge q and mass m in an electromagnetic field with characteristics $\mathbf{E}$ and $\mathbf{B}$ is given by

$$\mathcal{L} = \frac{1}{2} m\dot{\mathbf{x}}^T\dot{\mathbf{x}} + q\dot{\mathbf{x}}^T\mathbf{A}(t, \mathbf{x}) - q\Phi(t, \mathbf{x}), \tag{65}$$

where $\mathbf{x}$ is the position of the particle. The scalar potential Φ and the vector potential $\mathbf{A}$ are such that

$$\mathbf{B} = \text{curl } \mathbf{A}, \quad \mathbf{E} = -\nabla\Phi - \frac{\partial \mathbf{A}}{\partial t},$$

where curl $\mathbf{A}$ is the curl of $\mathbf{A}$, and $\nabla\Phi$ is the gradient of Φ.

For the case of several particle charges, a Lagrangian is obtained as a sum of the Lagrangians of all particles. There is also a relativistic counterpart of Lagrangian (65) [4].

The variational principle for an electric circuit arises from imposing that the energy losses must be as small as possible [4]. A Lagrangian for interconnected electrical circuits consisting of linear elements can be written as follows [13]

$$\mathcal{L} = \frac{1}{2}\sum_{j,k} M_{jk}\dot{I}_j\dot{I}_k - \frac{1}{2}\sum_j \frac{1}{C_j}I_j^2 + \sum_j \dot{E}_j\dot{I}_j, \quad (66)$$

where I_j is the current through the j-th element, M_{jk} is the mutual inductance between the j-th and k-th inductors, C_j is the capacitance of the j-th capacitor, and E_j is the j-th electromotive force. Lagrangian (66) describes the energy exchange between the elements.

Resistors in an electric circuit act as generalized dissipative forces Q_j described by the function

$$W = \frac{1}{2}\sum_j R_j \dot{I}_j^2,$$

where

$$Q_j = -\frac{\partial W}{\partial \dot{I}_j} = -R_j\dot{I}_j$$

and R_j is the resistance of the j-th resistor.

Then the energy conservation law states that the loss of energy described by $\mathcal{L}$ is equal to the work done by the dissipative forces to create a change δI_j.

Lagrangian (66) can be also used to describe a mechanical system of masses connected by springs if parameters are defined as following: I_j is the displacement of the j-th mass, M is the reciprocal mass tensor, $1/C_j$ is the spring constant of the j-th spring, $\dot{E}_j$ is the j-th driving force, R_j is the viscous force of the j-th damper [13], and W is the Rayleigh dissipation function.

Optimal control problems can be formulated as problems of maximizing a functional using the Pontrjagin maximum principle [5].

There is no unique, general, automatic method for determining a Lagrangian for a physical system. Typically, the derivation of a Lagrangian is based on the invariance properties of the system [13]. For example, a free particle in a gravitational field is invariant under changes of time $t \to t + t_0$ without any change in coordinates, i.e. time has no privilege of origin, and, therefore, $\partial\mathcal{L}/\partial t = 0$. Space also has no privilege of origin for the particle, and, therefore, $\partial\mathcal{L}/\partial x_i = 0$. Finally, rotation invariance of the particle implies that $\mathcal{L}$ can only depend on the square of the velocity, i.e., it is of the form $\mathcal{L}(\dot{\mathbf{x}}^2)$ [4].

To tackle optimization problem (63), (64), we formulate the necessary condition for a functional $\mathcal{F}(\mathbf{x})$ subject to constraints $\psi(\mathbf{x}) = \mathbf{y}_0$ to have an extremum at $\mathbf{x}_0(t)$ [9].

THEOREM 2. *Let $U \subset \mathbb{R}^n$ be open and $\mathcal{F} : U \to \mathbb{R}$ a differential functional; furthermore, let $\psi : U \to \mathbb{R}^m$ be a continuously differential mapping. Let the functional $\mathcal{F}$ attain a local extremum at a regular point $\mathbf{x}_0 \in U$ of the mapping ψ subject to the condition $\psi(\mathbf{x}) = \mathbf{y}_0$. Then there exist real numbers $\lambda_1, \dots \lambda_m$ such that*

$$\frac{\partial \mathcal{F}}{\partial x_i}(\mathbf{x}_0) = \sum_{j=1}^m \lambda_j \frac{\partial \psi_j}{\partial x_i}(\mathbf{x}_0), \quad i = 1, \dots, n,$$

where all derivatives are understood in the Fréchet sense.

From Theorem 2 it follows that to find the extreme points of functional (63) under constraints (64), we need to solve the following system of DAEs:

$$\frac{\partial \mathcal{L}}{\partial x_i} - \frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{x}_i}\right) + \dots + (-1)^k \frac{d^k}{dt^k}\left(\frac{\partial \mathcal{L}}{\partial\left(\delta^k x_i\right)}\right) = \sum_{j=1}^m \lambda_j \frac{\partial h_j}{\partial x_i}, \quad 1 \leqslant i \leqslant n, \quad (67a)$$

$$h_j(x_1, \dots, x_n, t) = 0, \qquad 1 \leqslant j \leqslant m. \quad (67b)$$

For Lagrangian (66) describing an electric circuit in the absence of dissipative forces, equations (67) take the form

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{I}_i}\right) - \frac{\partial \mathcal{L}}{\partial I_i} = \sum_{j=1}^m \lambda_j \frac{\partial h_j}{\partial x_i}, \quad 1 \leqslant i \leqslant n.$$

In the presence of dissipative forces the energy loss is equal to the work done by the dissipative forces

$$\frac{\partial F}{\partial I_j} = \int_{t_1}^{t_2} -\frac{\partial W}{\partial \dot{I}_j}\delta I_j$$

and it has to be as small as possible. Then equations (67) become:

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{I}_i}\right) - \frac{\partial \mathcal{L}}{\partial I_i} + \frac{\partial W}{\partial \dot{I}_j} = \sum_{j=1}^m \lambda_j \frac{\partial h_j}{\partial x_i}, \quad 1 \leqslant i \leqslant n. \quad (68)$$

We can re-write equation (67) as:

$$\mathbf{M}(t, \mathbf{x}, \dot{\mathbf{x}}, \dots, \delta^{2k-1}\mathbf{x})\delta^{2k}\mathbf{x} = \mathbf{f}(t, \mathbf{x}, \dot{\mathbf{x}}, \dots, \delta^{2k-1}\mathbf{x}) + \mathbf{C}^T\lambda, \quad (69a)$$

$$\mathbf{h}(t, \mathbf{x}) = 0, \quad (69b)$$

where $\mathbf{C} = \partial\mathbf{h}/\partial\mathbf{x}$, $\lambda = (\lambda_1, \dots \lambda_m)^T$, and $\mathbf{f}$ is a function of the derivatives of $\mathbf{x}$ of order $< 2k$. System (69) is an instance of Special case 4 from Section 4. In order to see this we re-write system (69) as a system of first-order DAEs. Since we know that in Special case 4 we obtain a significant simplification by applying the projection method, we follow the steps in Section 4 and introduce a set of $(n-m)$-dimensional variables $\mathbf{u}_1, \dots, \mathbf{u}_{2k-1}$ such that

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{D}\mathbf{u}_1 + \mathbf{C}^T\mathbf{v}, \\ \dot{\mathbf{u}}_1 &= \mathbf{u}_2, \\ &\vdots \\ \dot{\mathbf{u}}_{2k-2} &= \mathbf{u}_{2k-1}. \end{aligned} \quad (70)$$

Then equation (69) can be rewritten as

$$\widetilde{\mathbf{M}}\mathbf{D}\dot{\mathbf{u}}_{2k-1} = \widetilde{\mathbf{f}} + \mathbf{C}^T\lambda - \widetilde{\mathbf{M}}\widetilde{\sigma} - \widetilde{\mathbf{M}}\delta^{2k-1}\left(\mathbf{C}^T\mathbf{v}\right), \quad (71)$$

where $\mathbf{D}$ is such that $\mathbf{C}\mathbf{D} = 0$,

$$\begin{aligned} \mathbf{v} &= -\left(\mathbf{C}\mathbf{C}^T\right)^{-1}\frac{\partial \mathbf{h}}{\partial t}, \\ \widetilde{\mathbf{M}} &= \widetilde{\mathbf{M}}(t, \mathbf{x}, \mathbf{u}_1, \dots, \mathbf{u}_{2k-1}) \\ &= \mathbf{M}(t, \mathbf{x}, \mathbf{D}\mathbf{u}_1 + \mathbf{C}^T\mathbf{v}, \dots, \delta^{2k-2}(\mathbf{D}\mathbf{u}_1 + \mathbf{C}^T\mathbf{v})), \end{aligned}$$

$$\begin{aligned}\widetilde{\mathbf{f}} &= \widetilde{\mathbf{f}}(t, \mathbf{x}, \mathbf{u}_1, \ldots, \mathbf{u}_{2k-1}) \\ &= \mathbf{f}(t, \mathbf{x}, \mathbf{D}\mathbf{u}_1 + \mathbf{C}^T\mathbf{v}, \ldots, \delta^{2k-2}(\mathbf{D}\mathbf{u}_1 + \mathbf{C}^T\mathbf{v})), \\ \widetilde{\sigma} &= \widetilde{\sigma}(t, \mathbf{x}, \mathbf{u}_1, \ldots, \mathbf{u}_{2k-1}) \\ &= \delta^{2k-1}(\mathbf{D}\mathbf{u}_1) - \mathbf{D}\delta^{2k-1}\mathbf{u}_1\end{aligned}$$

and we have replaced the derivatives of $\mathbf{u}_1$ with their expressions in terms of $\mathbf{u}_2, \ldots, \mathbf{u}_{2k-1}$ according to (71).

Now we multiply equation (71) by $\mathbf{D}^T$ from the left and, combining with (70), get the following simplified system:

$$\dot{\mathbf{x}} = \mathbf{D}\mathbf{u}^{(1)} + \mathbf{C}^T\mathbf{v}, \tag{72a}$$

$$\dot{\mathbf{u}}^{(1)} = \mathbf{u}^{(2)}, \tag{72b}$$

$$\vdots$$

$$\dot{\mathbf{u}}^{(2k-2)} = \mathbf{u}^{(2k-1)}, \tag{72c}$$

$$\dot{\mathbf{u}}^{(2k-1)} = \left(\mathbf{D}^T\widetilde{\mathbf{M}}\mathbf{D}\right)^{-1}\mathbf{D}^T\left[\widetilde{\mathbf{f}} - \widetilde{\mathbf{M}}\widetilde{\sigma} - \widetilde{\mathbf{M}}\delta^{2k-1}\left(\mathbf{C}^T\mathbf{v}\right)\right], \tag{72d}$$

System (69) is equivalent to a system of DAEs with respect to $2kn$ differential and m algebraic variables. System (72), obtained by application of the projection method to (69), is an ODE system with respect to $n + (2k-1)(n-m)$ variables $(\mathbf{x}, \mathbf{u}^{(1)}, \mathbf{u}^{(2)}, \ldots, \mathbf{u}^{(2k-1)})$. Thus, we have eliminated $(2k-1)m$ differential and m algebraic variables.

Given the solution of equation (71) we can find the values of the Lagrange multipliers as

$$\lambda = \left(\mathbf{C}\widetilde{\mathbf{M}}^{-1}\mathbf{C}^T\right)^{-1}\mathbf{C}\left[\widetilde{\sigma} + \delta^{2k-1}\left(\mathbf{C}^T\mathbf{v}\right) - \widetilde{\mathbf{f}}\right].$$

For equations (72) we can also formulate an analog of the Special case 1 for Hessenberg index-3 DAEs. We repeat the derivations, since we will need the formulas in the example later.

Special case 5. *Some of the algebraic constraints are linear in* $\mathbf{x}$.

The function $\mathbf{h}$ is of the form

$$\begin{aligned} 0 &= \mathbf{C}_1(t)\mathbf{x} + \alpha(t), \\ 0 &= \mathbf{h}_2(t, \mathbf{x}), \end{aligned}$$

where $\mathbf{C}_1$ is an $l \times n$ matrix of maximal row-rank, such that $l \le n$, and $\mathbf{h}_2$ is a $(k-l)$-dimensional vector function which is nonlinear in $\mathbf{x}$. We can represent $\mathbf{x}$ as

$$\mathbf{x} = \mathbf{D}_1\chi + \mathbf{C}_1^T\psi, \tag{73}$$

where $\mathbf{D}_1$ is such that $\mathbf{C}_1\mathbf{D}_1 = 0$, $\chi = (\chi_1, \ldots, \chi_{n-l})^T$, and $\psi = (\psi_1, \ldots, \psi_l)^T$ is given by

$$\psi = -\left(\mathbf{C}_1\mathbf{C}_1^T\right)^{-1}\alpha.$$

Substituting representation (73) into equation (72a), we obtain an equation for $\dot{\chi}$:

$$\dot{\chi} = (\mathbf{D}_1\mathbf{D}_1)^{-1}\mathbf{D}_1^T\left[\mathbf{D}\mathbf{u}^{(1)} + \mathbf{C}^T\mathbf{v} - \dot{\mathbf{D}}_1\chi - \dot{\mathbf{C}}_1\psi\right]. \tag{74}$$

By replacing $\mathbf{x}$ with its representation (73) and equation (72a) with equation (74), we can eliminate l variables from ODEs (72). □

We now demonstrate the application of the projection method to the optimization of a functional with Lagrangian (66).

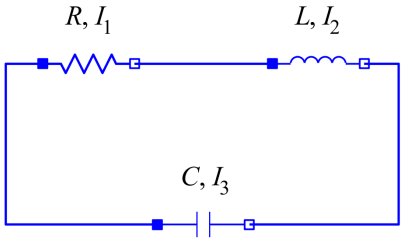


Figure 1. Serial *RLC* circuit.

7. Example: Electric Circuit

We consider an electric circuit consisting of a resistor of resistance R, an inductor of inductance L, and a capacitor of capacitance C connected in series as depicted in Figure 1.

From (66) a Lagrangian for the circuit is given by

$$\mathcal{L} = \frac{1}{2}L\dot{I}_1^2 - \frac{1}{2}\frac{1}{C}I_3^2 + \lambda_1(I_1 - I_2) + \lambda_2(I_2 - I_3), \tag{75a}$$

$$W = \frac{1}{2}R\dot{I}_1^2, \tag{75b}$$

where the constraints, $I_1 = I_2$ and $I_2 = I_3$, represent Kirchhoff's Current Law.

From equations (67) and (68), the necessary condition for a functional with $\mathcal{L}$ and W given by equations (75) takes form of the following DAE system:

$$RI_1 = \lambda_1, \tag{76a}$$

$$L\ddot{I}_2 = -\lambda_1 + \lambda_2, \tag{76b}$$

$$\frac{1}{C}I_3 = -\lambda_2, \tag{76c}$$

$$0 = I_1 - I_2, \tag{76d}$$

$$0 = I_2 - I_3, \tag{76e}$$

where equations (76d) and (76e) are as described in Special case 5. We define:

$$\mathbf{C} = \frac{\partial \mathbf{h}}{\partial \mathbf{x}} = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \end{pmatrix}.$$

It is easy to check that $\mathbf{D} = (1, 1, 1)^T$. Note that the right-hand side of equations (76a), (76b), and (76e) is indeed given by $\mathbf{C}^T\lambda$. From equation (73)

$$I_1 = u^{(1)},\ I_2 = u^{(1)},\ I_3 = u^{(1)}, \tag{77}$$

where $u^{(1)}$ satisfies the system

$$\dot{u}^{(1)} = u^{(2)}, \tag{78a}$$

$$\dot{u}^{(2)} = -\frac{R}{L}u^{(2)} - \frac{1}{LC}u^{(1)} \tag{78b}$$

that follows from equation (72).

Equations (78) are the desired ODEs. After solving them, we find the current using (77).

8. Implementation and Benchmarks

We implemented our algorithm in the computer algebra system Maple (version 16). Table 2 gives some benchmarks for 5 DAEs of index 3 that were created using a conserved

Model	Version	DE	AE	SE	Time
DoublePendulum1	HF	22	7	112	
	PM	14	0	129	0.91s
FourBar	HF	30	11	166	
	PM	16	0	194	13.03s
Pendulum1	HF	12	3	58	
	PM	8	0	67	0.29s
SliderCrank	HF	29	10	215	
	PM	18	0	231	2.15s
TriplePendulum1	HF	32	11	165	
	PM	20	0	199	1.86s

Table 2. Implementation benchmarks

quantities based modeling tool, HLMT [14, 3], and then converted to (mixed) Hessenberg index-3 form. (Such models tend to be more verbose and contain more redundancy than DAEs that were created using other modeling tools.)

There are two rows per model. The first row lists the number of equations in the Hessenberg form that is input to our algorithm. The second row gives the number of equations in the resulting model after applying the projection method, as well as the running time (3 GHz Intel Core 2 Duo) of our implementation.

Three types of equations are counted in each row: the differential equations (DE) and the algebraic equations (AE) together form the *core dynamical system*. The solved equations (SE) are explicit algebraic equations expressing variables that do not appear in the core system exclusively in terms of core system variables. In all 5 examples, the projection method was able to remove all (possibly nonlinear and implicit) algebraic constraints, thanks to Special cases 2 and 3 above.

9. Conclusions

We have derived how the projection method introduced in [16] for mechanical systems can be extended to other types of DAEs. The projection method not necessarily simplifies DAEs better than index reduction alone. However, in some cases the application of the projection method after index reduction can decrease the number of variables and even simplify the reduced index-1 DAE to an ODE. We have described these cases and demonstrated that optimization under constraints is one of them. We have shown the significance of optimization problems in various applications and derived an effective simplification procedure for the optimization of a functional under constraints. As an example, we have formulated a problem of current flowing through an electric circuit as an optimization problem and used it to illustrate the applicability of the projection method. Finally, we have given benchmarks for an implementation of the projection method in Maple.

10. Acknowledgements

The authors gratefully acknowledge the funding of this research by Toyota Motor Engine & Manufacturing North America Inc., and the inspiration from Mr. Akira Ohata, Toyota Motor Corporation.

References

[1] K. Arczewski and W. Blajer, A unified approach to the modeling of holonomic and nonholonomic mechanical systems, *Math. Modeling of Systems* **2** (1996), 157–174.

[2] U. M. Ascher, L. R. Petzold, *Computer Methods for Ordinary Differential Equations and Differential Algebraic Equations*, SIAM, 1998.

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

[4] J.-L. Basdevant, *Variational principles in physics*, Springer, New York, 2007.

[5] L. D. Berkovitz, Variational methods in problems of control and programming, *J. Math. Analysis and Applications* **3** (1961), 145–169.

[6] W. Blajer, A projection method approach to constrained dynamic analysis, *J. Appl. Mech.* **59** (1992), 643–649.

[7] W. Blajer, Projective formulation of Maggi's method for nonholonomic system analysis, *J. Guidance Control Dyn.* **15** (1992), 522–525.

[8] W. Blajer, Elimination of constraint violation and accuracy aspects in numerical simulation of multibody systems, *Multibody System Dynamics*, **7** (2002), 265–284.

[9] P. Blanchard and E. Brüning, *Variational Methods in Mathematical Physics: A Unified Approach*, Springer-Verlag, Berlin, Heidelberg, New York 1992.

[10] K. E. Brenan, S. L. Campbell, L. R. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, SIAM, 1996.

[11] A. N. Kolmagorov, S. V. Fomin, *Elements of the Theory of Functions and Functional Analysis*, Courier Dover Publications, 1999.

[12] S. G. Krantz, H. R. Parks, *The Implicit Function Theorem: History, Theory, and Applications*, Birkhäuser, 2002.

[13] R. K. Nesbet, *Variational Principles and Methods in Theoretical Physics and Chemistry*, Cambridge University Press, Cambridge, 2003.

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

[15] C. M. Roithmayr, *Relating Constrained Motion to Force Through Newton's Second Law*, Ph.D. thesis, Georgia Institute of Technology, 2007.

[16] D. Scott, Can a projection method of obtaining equations of motion compete with Lagrange's equations? *Am. J. Phys* **56** (1988), 451–456.

Initialization of Equation-based Hybrid Models within OpenModelica

Lennart A. Ochel[1], Bernhard Bachmann[1]

[1]Department Mathematics and Engineering, University of Applied Sciences Bielefeld, Germany,
{lennart.ochel,bernhard.bachmann}@fh-bielefeld.de

Abstract

Modelica is a multi-domain object-oriented modeling language designed for time-dependent systems. The time-dependent part is usually described with "ordinary differential equations". In addition to that, it is possible to express algebraic and difference equations. As a result a Modelica model will be merged to a hybrid differential algebraic equation system.

The initialization process is prior to each simulation and must therefore be solved before any simulation can be started. Modelica provides high-level features to describe the initialization problem. This leads often into various problems. The initialization is usually a system-level issue. Therefore, high knowledge about the system is necessary.

In OpenModelica two major methods are implemented to solve the initialization problem. Both methods are totally different and are used for different initialization issues. Both methods will be discussed within this paper.

Keywords initialization, hybrid models, homotopy, start value, OpenModelica

1. Introduction

Primary linguistic constructs of Modelica to specify the initialization are initial equations and initial algorithms. It is possible that the initialization is not unique, even if initial conditions are fully specified. This means that the number of unknowns (in the case of initialization) is equal to the number of initial conditions. The non-uniqueness is caused by nonlinearities.

As a result, the modeler is not able to control the initialization completely, if just initial equations and initial algorithms are used. To retain control of the initialization, additional linguistic devices such as the homotopy operator and variable attributes (e.g. start values) are available in Modelica. Since homotopy is quite an advanced feature, the modeler may prefer the use of start values.

The influence of start values is in most implementations rather small. They are mostly used as an initial guess for nonlinear systems. Therefore, it is necessary to provide start values for variables, which are involved in these nonlinear systems. Moreover, it is important to know about dependencies of a given model and about suitable start values for just these variables.

This work will show how it is possible to increase the influence of start values during initialization a lot and to provide the modeler full control on the solution for his initialization problem. This is done by the first major method based on numerical algorithms and an extension called "Start Value Homotopy".

The second major approach is based on symbolical transformations. It creates a complete dependence graph for the initialization. Hence, the initial equations are sorted and transformed to a system that can be explicitly evaluated, except involved algebraic loops. This leads to a much faster and more accurate solution compared to the numeric approach.

2. Modelica Constructs for Initialization

Modelica contains several language constructs that influence the initialization (see [1]). These constructs can be categorized like follows:

Firstly, there are initial equations and initial algorithms that declare additional equations and algorithms to the time-dependent system. These special equations and algorithms are only active during initialization and are added to the simulation equations. In case of a hybrid model, when-equations are only considered, if activated using the `initial()` operator.

attribute	Real	Integer	Boolean	String
start	X	X	X	X
fixed	X	X	X	
min/max	X	X		
nominal	X			

Table 2.1. Some available variable attributes that can be important for the initialization process.

Secondly, Modelica provides the possibility to define variable attributes for each variable. What attributes are actually available depend on their type as listed above.

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, 19 April, 2013, University of Nottingham, UK.

EOOLT 2013 website:
http://www.eoolt.org/2013/

These attributes affect the initial solution either primarily or secondarily. The usage of the start value depends on the variable type (continuous/discrete) and the fixed attribute.

The start value is used as an initial guess, if `fixed=false`. Otherwise, the start value implicitly generates an initial equation `v=start(v)` for a continuous variable and `pre(v)=start(v)` for a discrete variable.

v(start=v^{start})	fixed=true	fixed=false
type of v: continuous	initial equation: $v = v^{start}$	initial guess of v
type of v: discrete	initial equation: $pre(v) = v^{start}$	initial guess of $pre(v)$

Table 2.2. Interpretation of start attribute depending on fixed attribute and variable kind.

For this reason, it is important to know about the variable type, if the initial condition should be described using these attributes. In Modelica it is not necessary to explicitly declare a variable as discrete or not. This will be automatically detected by a Modelica tool. Due to that reason, it is possible that a variable type is changed in a higher hierarchical component. This can directly affect the corresponding initial equation as introduced above and has to be taken into account during the modeling process.

Using the min and max attribute may restrict the solution space. This can be utilized, for example, to remove physical impractical solutions from the solution space.

The nominal value can be used to setup scaling coefficients. This is the only attribute with no default value. If a Modelica tool detects that there is no nominal value, it can perform some analysis to determine some suitable nominal values by itself.

Finally, Modelica provides the homotopy operator [1] that gives the possibility to formulate actual and simplified expressions for equations. This concept is utilized to improve the convergence properties of the nonlinear iterative solver. A Modelica tool is supposed to introduce the expression (2.1) with a homotopy parameter λ going from 0 to 1.

$$actual \cdot \lambda + simplified \cdot (1 - \lambda) \tag{2.1}$$

3. Mathematical Representation

The mathematical representation will be kept as simple as possible in order to focus on the important aspects. Therefore, some vectors (e.g. inputs) with no effect to the described issues are ignored. With this limitation a hybrid Modelica model can be represented using the following notation described in Table 3.1.

symbol	description
$\underline{x}(t)$	vector of all states
$\underline{\dot{x}}(t)$	vector of all differentiated states
$\underline{y}(t)$	vector of all continuous algebraic variables
$\underline{d}(t)$	vector of all discrete variables
$\underline{p}$	vector of all parameters

Table 3.1. Used symbols for mathematical representation.

The variables $\dot{x}$ and y are unknowns during simulation and are combined to $z(t)$.

$$\underline{z}(t) := \left(\underline{\dot{x}}(t) \quad \underline{y}(t) \quad \underline{d}(t)\right)^{\mathsf{T}} \tag{3.1}$$

The simulation equations can be written as given below.

$$\begin{aligned} f_1\left(\underline{x}(t), \underline{\dot{x}}(t), \underline{y}(t), \underline{d}(t), \underline{d}^{pre}(t), \underline{p}, t\right) &= 0 \\ \vdots \\ f_n\left(\underline{x}(t), \underline{\dot{x}}(t), \underline{y}(t), \underline{d}(t), \underline{d}^{pre}(t), \underline{p}, t\right) &= 0 \end{aligned} \tag{3.2}$$

To efficiently evaluate $\underline{z}(t)$, equations (3.2) are transformed to explicit state space representation (3.3).

$$\underline{z}(t) = \underline{g}\left(\underline{x}(t), \underline{d}^{pre}(t), \underline{p}, t\right) \tag{3.3}$$

The representation (3.3) is not always achievable in an analytic form. But, due to the implicit function theorem such a representation exists, if the corresponding Jacobian is regular. A Modelica tool typically performs the following transformation steps, in order to increase the efficiency. This mathematical representation and the transformation steps of a Modelica model are illustrated using the following example.

```
      model MathRep
x     | Real x1(start=2.0, fixed=true),
      |      x2(start=4);
y     | Real y1, y2, y3(start=-1.5);
d     | Real d1;
      initial equation
h1    | pre(d1) = -0.5 + y1;
      equation
f1    | 0 = -y2 + sin(y3);
f2    | der(x1) = sqrt(x1) + time - d1;
f3    | 0 = x1 + y2 + y3 + 1;
f4    | 0 = x1 + y1 + x1*y1;
      | when {initial(), sample(0.1, 0.1)}
f5    | then
      |   d1 = pre(d1) - y1 + y2;
      | end when;
f6    | der(x2) = x1 + y1;
      end MathRep;
```

Listing 3.1. Example model "MathRep".

Based on a bipartite graph representation of the equation system (see Figure 3.1) a matching algorithm assigns each variable exactly one equation [4].

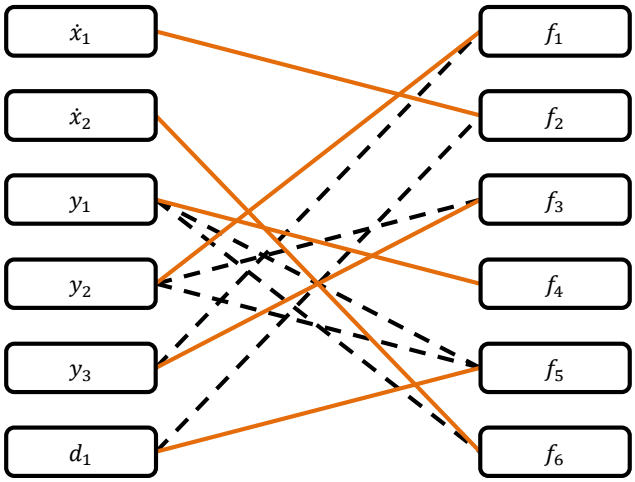


Figure 3.1. Bipartite graph representation and result of the matching for the time-dependent system of example model "MathRep".

The next step is to determine a recursive evaluation order. Due to algebraic loops the result is a block-lower-triangular form (see (3.4)). This is done by determine the strong components using methods like Tarjan's algorithm [5].

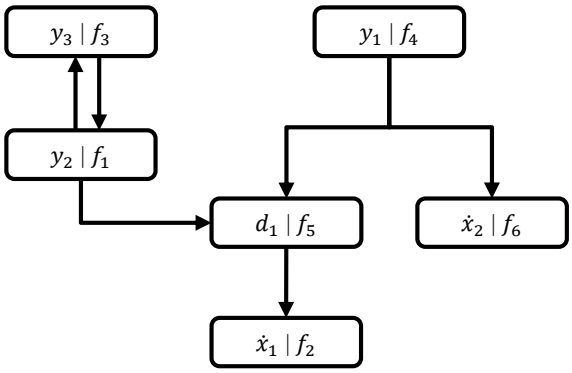


Figure 3.2. Directed graph representation and result of the sorting for the example model "MathRep".

$$\begin{array}{c} \\ f_3 \\ f_1 \\ f_4 \\ f_5 \\ f_6 \\ f_2 \end{array} \begin{array}{c} \begin{array}{cccccc} y_3 & y_2 & y_1 & d_1 & \dot{x}_2 & \dot{x}_1 \end{array} \\ \begin{pmatrix} \boxed{1} & \boxed{1} & 0 & 0 & 0 & 0 \\ \boxed{1} & \boxed{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & \boxed{1} & 0 & 0 & 0 \\ 0 & 1 & 1 & \boxed{1} & 0 & 0 \\ 0 & 0 & 1 & 0 & \boxed{1} & 0 \\ 0 & 0 & 0 & 1 & 0 & \boxed{1} \end{pmatrix} \end{array} \tag{3.4}$$

More efficiency is gained by so-called tearing algorithms [6], [7], which further reduce the size of algebraic loops.

The principle of the simulation is based on the concept that at a given point in time, especially at the initial time t_0, the states, left limit of discrete variables as well as all free parameters are known. A basic principle is that parameters are constant during simulation and set by the user. Modelica provides via the attribute fixed the possibility that parameters are "free" during the initialization process. The initialization process calculates all needed variables at t_0.

The vector $\underline{\omega}(t_0)$ contains these additional unknowns during initialization. It consists of all states $\underline{x}(t_0)$, all unfixed parameters $\underline{p}^{free}$ and the left limit of all discrete variables $\underline{d}^{pre}(t_0)$.

$$\underline{\omega}(t_0) := \left(\underline{x}(t_0) \quad \underline{p}^{free} \quad \underline{d}^{pre}(t_0)\right)^{\mathrm{T}} \tag{3.5}$$

In order to describe initial conditions additional equations are needed. Mathematically, it is helpful to define the same number of equations than unknowns. If less or more equations than unknowns are given, special treatments are necessary and are described further down.

The initial conditions can be written as illustrated below.

$$\begin{aligned} h_1\left(\underline{x}(t_0), \underline{\dot{x}}(t_0), \underline{y}(t_0), \underline{d}(t_0), \underline{d}^{pre}(t_0), \underline{p}, t_0\right) &= 0 \\ \vdots & \\ h_m\left(\underline{x}(t_0), \underline{\dot{x}}(t_0), \underline{y}(t_0), \underline{d}(t_0), \underline{d}^{pre}(t_0), \underline{p}, t_0\right) &= 0 \end{aligned} \tag{3.6}$$

The goal of the initialization is to determine valid values for $\underline{\omega}(t_0)$. Example models will be discussed within the following sections.

4. Numeric Approach

The numeric approach is the first of two major approaches in OpenModelica to solve the initialization problem. The basic version was already presented in [2] and has been successfully applied in [3].

$$\min_{\underline{\omega}(t_0)} \phi\left(\underline{\omega}(t_0), \underline{z}(t_0), \underline{p}^{fixed}, t_0\right) \rightarrow 0$$

s.t.

$$\begin{aligned} \underline{z}(t_0) &= \underline{g}\left(\underline{\omega}(t_0), \underline{p}^{fixed}, t_0\right) \\ \underline{\omega}^{min} &\leq \underline{\omega}(t_0) \leq \underline{\omega}^{max} \end{aligned} \tag{4.1}$$

with

$$\phi(.) = \sum_i h_i\left(\underline{\omega}(t_0), \underline{z}(t_0), \underline{p}^{fixed}, t_0\right)^2$$

The basic idea is to transform the initialization problem into an optimization problem with an optimum that is equal to the initial solution. This is done by interpreting all initial equations as residual ones, which are squared and accumulated to the objective function ϕ. It becomes zero if all equations are satisfied.

4.1 Under/Over-Determined Systems

Often, the modeler misses to fully describe initial conditions to a Modelica model. For the initialization process of such a model it is crucial to provide a determined system. The numeric approach adds additional initial equations by numeric model analysis. Therefore, the Jacobian $\frac{\partial \underline{h}}{\partial \underline{\omega}}\left(\underline{\omega}^{start}\right)$ is numerically approximated and the maximum of the absolute values is selected for each column. Until the initial system is determined the fixed attribute of the variable related to the smallest values are set to true. The heuristics is based on the fact that these variables have the least influence on the initial system near to the start values $\underline{\omega}^{start}$. Therefore, additional initial equations are introduced, which hopefully provide a solvable initial system.

In some cases, Modelica models can be over-determined, e.g. when initial equations are formulated locally in sub-components. If the over-all initial system consists of redundant equations, however fully determines the solution, the current numeric approach can deal with such systems by design [2], [3].

4.2 Scaling

Many problems are hard to solve, since the values of the involved variables are of different magnitudes. To handle such systems variables and equations need to be scaled. In general the nominal attribute $\underline{\omega}^{nom}$ is used for scaling and should be provided by the modeler. By this, the scaling of the variable is straightforward like shown in (4.2).

$$\omega_i^{scaled} := (\omega_i^{nom})^{-1} \cdot \omega_i \tag{4.2}$$

Since the nominal attribute is not available for equations suitable scaling coefficients have to be calculated using differential error analysis. Therefore, each initial equation is approximated by first order taylor expansion. The corresponding scaling factor for each equation is constructed as illustrated in (4.3).

$$\begin{aligned} h(\underline{\omega}) \approx h(\underline{\omega}^{nom}) &+ \underbrace{\frac{\partial h(\underline{\omega}^{nom})}{\partial \omega_1} \cdot \omega_1^{nom}}_{\hat{s}_1^{h_j}} \cdot \frac{\omega_1 - \omega_1^{nom}}{\omega_1^{nom}} \\ &+ \cdots + \underbrace{\frac{\partial h(\underline{\omega}^{nom})}{\partial \omega_n} \cdot \omega_n^{nom}}_{\hat{s}_n^{h_j}} \cdot \frac{\omega_n - \omega_n^{nom}}{\omega_n^{nom}} \\ \tilde{s}_i^h := \begin{cases} |\hat{s}_i^h| & if\ \varepsilon < |\hat{s}_i^h| \\ 1 & else \end{cases} &\ for\ i = 1 \dots n \\ s^h := (max\{\tilde{s}_1^h; \dots; \tilde{s}_n^h\})^{-1} & \\ h^{scaled} := (s^h)^{-1} \cdot h & \end{aligned} \tag{4.3}$$

4.3 Start Value Homotopy

Start Value Homotopy is the name of an extension to the basic numeric approach within OpenModelica. This method uses a different objective function ϕ.

$$\begin{aligned} \phi(.) &= (1-\lambda) \cdot \phi_0 + \lambda \cdot \phi_1 \\ \lambda &\in [0;1] \subset R \end{aligned} \tag{4.4}$$

with

$$\begin{aligned} \phi_0(.) &= \sum_{\forall v} (v - v^{start})^2 \\ \phi_1(.) &= \sum_i h_i\left(\underline{\omega}(t_0), \underline{z}(t_0), \underline{p}^{fixed}, t_0\right)^2 \end{aligned} \tag{4.5}$$

The new objective function is a combination of two sub-objective functions ϕ_0 and ϕ_1. Both are weighted with the homotopy parameter λ (see (4.4)). In the beginning λ is equal to zero and gets increased during the initialization phase until it is one. As a result the initialization algorithm considers in the beginning just ϕ_0 and in the end ϕ_1.

Equation (4.5) shows these sub-objective functions. ϕ_1 is the same function as the objective function of the basic approach. This is quite reasonable, since it should solve the same problem. ϕ_0 is a much simpler function which is based on all explicitly given start attributes.

```
      model forest
x1      Real foxes;
x2      Real rabbits;
y1      Real population(start=350);
y2      Real value;
p       […] // used parameters
      initial equation
h1      der(foxes) = 20;
h2      value      = 11000;
      equation
f1      der(rabbits) = rabbits*g_r -
          rabbits*foxes*d_rf;
f2      der(foxes)   = -foxes*d_f +
          rabbits*foxes*d_rf*g_fr;
f3      population   = foxes+rabbits;
f4      value        = priceFox*foxes +
          priceRabbit*rabbits;
      end forest;
```

Listing 4.1. Example model "forest".

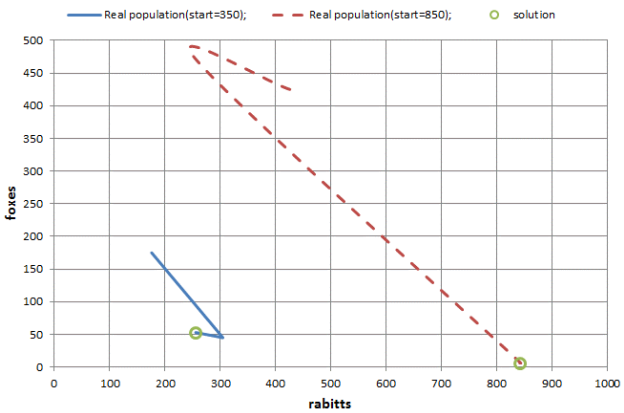


Figure 4.1. Paths within the state space for both initial solutions of the example model "forest".

Figure 4.1 shows the iteration paths for two versions of the example model "forest" from Listing 4.1. The difference between both versions is the start value of `population`. The solid line and the dashed line represent the iteration paths for `population(start=350)` and `population(start=850)`, respectively. The two small circles are exact solutions of the related nonlinear equation system.

The population equation f_3 equally involves both states. In the beginning of the initialization process λ is set to zero, which yields that the objective function consists just of ϕ_0 and considers therefore only the start value of the population. As a result both states are set to half of that start value in the first iteration. As expected both paths continue straight to the corresponding solution.

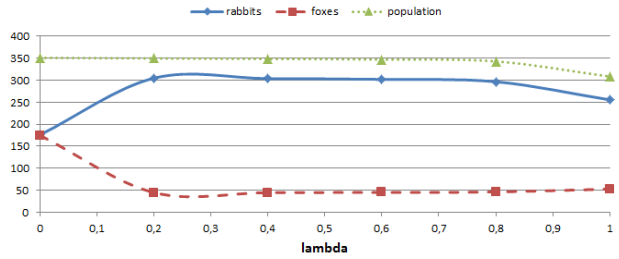


Figure 4.2. Homotopy path with population(start=350).

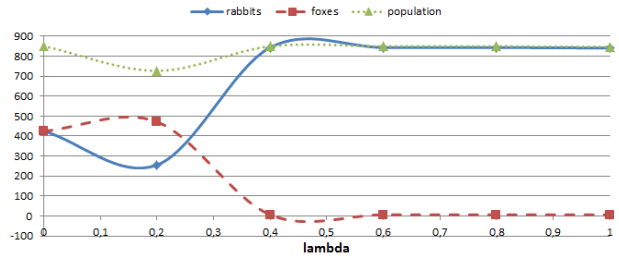


Figure 4.3. Homotopy path with population(start=850).

This numeric approach including the Start Value Homotopy feature has been used as the default initialization method since the end of 2011.

5. Symbolic Approach

This chapter describes the newly developed symbolic approach for solving the initialization problem, which has been investigated and implemented since the beginning of 2012. The main idea behind this approach is the use of symbolic transformation algorithms (matching, sorting, tearing, etc.) that have been sketched in chapter 3 and are already available in the OpenModelica environment. Using the dependence graph with respect to the initialization problem the corresponding equation system is transformed to a block-lower-triangular form. Involved algebraic loops are further reduced by using tearing techniques.

So far, this approach can only be used for determined and under-determined initialization problems. In case of an under-determined initialization problem additional equations are added automatically, based on symbolic model analysis (see section 5.2), until the number of unknowns and equations match. The resulting initialization equation system can finally be described by (5.1) and needs to be solved for the unknowns given by (5.2) during initialization.

$$\begin{aligned} 0 &= f_1\left(\underline{\omega}(t_0), \underline{z}(t_0), \underline{p}^{fixed}, t_0\right) \\ &\vdots \\ 0 &= f_n\left(\underline{\omega}(t_0), \underline{z}(t_0), \underline{p}^{fixed}, t_0\right) \\ 0 &= h_1\left(\underline{\omega}(t_0), \underline{z}(t_0), \underline{p}^{fixed}, t_0\right) \\ &\vdots \\ 0 &= h_m\left(\underline{\omega}(t_0), \underline{z}(t_0), \underline{p}^{fixed}, t_0\right) \end{aligned} \tag{5.1}$$

$$\begin{aligned} \underline{z}(t_0) &:= \left(\dot{\underline{x}}(t_0) \quad \underline{y}(t_0) \quad \underline{d}(t_0)\right)^{\mathsf{T}} \\ \underline{\omega}(t_0) &:= \left(\underline{x}(t_0) \quad \underline{p}^{free} \quad \underline{d}^{pre}(t_0)\right)^{\mathsf{T}} \end{aligned} \tag{5.2}$$

5.1 Dependence Graph

The solution process is firstly described on the example model "forest" that only involves fixed parameters and continuous variables. The result of the matching algorithm performed on the corresponding bipartite graph is presented in Figure 5.1.

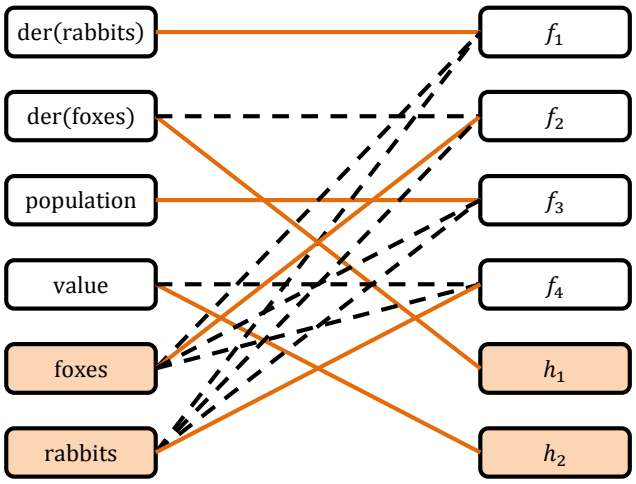


Figure 5.1. Bipartite graph representation and result of the matching for the example model "forest".

Tarjan's algorithm produces the dependence graph below.

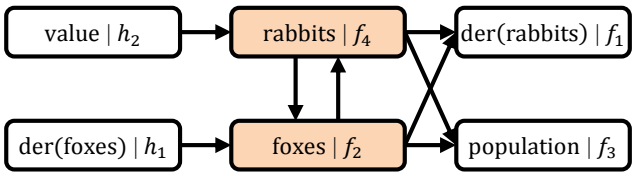


Figure 5.2. Dependence graph of the initialization problem for the example model "forest".

Processing the sorted equation system means that at first the variables $value$ and $der(foxes)$ are calculated in equation h_1 and h_2, respectively. Then, the variables $rabbits$ and $foxes$ are calculated simultaneously through the equation system f_2 and f_4. Finally, the variables $der(rabbits)$ and $population$ are determined using f_1 and f_3, respectively. From this evaluation order it is immediately clear that a starting value for the population will have no influence on the initialization process. This behavior was different when using the Start Value Homotopy approach. In addition, this initialization process is much faster than using the numeric approach. But, the modeler has less influence on the final result of the initialization.

5.2 Under-Determined Systems

As described in section 4.1 it is important to provide a determined equation system for initialization. Since it can happen that the initial conditions are not fully specified, additional equations have to be added to the initialization problem. The symbolic initialization approach in OpenModelica automatically augments these equations based on symbolic model analysis. Additional equations are determined by setting the fixed attribute to true of such components of $\underline{\omega}$ that so far cannot be determined from the initial equation system.

This information can be extracted by processing the sparsity pattern for the Jacobian $\frac{\partial \underline{h}}{\partial \underline{\omega}}$, which can be seen as the collapsed dependence graph of $\underline{\omega}$. If any component of $\underline{\omega}$ cannot be calculated from the initial equation system the whole column is zero. This symbolic method does not depend on $\underline{\omega}^{start}$ as well as other numerical issues compared to the numeric approach.

5.3 Scaling

Using the symbolic approach the initialization problem is transformed to a block-lower-triangular form. As motivated earlier scaling is necessary for finding accurate solutions even when values of variables are of different magnitudes. Same principles are used as described in section 4.2 but only applied on algebraic loops. This reduces enormously the number of Jacobian elements to be calculated.

5.4 Hybrid Models

As mentioned in chapter 3, it is necessary to initialize the continuous as well as the discrete part of a Modelica model. Using the numerical approach the complete hybrid equation system necessary for simulation is considered as constraint for the optimization process. This often leads to a high-dimensional nonlinear optimization problem involving real and discrete variables. Such optimization problems are numerically hard to solve. This issue can be avoided by symbolic transformation steps, which are also used for the simulation.

In the following, the example model "MathRep" from Listing 3.1 will be further analyzed with respect to the initialization apporach. This model contains two states and one discrete variable. Therefore, $\underline{\omega}$ becomes the following:

$$\underline{\omega}(t_0) := \begin{pmatrix} x_1(t_0) & x_2(t_0) & d_1^{pre}(t_0) \end{pmatrix}^{\mathsf{T}} \tag{5.3}$$

Because there are three variables that need to be initialized, it would be necessary that there are also three initial conditions given. The model contains just the initial conditions h_1 as explicitly declared and h_2 (see (5.4)) as implicitly declared. Therefore, the corresponding dependencies from the three unknowns are analyzed and the additional equation h_3 is automatically derived.

$$\begin{array}{l|ll} h_2 & x_1 = x_1^{start} & \text{(implicitly declared)} \\ h_3 & x_2 = x_2^{start} & \text{(automatically declared)} \end{array} \tag{5.4}$$

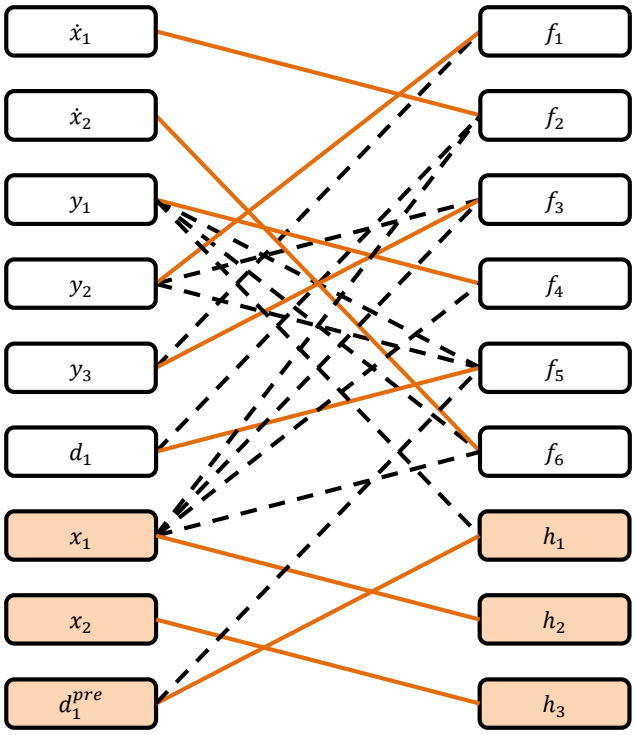


Figure 5.3. Bipartite graph representation and result of the matching for the initial system of example model "MathRep".

Adding this additional information (5.4) to the initial equation system the bipartite graph from Figure 5.3 is generated. Utilizing Tarjan's algorithm the dependence graph presented in Figure 5.4 is produced.

Due to this symbolic approach, the original high-dimensional nonlinear optimization problem involving real and discrete variables is to a large extent reduced to block-lower triangular form.

If corresponding algebraic loops still include real and discrete variables further techniques need to be applied in order to solve these equations. In some cases OpenModelica's tearing heuristic [7] eliminates involved

discrete variables. Same applies, if the involved variables are of boolean or integer type.

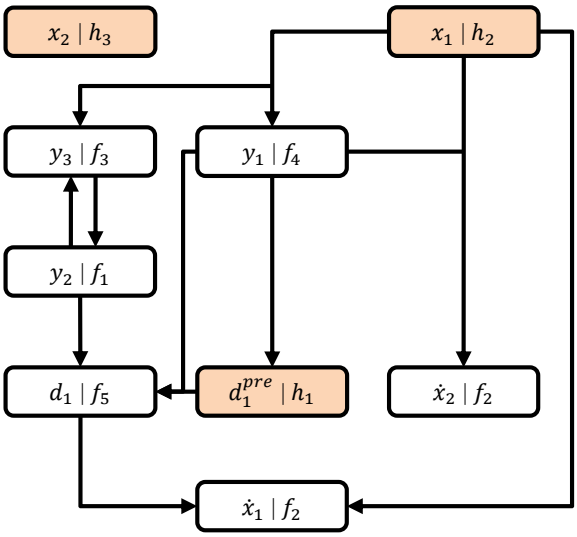


Figure 5.4. Directed graph representation and result of the sorting for the initial system of example model "MathRep".

6. Conclusions and Future Work

This paper describes the principles implemented in the OpenModelica environment, which are utilized to initialize complex hybrid Modelica models. Two major methods, the numeric and symbolic approach, are discussed in detail and advantages and disadvantages have been pointed out.

The numeric approach can deal with over-determined systems and has been successfully applied in [3]. Furthermore, this approach has been extended by the Start Value Homotopy method, which gives the modeler more control on the initialization process.

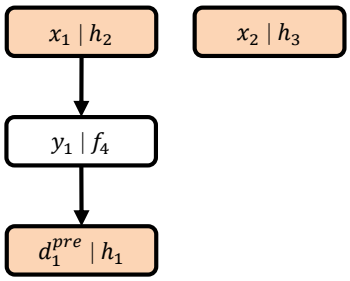


Figure 6.1. Reduced directed graph representation of the initialization problem for the example model "MathRep".

The symbolic approach outperforms the numeric treatment of the initialization problem with respect to performance and solvability in case of large and hybrid systems. With the numeric approach it was so far not possible to initialize the bigger part of model examples in the Modelica Standard Library (MSL). Today, most of MSL examples are initialized efficiently using the symbolic approach.

In case of under-determined initialization problems both approaches introduce additional equations, based on model analysis, in order to generate determined initial systems.

In the future, the two approaches will be more enhanced within the OpenModelica environment. The dependence graph achieved by the symbolic approach can be reduced to represent only the information necessary for determining the initial unknown vector $\underline{\omega}$ (see Figure 6.1 in comparison to Figure 5.4).

Up to now, the Start Value Homotopy method considers all explicitly given start values, which might be not desirable within an object-oriented Modelica drag-and-drop environment. This should be improved by introducing a special Start Value Homotopy annotation keyword. In addition, the Start Value Homotopy feature as well as methods for over-determined systems will be further investigated in order to be integrated into the symbolic approach.

References

[1] Modelica Association, *Modelica® - A Unified Object-Oriented Language for Systems Modeling - Language Specification - Version 3.3*, 2012

[2] Bernhard Bachmann, et.al., *Robust Initialization of Differential Algebraic Equations.* Modelica'2006 Proceedings - Volume 2, pp. 607, 2006.

[3] Francesco Casella, et.al., *Overdetermined Steady-State Initialization Problems in Object-Oriented Fluid System Models.* Modelica'2008 Proceedings - Volume 1, pp. 311, 2008.

[4] Jens Frenkel, et.al., *Survey of appropriate matching algorithms for large scale systems of differential algebraic equations.* Modelica'2012 Proceedings, 2012.

[5] Robert Tarjan, *Depth-first search and linear graph algorithms.* SIAM Journal on Computing, Vol. 1, No. 2, 1972.

[6] Hilding Elmqvist and Martin Otter, *Methods for Tearing Systems of Equations in Object-Oriented Modeling.* Proceedings of the Conference on Modeling and Simulation, eds. Guasch and Huber, pp. 326-332., 1994.

[7] Emanuele Carpanzano, *Order reduction of General Nonlinear DAE Systems by Automatic Tearing*, Mathematical and Computer Modeling of Dynamical Systems. Vol. 6 No. 2, pp. 145-168, 2000.

Session V: Other Topics

Tool Demonstration Abstract: OpenModelica and CasADi for Model-Based Dynamic Optimization

Alachew Shitahun[1] Vitalij Ruge[2] Mahder Gebremedhin[1] Bernhard Bachmann[2]
Lars Eriksson[3] Joel Andersson[4] Moritz Diehl[4] Peter Fritzson[1]

[1]Department of Computer and Information Science, Linkoping University, Sweden,
(alash325@student.liu.se, {mahder.gebremedhin, peter.fritzson}@liu.se)
[2]Department of Mathematics and Engineering, University of Applied Sciences, Germany,
{vitalij.ruge, bernhard.bachmann}@fh-bielefeld.de
[3]Department of Electrical Engineering, Linköping University, Sweden, lars.eriksson@liu.se
[4]Department of Electrical Engineering and Optimization in Engineering Center (OPTEC), K.U. Leuven, Belgium,
{joel.andersson, moritz.diehl}@esat.kuleuven.be

Abstract

This paper demonstrates model-based dynamic optimization through the coupling of two open source tools: OpenModelica, which is a Modelica-based modeling and simulation platform, and CasADi, a framework for numerical optimization. The coupling uses a standardized XML format for exchange of differential-algebraic equations (DAE) models. OpenModelica supports export of models written in Modelica and the Optimica language extension using this XML format, while CasADi supports import of models represented in this format. This allows users to define optimal control problems (OCP) using Modelica and Optimica specifications, and solve the underlying model formulation using a range of optimization methods, including direct collocation and direct multiple shooting. The proposed solution has been tested on several industrially relevant optimal control problems, including a diesel-electric power train, a free-floating robot, and a stirred-tank.

Keywords Model-Based Optimization, OpenModelica, Dynamic Optimization, Modelica, CasADi

1. Introduction

During the last decade, nonlinear model predictive control (NMPC) and nonlinear optimal control problems (NOCP) based on differential-algebraic equations (DAE) have had a significant impact in the industrial community, particularly in the control engineering area [1, 2]. State-of-the-art methods are using numerical algorithms for dynamic optimization based on direct multiple shooting [3] or collocation algorithms [1].

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, 19 April, 2013, University of Nottingham, UK.

Equation-based, object-oriented modeling languages such as Modelica [4] have become increasingly used for industrial applications. These languages enable users to conveniently model large-scale physical systems described by differential, algebraic, and discrete equations, primarily with the goal of performing virtual experiments (simulation) on these systems, but recently also optimization.

Due to the influence of such equation-based, object-oriented modeling languages in the industrial community, there have been results of an effort where model-based dynamic optimization has been done by coupling of OpenModelica and CasADi [5], which is a numerical algorithmic tool. The problem formulation and modeling is done in Modelica [6] including the Optimica language extensions described in [7] using the OpenModelica graphical editor (OMEdit) and then export the model and optimization descriptions into an XML format (Figure 1). This enables users to formulate and use model-based NOCP that can be solved by CasADi, see also [12].

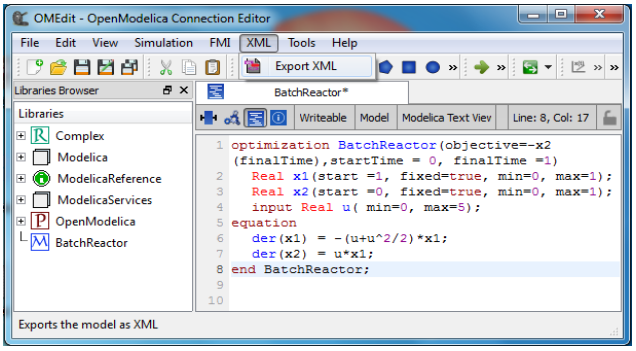


Figure 1: Modeling NOCP using the OpenModelica Graphical and Textual Editor

2. OpenModelica Compiler and CasADi

The OpenModelica compiler front-end has been extended to support the Optimica language extensions. In addition, the OpenModelica compiler has recently been extended with XML export of models [8] based on the XML format

defined in [9], also including the Optimica extensions. In essence, the task of OpenModelica is to read the Modelica and Optimica source code, translate into a flat model description and then export the model and optimization descriptions into an XML format which can be solved by a numerical algorithmic tool.

The exported XML document can then be imported to CasADi tool. The tool supports symbolic import of OCPs via this XML format. This OCP can then be transcribed into a nonlinear programming problem (NLP) using the approach outlined in [10] of Section 5, and solved with one of CasADi's interfaced NLP solvers. The complete tool chain is visualized in Figure 2.

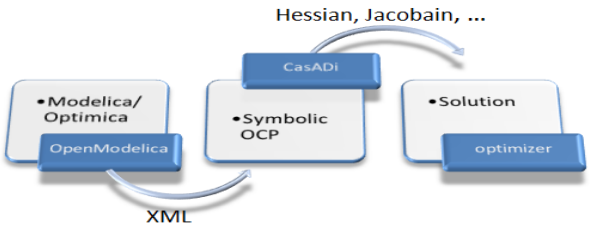


Figure 2: Optimization tool chain for OpenModelica and CasADi

3. Demonstration

In order to present the proposed concept, we demonstrate the solution of an industrial-relevant optimal control problem of diesel engine model. The Diesel-electric powertrain model presented in [11, 10] is a nonlinear mean value engine model (MVEM) containing four states and two control inputs. The problem solved here is a minimum fuel problem for a transient from idle to 170 kW, for an end time of 0.5 s. The control and state trajectories of the optimization results are shown in Figure 3 and Figure 4 respectively.

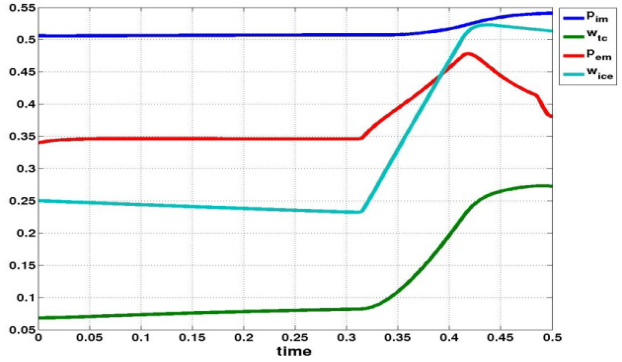


Figure 3: Optimization results of the Diesel-electric powertrain model– state variables**.**

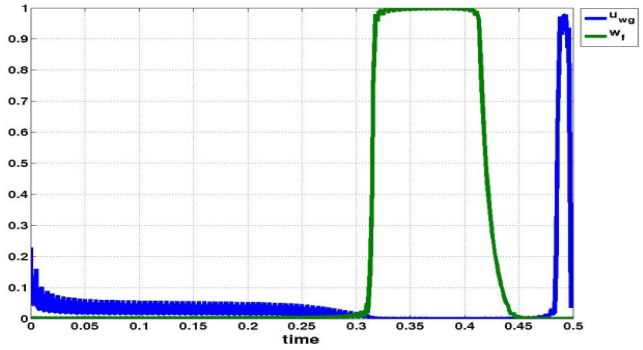


Figure 4: Optimization results the Diesel-electric powertrain model – control variables.

Acknowledgements

This work has been partially supported by Serc, by SSF in the EDOp project and by Vinnova as well as the German Ministry BMBF (BMBF Förderkennzeichen: 01IS09029C) in the ITEA2 OPENPROD project and in the ITEA2 MODRIO project. The Open Source Modelica Consortium supports the OpenModelica work.

J. Andersson and M. Diehl acknowledge support by PFV/10/002 OPTEC, GOA/10/09 and GOA/10/11, FWO G.0320.08, G.0377.09, SBO LeCoPro; Belspo IUAP P7 DYSCO, FP7-EMBOCON (ICT-248940), SADCO (MC ITN-264735), ERC ST HIGHWIND (259 166), Eurostars SMART, vicerp, ACCM.

References

[1] Biegler, L.T. Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes. s.l.: Society for Industrial Mathematics, 2010.

[2] Tamimi, J. and Li, P. A combined approach to nonlinear model predictive control of fast systems. *Journal of Process Control*, 20: 1092–1102, 2010.

[3] Bock, H.G. and Plitt K.J. A multiple shooting algorithm for direct solution of optimal control problems. In Proc. of 9th IFAC World Congress, Budapest, pp: 243-247, 1984

[4] Fritzson, P. Principles of Object-Oriented Modeling and Simulation with Modelica, Wiley-IEEE Press, 2003.

[5] Andersson, J., Åkesson, J. and Diehl, M. CasADi -- A symbolic package for automatic differentiation and optimal control, Recent Advances in Algorithmic Differentiation, Lecture Notes in Computational Science and Engineering Volume 87: 297-307, 2012.

[6] Modelica Association. The Modelica Language Specification Version 3.2, March 24th 2010. Available at: http://www.modelica.org/(Accessed 8 December 2012).

[7] Akesson, J. Optimica—An Extension of Modelica Supporting Dynamic Optimization. In Proc. of 6th International Modelica Conference, March 3-4, 2008.

[8] Shitahun, A.. Template Based XML and Modelica Unparsers in OpenModelica. Master thesis. Linköping University, August 30, 2012

[9] Parrotto, R., Åkesson, J. and Casella, F. An XML representation of DAE systems obtained from continuous-time Modelica models. In Proc. of EOOLT 2010, September 2010. www.eoolt.org

[10] Bachmann, B., et al. Parallel Multiple-Shooting and Collocation Optimization with OpenModelica. In Proc. 9th Int. Modelica Conf. Munich, Germany, Sept 3-5, 2012.

[11] Sivertsson, M. and Eriksson, L. Time and Fuel Optimal Power Response of a Diesel-Electric Powertrain. E-CoSM'12 – IFAC Workshop on Engine and Powertrain Control, Simulation and Modeling, 2012.

[12] Shitahun, A., Ruge, V., Gebremedhin, M., Bachmann, B., Eriksson, L., Andersson, J., Diehl, M., Fritzson, P. Model-Based Optimization with OpenModelica and CasADi. Accepted to IFAC Sept. 2013, Tokyo, Jan. 2013.

Tool Demonstration Abstract: OpenModelica Graphical Editor and Debugger

Adeel Asghar Peter Fritzson

Department of Computer and Information Science, Linköping University, Sweden,
{adeel.asghar,peter.fritzson}@liu.se

Abstract

This paper demonstrates the OpenModelica graphic editor for easy-to-use graphic modeling of Modelica models and the Modelica debugger.

The graphic editor aims at providing a user friendly open source Modelica modeling graphical user interface since most of the already existing open source tools were either textual or not so user friendly. The target audiences for the tool are the Modelica users who want easy-to-use model creation, library browsing, connection editing, simulation of models, plotting results and visualization of components.

Modeling errors and problems are often hard to find because of the high abstraction level of languages like Modelica. Models containing functions with huge algorithm sections increase the need for run-time debugging. The OpenModelica debugger provides a debugging of such models. The debugger currently supports debugging of algorithmic code. The debugger uses the Gnu low-level C-language debugger (GDB) for low-level manipulation and control of the executing program during debugging.

Keywords Graphic editor, Connection Diagrams, Run-time Debugging, Modeling and Simulation, Algorithmic code.

1. OpenModelica Graphical Editor

The OpenModelica graphical editor (OMEdit) is an integrated development environment for Modelica where users can model, simulate and plot their physical systems designs.

It supports the Modelica standard library 3.2.1 through the graphical annotations. It is based on OpenModelica's interactive scripting environment. The scripting environment is part of the OpenModelica compiler. The communication with the scripting environment is carried out through the CORBA interface. During the communication the OpenModelica compiler acts as server while the graphic editor acts as a client.

OMEdit provides user friendly features like;

- Modeling – easy Modelica model creation.
- Browsing – Modelica standard library browsing.
- Component interfaces – smart connection editing for drawing and editing connections between model interfaces.
- Simulation subsystem – subsystem for running simulations and specifying simulation parameters start and stop time, etc.
- Plotting – interface to plot variables from simulated models.

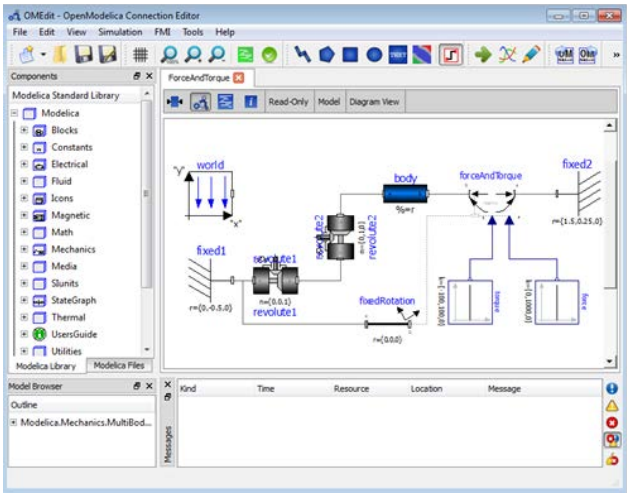


Figure 1. OpenModelica Graphical Editor.

2. OpenModelica Debugger

The debugger is integrated within the Modelica Development Tooling (MDT) which is an Eclipse plugin. It communicates with the Gnu debugger (GDB) via its Machine Interface (MI) channel. Figure 2 shows the Eclipse-based user interface of the debugger.

The debugger provides the following general functionalities:

- Adding/Removing breakpoints.
- Step Over – moves to the next line, skipping the function calls.

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, 19 April, 2013, University of Nottingham, UK.

EOOLT 2013 website:
http://www.eoolt.org/2013/

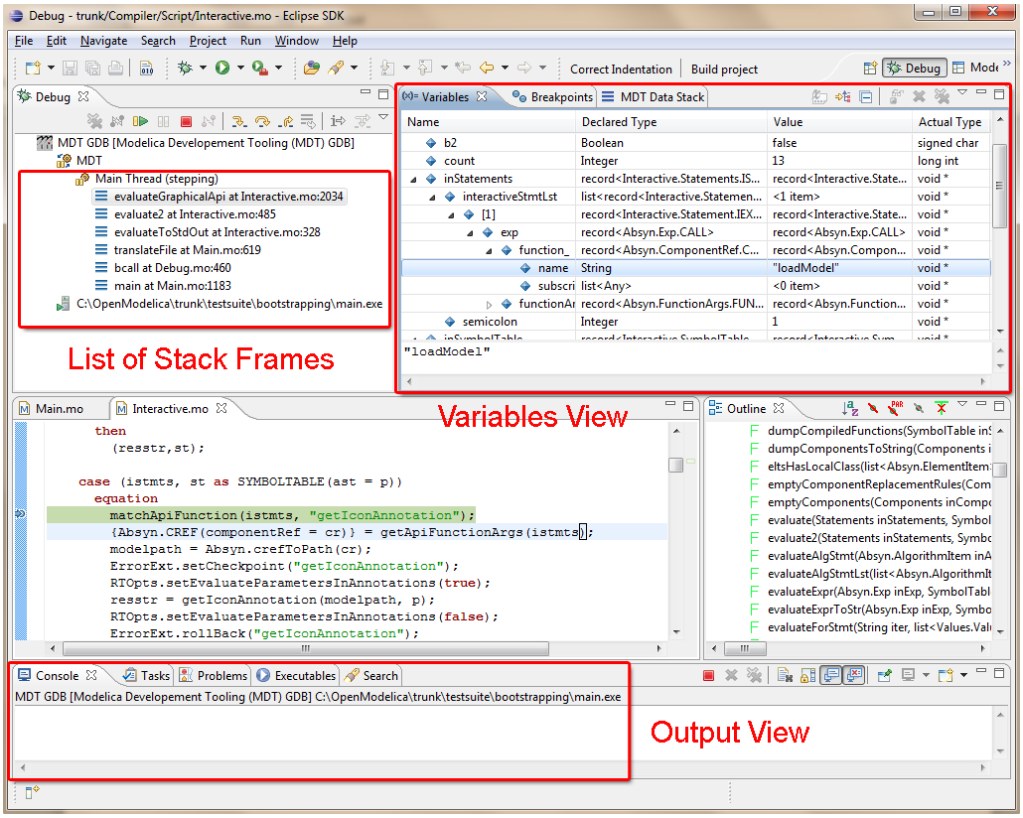


Figure 2. The debug view of the debugger within the MDT Eclipse plugin.

- Step In – takes the user into the function call.
- Step Return – completes the execution of the function and takes the user back to the point from where the function is called.
- Suspend – interrupts the running program.

The debug view primarily consists of two main views:

- Stack Frames View
- Variables View

The stack frames view shows a list of frames that indicates the program flow. The variables view shows the list of variables at the current stack position.

Acknowledgements

This work has been supported by the Open Source Modelica Consortium (OSMC), by Vinnova in the RTSIM, ITEA2 OPENPROD and ITEA2 MODRIO projects, and by SSF in the EDOp project.

References

[1] Peter Fritzson. Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, 940 pp., ISBN 0-471-471631, Wiley-IEEE Press, 2004.

[2] The Modelica Association. The Modelica Language Specification Version 3.2. http://www.modelica.org

[3] Adeel Asghar, Sonia Tariq, Mohsen Torabzadeh-Tari, Peter Fritzson, Adrian Pop, Martin Sjölund, Parham Vasaiely, and Wladimir Schamai. An Open Source Modelica Graphic Editor Integrated with Electronic Notebooks and Interactive Simulation. In Proceedings of the 8th International Modelica Conference, Dresden, Germany, March.20-22, 2011.

[4] Peter Bunus. Debugging Techniques for Equation-Based Languages. PhD Thesis. Department of Computer and Information Science, Linköping University, 2004.

[5] Adrian Pop and Peter Fritzson: A Portable Debugger for Algorithmic Modelica Code. In Proceedings of the 4th International Modelica Conference, Hamburg, Germany, March 7-8, 2005.

[6] Adeel Asghar, Adrian Pop, Martin Sjölund, and Peter Fritzson. Efficient Debugging of Large Algorithmic Modelica Applications. In Proceedings of MATHMOD 2012 7th Vienna International Conference on Mathematical Modelling, Vienna University of Technology, Vienna, Austria, February 15 - 17, 2012.

[7] Adrian Pop, Peter Fritzson, Andreas Remar, Elmir Jagudin, and David Akhvlediani. OpenModelica Development Environment with Eclipse Integration for Browsing, Modeling, and Debugging. In Proceedings of the Modelica'2006, Vienna, Austria, Sept. 4-5, 2006.

[8] Adrian Pop, David Akhvlediani, and Peter Fritzson. Towards Run-time Debugging of Equation-based Object-oriented Languages. In Proceedings of the 48th Scandinavian Conference on Simulation and Modeling (SIMS'2007), available at www.scan-sims.org and http://www.ep.liu.se. Göteborg, Sweden. October 30-31, 2007.

[9] Martin Sjölund and Peter Fritzson. Debugging Symbolic Transformations in Equation Systems. In Proceedings of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, (EOOLT'2011), Zürich, Switzerland, Sept 5, 2011.

[10] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with GDB. Free Software Foundation, 2011.

Modelica on the Java Virtual Machine

Christoph Höger

Technische Universität Berlin, Germany, christoph.hoeger@tu-berlin.de

Abstract

Modelica has seen a steady growth of adaption in industry and research. Yet, most of the currently available tools follow the same technological path: A Modelica model is usually interpreted into a system of equations which is then compiled into e.g. C.

In this work, we demonstrate how a compiler can translate Modelica models into Java classes. Those Java classes can be evaluated into a system of equations which can be solved directly on the JVM.

Implementing this tool yields some interesting problems. Among these are the representation of polymorphic data, runtime-causalisation and equation optimization and Modelica's modification system. All those problems can be solved efficiently on the JVM.

Keywords Modelica, separate compilation, Java

1. Introduction

Modelica [1] is an open, standardized language for the description of hybrid systems of differential and algebraic equations. It brings well-known features from object-oriented languages (like hierarchic composition, inheritance and encapsulation) into the domain of multi-physics modeling. Our focus is the *implementation* of Modelica. In this work, we present a prototypical extension of modim [10] that allows us to compile Modelica to the Java Virtual Machine.

The rest of the paper is organized as follows: First, we will motivate the need for a Modelica Compiler and the JVM as its target platform. Afterwards, we show some details of the translation and the runtime system. Finally we will demonstrate how our prototype performs in model instantiation and simulation.

2. Compiling Modelica

As Modelica's adoption in Industry and Science grows, three aspects of the language gain more and more attention:

First, Modelica allows for the easy exchange of models in forms of libraries. Those libraries are not developed by a tool vendor and tested and shipped with a specific implementation of the language. They rather depend (ideally) only on the language's specification and are platform-independent. When a Modelica model shall be simulated, it is first *instantiated* (or elaborated). During this process, all the equations that make up the final mathematical model are generated.

A Modelica library is thus basically a collection of more or less sophisticated methods to create a mathematical model. In that sense it is comparable to any other library of software. And as for any other computer-language it becomes important for Modelica's library-developers that their models behave *correctly* when being instantiated by a client (i.e. that they do not cause the model instantiation to fail in some situations).

Since Modelica is a statically-typed language, this question can, in theory, be answered by checking that any model conforms to its interface. Unfortunately, there is no formal algorithmic specification of the model instantiation process for Modelica. Instead, every tool provides its own, slightly different interpretation of the specification. Therefore a actual useful typecheck would have to assume a certain execution model. Hence, any safety assumption is only valid for a given interpretation of the Modelica specification.

The second problem follows directly from the usage of libraries. Naturally, libraries tend to offer more features than actually required for a single application. From the modelers perspective this may lead to unexpected large systems of equations. A simple pendulum, for example might be modeled by using the Modelica Multibody library as well as by a few equations from the textbook. Naturally, the Multibody approach is much more generic, since it might easily be adapted to (and used in) more complex mechanical systems. Such systems require the a fast model instantiation since otherwise instantiation time might exceed the actual simulation. In fact it may happen that model engineers are slowed down in their development process just because of the model instantiation overhead.

Finally, Modelica offers a lot of means to actually *compute* models. Currently, those computations range from arrays and loops to redeclarations. Yet it is not hard to predict, that more higher-order modeling features (recursive models, models as parameters etc.) will find their way into the language. All those features share one common pattern: The relation between models and their instances becomes

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. 19 April, 2013, University of Nottingham, UK.

 Proceedings available at: http://www.ep.liu.se/ecp/TBD/

EOOLT 2013 website: http://www.eoolt.org/2013/

$1 : n$ with very large n. In such a case it may become impossible to actually generate code for each instance due to practical reasons: As we have shown in [9], such an attempt yields C-code with a size proportional to the number of model-instances. Therefore for very large n, at least the C-Compiler will usually be unable to generate an executable model.

2.1 Interpreting vs. Compiling Modelica

To all of those problems exists a common solution: Modelica models should be *compiled separately*. When we think of a Modelica model as a method to describe a system of equations, it becomes quite obvious that most current tools (and even the specification) insist on *interpreting* Modelica. There are a few indicators for this:

- Most current tools can be steered into an endless loop during instantiation. A compiler should guarantee termination (for finite input).
- Some implementations will even try to call external functions during instantiation. This may lead to severe problems e.g. in case of cross-compilation.
- The language specification defines array-types to include the actual size. This kind of dependent typing naturally requires an interpreter for a type-check. Since Modelica does not place any syntactical bounds on the expressions used for array-access, this means that a fully compliant type-checker must contain a full Modelica interpreter.

In contrast, a compiler does not need to evaluate any Modelica expressions at all [1]. Instead, a model or a group of models (called the *compilation unit*) is translated into a target language. The *effect* of the model instantiation, i.e. the creation of a system of equations is done by evaluating the compilation result according to the semantics of the target language.

This way, we gain multiple benefits:

- A compiler can be implemented in a way that guarantees termination.
- The compiled fragments can be reused in different contexts.
- The instantiated model is memory-efficient: Only the parts that actually differ between different instances need to be allocated dynamically.

It is important to note that to achieve all those benefits, it is not necessary that the compilation targets are portable (i.e. compatible between different tools). It suffices that there is a known way to compile models for a known tool.

2.2 Separate Compilation & Variable Structure

Besides better language scalability and safety an additional point motivates the compilation of models: In case of a variable-structure system, every mode of the system needs to be processed into a computable form (causalisation, index reduction etc.). A way to achieve this with current implementations is to generate a model for each mode and process it *before* simulation [15].

This method has two drawbacks:

- Every possible mode needs to be known before the simulation starts. This excludes sophisticated models, where some mode depends on the simulation results of another as well as models that contain a large number of possible modes.
- It is impossible to decide which mode will become active during a simulation. Therefore it is quite likely that computational effort is wasted in the translation of unneeded modes.

On the other hand it is quite obvious that a compiler has to postpone all this processing until the instantiation was successful. Naturally, this means that variable-structure processing can be solved by the same machinery as a compiled model: Once a mode-change occurs, instantiate the new model the same way as the old one and continue simulation. Of course, the efficiency of this transition is quite important, but technically, compilation and variable-structure systems share a lot of problems.

2.3 JVM

Our compiler requires a much more sophisticated runtime system than an interpreter: We need to be able to describe and instantiate data-structures and higher-order functions (part of Modelica since version 3.2). Additionally, our runtime needs to support sophisticated graph-algorithms and some numerical methods. For reasons that are explained later, we need an accessible intermediate representation of code. Finally our runtime system needs to be fast enough for real-world applications.

At least two platforms fulfill those requirements: The Java Virtual Machine (JVM, [12]) and the Low Level Virtual Machine (LLVM, [11]). Although both are available for practically every platform we preferred the JVM for the following reasons:

- The JVM is tightly integrated with (but not bound to) Java, which is far more productive than C. This makes it easier to implement the necessary runtime system.
- The JVM is the foundation of a large, active ecosystem. Currently there exist hundreds of thousands of free Java libraries[2]. Currently this is mostly beneficial for the implementation of the runtime system. But integrating Modelica into this ecosystem might yield additional benefits in the future.
- The mainstream JVM implementation, Hotspot is fast. In fact, a modern Java program may beat an equivalent C program in certain areas [3].
- In contrast to LLVM, the JVM already contains a notion of classes and objects which makes it a natural goal for the compilation of Modelica.

[1] It might choose to do so for certain optimizations

[2] available e.g. on http://search.maven.org/

3. Modelica on the JVM

In this section we will explain the principles of the compilation from Modelica to Java. To do so, we will answer a few questions:

- What kind of Java code shall be generated? This is, more or less, a pragmatic design decision. Yet the general layout of the target code influences the runtime system and vice-versa, thus it shall be explained.
- How do we implement Modelica's data structures and the operations defined on them? Especially the translation of Modelica-models and the "`.`"-operator for projection are of interest here.
- How do we translate multiple inheritance?
- How can equations be structured as first class citizens? This is important for efficient simulation: Since we do not want to interpret equations during simulation (rather evaluate their compiled bytecode), it is an interesting question how we can package them into Java-classes.
- How can relations be structured as first class citizens? The availability of relations as first class citizens would enable to define the relation resolution outside of the runtime system and possibly by the user.

3.1 Generated Java Code

As already mentioned, separate compilation means to transform *compilation units* from one language into another. What exactly makes up a compilation unit remains a design decision. The default Java compilation unit is a class. Java-classes are not one-to-one mapped to java-source files. Instead, a `.java` file might contain several nested classes.

Since Java and Modelica share a notion of classes, it seems natural to establish a one-to-one relationship between them. Thus, our compiler will not translate Modelica source files into Java source files[3] but classes into classes.

This decision determines two other aspects of the compilation: Since Java allows aggregation, we can directly translate Modelica's aggregation into it. The same holds for model instantiation. While Modelica has no *explicit* constructors except for records, we are still forced to use them on the Java side.

Other important requirements for our target language are:

- Meaningful names. We will try to keep the names of compiled classes as close as possible to the ones used in the Modelica source. This should make the generated code more readable.
- No compilation into Generics. Java's version of polymorphic types is completely erased at object level. Therefore, there would be no gain in compiling Modelica's polymorphism into Java's.
- Creation of immutable objects. As explained e.g. in [2], immutable objects provide greater safety as well as better performance in many cases. Since Modelica is a declarative language, mutation of objects should be considered an error. Thus every class and all of its fields are declared **`final`**.

3.2 Modification by Reference

Choosing the JVM as our compilation target has one important consequence for model modification: We can directly reuse the JVM's object reference passing for modifications as proposed by Zimmer in [21]. Consider the following Modelica snippet:

```
model A
  model B
    Real x;
  end B;
  replaceable B b;
end A;

A a(b = c);
```

Traditional tools usually would interpret the modification on `a` as an additional set of equality constraints ranging over the fields that `b` and `c` have in common. Notably, most of these tools would also remove equalities in a later optimization step by storing them outside of the system of equations.

On the JVM we can instead directly pass `c` into the instance of `a`. This way, we do not need to create any additional equations or even instantiate `a.b`.

The downside of this approach is, that our runtime system needs to deal with the fact that `c` might have a different data layout than `b`. Even worse, this data layout is in general unknown at the compile time of `A`. We will deal with this issue in section 3.4.

3.3 Inheritance

Modelica allows a class to inherit from multiple distinct classes [4]. Such a class may look like this:

```
model M
  model C
    Real z;
  end C;
  extends A.B;
  extends C;
  A a;
end M;
```

The result of flattening M according to the Modelica specification would yield a representation like this:

```
model M_Flat
  Real a•b•x;
  Real x;
  Real z;
end M_Flat;
```

[3] As said, this is just a design decision. But since Modelica files tend to be rather large in practice, we think it is a well-founded one.

[4] See section 6.2 for a discussion of the "same" source rule in Modelica

Where • is filled in to make a name unique (Some tools use the same character as for projection, while others might prefer underscores etc.). While the generation of composite names is a technical detail, the semantic requirement is not: `M_Flat` contains three real-valued unknowns. This is exactly what any Modelica tool (be it an interpreter or a compiler) has to deliver.

To fulfill this requirement, we introduce special fields into the compiled classes, to represent super-objects (instances of super-classes). That way we can compile the inheritance of arbitrary many classes into a simple form of aggregation:

```
class M {
  public final MBase a;
  public final B superclass1;
  public final C superclass2;
  ...
}
```

Note that the superclasses are declared with the same (compiled) type as in the original model, while the child-object is of type `MBase`. We will motivate this in the next section.

3.4 Uniform Data Representation

Modelica has a structural type system. This means that for any compiled class the compiler cannot know all classes that are compatible (as there infinitely many of such compatible classes). When instantiating the model `A` from our example above, we could legally choose the type of `b` arbitrarily, as long as it contains a real-valued unknown named `x`.

For our implementation this means that there is no way to know where to actually look for e.g. the field `x` in `b`. It may be the first, second or 42nd field in whatever object gets passed in from the outside. Java, as our target language does not allow structural compatible expressions, but requires *nominal* compatibility. That is, every class has to denote the set of its subtypes at its definition site.

Note, that both kinds of systems are equivalent in case all types are known (by simply enumerating all existing types). Thus, the problem is tightly coupled to the proposed scheme of separate compilation.

In addition to structural subtyping, Modelica is a polymorphic language, capable of a restricted form of type-abstraction and type-application. This means we have to deal with the well known problem of compiling universally quantified types: A model may contain a so called `replaceable` type which may be substituted by any (yet unknown) concrete type at the instantiation site.

To both problems, there exists a common solution: Uniform data representation. This kind of compilation strategy aims at compiling every object-level data into the same form. This form then needs to support[5] any operation that is mapped from the source into the target language.

[5] Since Java is statically typed, this means we have to carefully design the runtime system to not cause any Java compile-time-errors.

For Modelica, the most important of these operations is the projection ("`.`"). This operation cannot be directly encoded into the Java-projection (using the same character), because of the difficulties noted above. Instead, every object in our target language has to support this operation via a Java-method `get`.

This method implements the dispatch of a projection among the local fields of a Java class (as already mentioned, aggregation is translated directly into Java). As every object in our runtime system implements this method, a Modelica projection `a.b` can be directly compiled into a Java expression `a.get("b")`

Yet, the important question remains how the compiled object itself can implement `get`. In the presence of polymorphic classes, does a compiled class know its own data layout?

Fortunately, Modelica contains two important restriction to type abstraction:

- It is only applicable in *aggregation*, not inheritance: Thus any class always knows all of it's concrete super-classes (not to confuse with it's super-types as every super-class is also a super-type but not vice-versa) and by conclusion the name of all of its fields. This rule is called the "transitively non replaceable rule" in the Modelica specification.
- Type application is *bounded*. That is, every applicable type has to be compatible with the type, it replaces. Thus, a compiled class might not know the concrete type of its fields, but it does know that they *exist*.

These two restrictions allow a compiler to compute the set of legal right hand sides for a projection: Either it is a local field, or a field inherited by some super-class. In case of a local field, the `get`-method simply returns the child-object. In case of an inherited field, the projection can be forwarded to Java's built-in projection operator, since the concrete type of all super-objects is known at compile-time.

Putting it all together our `get`-method looks like this:

```
public MObj get(String name) {
  switch(name) {
    case "a" : return a;
    case "y" : return superclass1.y;
    case "z" : return superclass1.z;
    case default: throw new
      RuntimeException("");
  }
}
```

The possibility to use a **`switch`**-statement on a `String` is a relatively new feature to Java. As it has to handle only constant input in our case (the field names are always translated into String literals), we can expect a significant performance improvement over techniques like reflection or dynamic method invocation.

3.5 Modification

One important feature of Modelica is the ability to modify certain values of an instantiated object. Although some

fields may be unmodifiable by the keyword `final`, we consider *all* fields modifiable for simplicity.

Such a modification works like a selective record update:

```
model X
  record N
    parameter Real x = 4;
    parameter Real y = 2;
  end N;
  N n(y = 1);
end X;
```

It is important to distinguish this kind of value modification from the type application mentioned above: The type application always implies a value modification, but not vice-versa.

Since we insist on creating immutable objects, value-modification cannot be implemented by assignments. Instead, we have to provide some kind of factory to the instantiated object which takes care of creating the necessary child-objects. The instantiated object is responsible to invoke the factory methods in the correct order.

Additionally the provider of the factory object does not know about the default modifications defined in the instantiated class (due to separate compilation). Therefore, every class provides a default factory which can be extended by any modifying factory at the instantiation site.

```
final class N {
  final MBase x;
  final MBase y;

  public N(NModification factory) {
    x = factory.newX();
    y = factory.newY();
  }

  public class NModification {
    public final MBase newX() {
     return new MParameter(4.0);
    }
    public final MBase newY() {
      return new MParameter(2.0);
    }
  }
}
```

3.6 Equations

The most important parts of a Modelica model are certainly its equations. In this section we will describe the compilation of such an equation on the basis of the Modelica equation below:

```
der(x) = 2*y;
```

As we have already proposed in [9], equations can be compiled into *solution functions* that can be used to compute a root for a given variable. Compiling these functions into Java-classes is then straightforward. In best object-oriented manner, an equation might access the surrounding object via a `self`-reference (comparable to Java's **`this`**). Additionally, an equation must be able to report the set of unknowns it depends on. Now the runtime system can ask an equation what value a certain unknown has as at a certain time (if the runtime system provides this equation with the values of all other unknowns).

```
final MObj self;

...

public final double solveFor(
  final MVar v, final SolvableDAE sys) {

  final MVar v1 = system.der.apply(
    self.get("x"));
  final MVar v2 = (MVar)self.get("y");

  if (v == v1) {
    final Double tmp1 = 2.0;
    final Double tmp2 = sys.valueOf(v2);
    final Double tmp3 = tmp1 * tmp2;
    return tmp3;
  } else if (v == v2) {
    final Double tmp4 = 0.5;
    final Double tmp5 = sys.valueOf(v1);
    final Double tmp6 = tmp4 * tmp5;
    return tmp6;
  }
  throw new RuntimeException();
}
```

This general scheme can of course be easily specialized: Our runtime system contains special classes for linear equations and direct equalities as these can be solved more directly (or even be removed before simulation).

Higher-order equations (e.g. `if`- or `for`- equations) can now be translated into corresponding Java code that instantiates the appropriate equation-objects. It is the task of the runtime system to collect all those objects, sort them and invoke their solution functions in a correct order to compile the global system result. See section 4.2 for a discussion on how this can be achieved.

3.7 Relation Semantics

Modelica does not only allow equations to be generated by model instantiation, but also by *relations*. Relations can take different forms, but the most widely known one is probably the `connect`-statement:

```
model C
  ...
  connect(x, y);
end C;
```

Modelica contains several different kinds of those relations that are all currently specified in the language standard. This increases the size and complexity of the specification significantly and makes it hard to implement Modelica from scratch. Although our favored solution to this

issue (userdefined connection semantics) is not the topic of this paper, our runtime system and compiler are naturally prepared for it. So we give a short overview:

Unlike equations, relations only have a meaning in a global context. Only the complete set of all relation instances (usually called the set of connections, but not limited to the `connect`-relation) can be used to generate a set of equations.

Fortunately, these generated equations always have a certain form. Only some parts of the equations change (e.g. flow sets always generate sum-to-zero equations etc.). Thus, those equations can be seen as some kind of higher order model (parameterized over the relation sets).

To implement this, our runtime contains relations as first-class citizens. As well as equations, they are computed by the model-instances (again allowing loops, conditionals etc.). For this purpose, every compiled model-instance contains a method `relations()` to compute all locally defined relation-objects.

Since relations are often defined on *ports*, which in turn often distribute a relation over their fields, the relation creation is invoked on the first argument of the relation definition. In case of our example model `C` above, the relation method would look like this:

```
public final Iterable<MRelation>
  relations() {

  final MObj tmp1 = self.get("x");
  final MFunction tmp2 =
    tmp1.get("connect");
  final MObj tmp3 = self.get("y");
  return tmp2.apply(tmp1, tmp3);
}
```

The runtime system collects all relations and hands them over to specialized relation-handlers, which are basically models[6] with a certain signature. In a future version, they should be implementable in pure Modelica.

4. The Runtime System

As already mentioned, the runtime system's responsibilities contain the collection of relations and equations, the symbolic manipulation of systems of equations and the control of the simulation.

4.1 Classes

To achieve this, it comes with a set of classes that are to be implemented or generated by the compiled models.

MBase Since we need to distinguish our object system from Java's, `MBase` was introduced. It is the base class of all other classes and serves as the same purpose in our system as `java.lang.Object` on the JVM.

MObj Every model instance may contain equations and relations. This is what distinguishes an instance of this class from the other runtime objects.

[6] This means, it is quite natural, that they create equations.

MEquation As already mentioned, equations have a special interface to enable causalisation.

MFunction As in any functional languages, functions are first-class citizens in our runtime environment. Once Modelica gets (full) support of closures, we will implement this here.

MRelation This class is basically a tag to distinguish a relation from the other runtime objects.

MArray Objects of type `MArray` implement Java's `List` interface. Additionally they provide a specialized implementation of the `get`-method, which is necessary for Modelica's vectorized field access.

MVar Variables basically serve two purposes: They decide what kind of information need to be written into a result file and they work as a placeholder for actual double-values in equations. In a future version, they will probably hold all the additional attributes (e.g. start, min, max) that Modelica supports.

MParameter Objects of type `MParameter` need to be instantiated from user provided values.

MPotentialVar Since Modelica contains a builtin definition for potential variables, our runtime contains a builtin class for them.

MFlowVar As for potentials, flows are builtin to the language and thus mirrored in the runtime system.

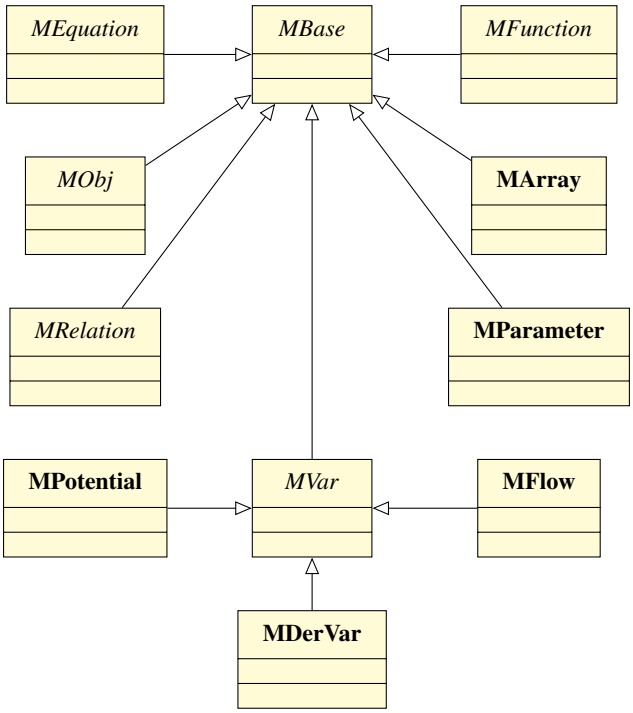


Figure 1. Modelica Runtime Classes

4.2 Causalisation & Index Reduction

Bringing Modelica models into an efficiently computable form usually requires two process-steps: Causalisation and Index-reduction. Informally, causalisation tries to use

Edges	Vertices	Hopcroft-Karp
80	60	5ms
800	600	28ms
8000	6000	135ms
80000	60000	278ms
800000	600000	1.131ms

Table 1. Causalisation runtime on the Automaton model

as much forward-substitution as possible while index-reduction aims at transforming a DAE into an ODE, which can be handled by numerical integration methods.

As we have shown in [9], runtime causalisation can be implemented as a graph algorithm without any symbolic processing if the solution functions are precompiled. We implemented this technique in modim for our compiler. Yet it remained an open question, whether the causalisation would be fast enough to be invoked on every startup. To answer this question we measured the the performance of the Hopcroft-Karp algorithm implemented in the JGraphT library [17] for different sizes of our Automaton model (see Appendix).

The results can be seen in table 4.2. In the given case the algorithm behaved much better than the theoretical limit of $O(|E|\sqrt{|V|})$. But more important is the fact that even for large models (400000 non-equality equations) the causalisation overhead would still be bearable (as $\approx 1s$ of additional instantiation time would be quite small compared to the expected simulation time).

With causalisation settled, index-reduction remains an open problem for our runtime system. Although the actual process is a solved problem with the Dummy Derivatives Method [14] or even more advanced methods ([13]), from our perspective, a more fundamental problem has to be solved first:

The purpose of an index-reduction algorithm is to identify equations that need to be differentiated at most n times to reduce a system of index n to an ODE. The actual value of n can only be computed, once the whole system of equations is known. And as we have stated in the beginning, our system will not do any symbolic processing at this point in time. So the question is not, which algorithm to choose for index-reduction, but how to resolve this conflict.

This is precisely the reason why we demanded an accessible just-in-time compilation format for our runtime: Automatic Differentiation [16] allows for precise computation of derivatives. It has been used with some success for Modelica already [4]. Yet, our approach goes further: With AD, we can compute the derivative of *virtually any* function that can be compiled to the JVM. Since we have access to the Bytecode, we do not need any symbolic information about the equations to do so.

Currently, work is ongoing to implement automatic differentiation for modim. This work aims to leverage the existing AD library nabla [7]. Once the implementation is mature, we can investigate, which index-reduction algorithm is best suited for our approach.

4.3 Numerical Methods

Naturally, numerical simulation requires a set of numerical tools. The numerical algorithms required for a Modelica simulation can roughly be divided in three categories:

- DAE solvers
- ODE solvers
- Algebraic solvers

Unfortunately, there is no implementation of a full-featured DAE solver like DASSL for Java. In theory, we could interface with an existing C/C++ implementation but the data transport overhead would probably cause a too severe performance penalty. In fact there seems to be a general lack of implicit integration methods for the JVM. Therefore future versions of our runtime will probably need to carry their own implementation.

The situation for ODE integrators is slightly better: The apache commons math library [8] contains a comprehensive collection of explicit ODE solvers. There we also find a comprehensive list of root-finding algorithms and some basic linear algebra support. For now modim's numeric runtime depends on those algorithms for simulation.

5. Performance

Benchmarking is a delicate business at best. This is especially true on the JVM: The Java Virtual Machine is a *garbage collecting* environment and issues a just-in-time compiler. For that reasons, the runtime behavior of a program might vary drastically depending on the state of the virtual machine. This is the reason why our performance measurements sometimes seem to run *faster* for larger inputs. It is thus not our goal to give precise information about runtime but to show the general time-frame of a certain problem size.

5.1 Compilation

Compilation performance is negligible in our approach, as long as it does not exceed the instantiation time for even large models. In fact, our current implementation is quite fast: Compiling the Automaton and Cell model below took less than 100ms on a warmed-up JVM.

5.2 Instantiation

There is no standardized or widely adopted Modelica-benchmark (as it exists for other multi-implementation languages like JavaScript). Even if there was one, our compiler would not be able to execute it, due to its prototypical state.

So instead of presenting some semi-accurate numbers comparing our compiled models to another implementation, we will present synthetic benchmarks for instantiation without comparing the results to another tool. The focus here is more on the general *ability* to instantiate such models at all in a timely manner.

[7] http://http://commons.apache.org/sandbox/nabla/

[8] http://http://commons.apache.org/math/

n	# variables	instances	time
1	1	1	15ms
2	2	3	20ms
3	6	10	1ms
4	24	41	2ms
5	120	206	10ms
6	720	1237	60ms
7	5.040	8660	308ms
8	40.320	69281	476ms
9	362.880	623530	873ms

Table 2. Instantiation time of `Faculty`

```
model Faculty "recurses n! times"
  constant Integer n;
  Faculty[n] faculties(each n = n - 1)
    if (n > 1);
  Real root if n == 1;
end Faculty;
```

The model `Faculty` above is instantiated recursively. As one might see at a glance it will create $n!$ unknowns for any parameter $n > 0$. But to do so, it also needs to instantiate a lot of models (actually $n! \sum_{k=1}^{n} \frac{1}{k!}$). Therefore although this model does not have any usage for simulation, we consider it a valuable stress-test for our runtime system. Table 2 shows the instantiation time results for up until $n = 9$.

width	# equations	instance	equations
10	60	28ms	145ms
100	600	3ms	18ms
1000	6000	41ms	202ms
10000	60000	237ms	682ms
100000	600000	41ms	1.175ms

Table 3. Instantiation time for `Automaton`

As a second, more practical test for instantiation performance, we created a small model of a cellular automaton. This kind of models is found in practice in e.g. in thermodynamic models, where space-discretization leads to higher accuracy. Thus we consider the instantiation performance of this model of practical relevance.

```
model Cell
  Real center;
  Real target;
  Real n,e,s,w;

  equation
  target = (n + e + s + w) / 4;
  der(center) = target - center;
end Cell;
```

To improve readability, the whole automaton model can be found at the end of this document. We instantiated the automaton model for different values of `width`. The `height` parameter was set to 1. In contrast to the `Faculty` model above, the Automaton actually creates equations. Thus, the time it takes to calculate them is also of interest in this benchmark. The results are shown in table 3.

5.3 Simulation

When it comes to simulation, a high performance is crucial. While it might seem acceptable to loose some percent against another tool for the benefit of faster compilation and instantiation, already a doubled simulation time would probably rule out our implementation in any relevant applications.

Thus unlike for instantiation, we have to demonstrate that a Java-based solution can be as fast as a C-based one. To show this, we need a reference implementation and a simple model (simple enough for our prototype to compile it as well as simple enough to rule out any relevance of the choice of integration algorithms).

Integration Steps	Java Runtime	OMC Runtime
10000	993ms	1090ms
25000	2464ms	2277ms
50000	5103ms	5304ms
100000	10034ms	11393ms
150000	14810ms	15904ms

Table 4. Simulation times of JVM prototype compared to OMC

For the reference computation, we favored OpenModelica due to its openness and availability. For the model we choose the cellular automaton described above. This model instantiates into an ODE and we do not want to compare sophisticated numerical algorithms but sheer floating point computation. Therefore we choose a simple forward Euler for the integration method.

Table 5.3 contains the results of this comparison for different integration periods. As can be seen our compiled model can indeed be as fast as the omc output. Yet, this does not mean that modim is already faster. Since omc is a much more mature implementation it is highly likely that the small difference between the two is due to additional tasks computed by the omc simulation.

6. Conclusion

Our compiler is still rather prototypical and does not cover even a small part of Modelica. Yet, we can already make some conclusions about the general ability to compile Modelica into Java:

- As we have proposed in [8], Modelica *can* be compiled separately.
- The Java Virtual Machine is a viable platform not only to host an object-oriented modeling language like Modelica, but also to simulate its output.
- The high-level compilation-scheme moves several parts of current tool's "Frontend" into a runtime environment or even a core-library. This decouples language aspects from simulation aspects and hopefully simplifies both further development and maintenance.

6.1 Benefits

The proposed compilation shows several benefits over the classic interpretation:

- Compiled fragments can be *safe*. Once a Modelica model has been typechecked compiled, it is guaranteed[9] to instantiate safely in any other context.
- Instantiation is *fast*. Even without special optimization, structurally changing a model does only involve insignificant cost. This is especially useful during the development cycle of a model, when changes and tests are iterated fast.
- A compiled model is *accessible*. Since Java is practically ubiquitous, a compiled model can be used in many different contexts at ease.
- There seems to be no *general* performance penalty at simulation time.
- Since our compiler naturally provides a foreign function interface (ffi) to Java, the whole JVM-ecosystem can be used easily from within the model.

6.2 Drawbacks

Obviously, our approach also comes with some caveats, that should be mentioned:

- The compiled model always *instantiates* itself before it simulates. If there are no structural changes, this effort is wasted. Of course the model could provide some caching-mechanisms to circumvent this problem, but this would increase both the complexity of implementation and usage.
- The Java-ffi is unlikely to be standardized and adopted by other tools soon. While it is easy to provide from within Java, it is equally hard to do so from within a C-implementation.
- It may conflict with some parts of the language specification. While this case should be rare, we cannot always avoid it. Consider the "same-element" rule in the Modelica specification: It basically states that multiple inheritance of the same element name is fine as long as the definition of that element is the same on all definition sites. Even if we ignore the fact, that “same-ness” is at least quite hard if not impossible to decide in the general case, we could still not fulfill this rule. Simply put, it demands, that a tool *knows* the right-hand side of the definition. This is a violation of the separate compilation principle. So either we could be too negative and simply forbid double definitions completely or be too positive and allow them as long as every right-hand-side evaluates to the same value. Since the last option would raise other questions (what context to evaluate in etc.), we simply forbid double definitions, thus leaving the Modelica Specification.

Although these are valid arguments, we think that the benefits outweigh the drawbacks significantly.

[9] Of course only if the compiler and the type-checker are correct.

6.3 Related Work

Reusing multiple instances of a Modelica model has already been proposed by Zimmer in [19]. This method is applied after instantiation and can only detect some cases of shared instances.

Sol [20] is a language with unbound structural dynamism. It contains an incremental mechanism for index-reduction and causalisation. In contrast to our approach, Sol is completely interpreted and does full symbolic analysis of equations. Yet, the proposed incremental index reduction might be of great value for the further development of modim.

First-class models have been proposed by Broman for the MKL [5] and Giorgidze for Hydra [7]. Hydra contains an implementation of just-in-time compilation and supports structural variable models. Thus, it is closely related to our approach. Yet it does not deal with the Modelica specific aspects of compilation and embeds equations as data-structures into the host language.

Using the JVM as simulation backend has been proposed for JModelica in [18]. Here Modelica is still interpreted. Another JVM simulation backend is implemented in openmodelica [6].

6.4 Future work

As it was mentioned multiple times, our system is far from being complete. Thus the implementation of larger parts of the Modelica specification remains an important, but academically mostly uninteresting task for the future.

Yet, some more interesting tasks remain:

- While a formal description of Modelica as a whole seems unfeasible because of the complexity of the language, our runtime system could be small enough to formally defined. If this was the case, a correctness proof of modim’s typechecker would become feasible.
- Optimization. The JVM gives raise to a large set of interesting features. One could research runtime bytecode generation or solver parallelization. As already mentioned, current work is focusing on the implementation of automatic differentiation. Since this method involves bytecode manipulation, it seems logical to also investigate the application of *bytecode specialisation* on Modelica models.
- Variable structure systems. As mentioned in the beginning, this work now gives the framework for a true variable structure Modelica implementation. Yet it remains a research topic, how efficient e.g. incremental index reduction can be implemented.

Acknowledgments

The author wants to thank the anonymous reviewers for their in-depth reviews and valuable suggestions to improve the quality of this work.

References

[1] Modelica - a unified object-oriented language for physical systems modeling, 2012.

[2] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[3] R.F. Boisvert, J. Moreira, M. Philippsen, and R. Pozo. Java and numerical computing. *Computing in Science Engineering*, 3(2):18 –24, mar/apr 2001.

[4] Willi Braun, Lennart Ochel, and Bernhard Bachmann. Symbolically derived jacobians using automatic differentiation - enhancement of the openmodelica compiler.

[5] David Broman. *Meta-Languages and Semantics for Equation- Based Modeling and Simulation*. PhD thesis, Department of Computer and Information Science, Linköping University, Sweden, 2010.

[6] 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.

[7] G. Giorgidze. *First-class models: On a noncausal language for higher-order and structurally dynamic modelling and simulation*. PhD thesis, The University of Nottingham, 2012.

[8] Christoph Höger, Florian Lorenzen, and Peter Pepper. Notes on the separate compilation of modelica. In Peter Fritzson, Edward Lee, François E. Cellier, and David Broman, editors, *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pages 43–51. Linköping University Electronic Press, 2010.

[9] Christoph Höger. Separate compilation of causalized equations -work in progress. In François E. Cellier, David Broman, Peter Fritzson, and Edward A. Lee, editors, *EOOLT*, volume 56 of *Linköping Electronic Conference Proceedings*, pages 113–120. Linköping University Electronic Press, 2011.

[10] Christoph Höger. Modim - a modelica frontend with static analysis. In *Vienna International Conference on Mathematical Modelling 2012*, Vienna, Austria, February 2012.

[11] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[12] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[13] Sven Erik Mattsson, Hans Olsson, and Hilding Elmqvist. Dynamic selection of states in dymola. In *Modelica Workshop 2000 Proceedings*, pages 61–67, 2000.

[14] Sven Erik Mattsson and Gustaf Söderlind. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM Journal on Scientific Computing*, 14(3):677–692, 1993.

[15] A. Mehlhase. A Python Package for Simulating Variable-Structure Models with Dymola. In Inge Troch, editor, *Proceedings of MATHMOD 2012*, Vienna, Austria, feb 2012. IFAC. submitted.

[16] Uwe Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. SIAM, 2012.

[17] Barak Naveh et al. Jgrapht. *Internet: http://jgrapht. sourceforge. net*, 2008.

[18] Franck Verdiere, Abir Rezgui, Sana Gaaloul, Benoit Delinchant, Laurent Gerbaud, Frédéric Wurtz, and Xavier Brunotte. Modelica models translation into java components for optimization and dae solving using automatic differentiation. In David Al-Dabass, Alessandra Orsoni, and Richard Cant, editors, *UKSim*, pages 340–344. IEEE, 2012.

[19] Dirk Zimmer. Module-preserving compilation of modelica models. In *Proceedings of the 7th International Modelica Conference, Como, Italy, 20-22 September 2009*, Linköping Electronic Conference Proceedings, pages 880–889. Linköping University Electronic Press, Linköpings universitet, 2009.

[20] Dirk Zimmer. *Equation-based Modeling of Variable-structure Systems*. PhD thesis, ETH Zürich, 2010.

[21] Dirk Zimmer. A reference-based parameterization scheme for equation-based object-oriented modeling languages-modim - a modelica frontend with static analysis. In *Vienna International Conference on Mathematical Modelling 2012*, Vienna, Austria, February 2012.

Appendix

```
model Automaton
 parameter Integer width=20, height=20;
 Cell[width, height] cells;

 initial equation
 cells[1,1].center = 1.0;

 equation

 for i in 1:width loop
  for j in 1:height loop
   if i > 1 then
    cells[i,j].w = cells[i-1, j].center;
    cells[i-1, j].e = cells[i, j].center;
   else
    cells[i,j].w = cells[width, j].center;
    cells[width, j].e = cells[i,j].center;
   end if;

   if j > 1 then
    cells[i,j].n = cells[i, j-1].center;
    cells[i, j-1].s = cells[i,j].center;
   else
    cells[i,j].n = cells[i, height].center;
    cells[i, height].s = cells[i,j].center;
   end if;
  end for;
 end for;
end Automaton;
```

An Approach to Cellular Automata Modeling in Modelica

Victorino Sanz Alfonso Urquia

Dpto. de Informática y Automática, ETSI Informática, UNED, Spain

{vsanz,aurquia}@dia.uned.es

Abstract

A new Modelica library, named CellularPDEVS, is introduced in this manuscript. This new library facilitates the description of one- and two-dimensional Cellular Automata (CA) models in Modelica. CellularPDEVS models have been specified using Parallel DEVS. The library has been implemented using the functionality of the DEVSLib library which supports the Parallel DEVS formalism in Modelica. CellularPDEVS allows the user to focus on describing the behavior of the cell and the characteristics of the cellular space. CellularPDEVS models are compatible with other DEVSLib models, facilitating the combination of CA, Parallel DEVS and other Modelica models. Three examples are presented: Wolfram's rule 30 and 110, and the Conway's Game of Life.

Keywords Modelica, Cellular Automata, Parallel DEVS, CellularPDEVS, DEVSLib

1. Introduction

Cellular automata (CA) are a class of models initially proposed in the 1940s by John von Neumann and Stanislaw Ulam [30, 29, 28]. CA are dynamic, discrete-time and discrete-space models. They are represented as a grid of identical discrete volumes, named cells [11]. The grid can be in any finite number of dimensions. The state of each single cell is finite and it is usually represented using integer numbers. The operational dynamics of the automata is described by a rule or transition function that is used to update the state of each cell at discrete time steps. This rule constitutes a function of the current state of the cell and the state of its neighbors, and defines the state of the cell for the next time step [27]. Examples of different neighborhoods are shown in Figure 1: the Moore's neighborhood that includes all the surrounding cells; the von Neumann's neighborhood that includes the cells adjoining the four faces of one cell; or the extended von Neumann's that also includes each cell just beyond one of the four adjoining cells [34].

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. 19 April, 2013, University of Nottingham, UK.

EOOLT 2013 website: http://www.eoolt.org/2013/

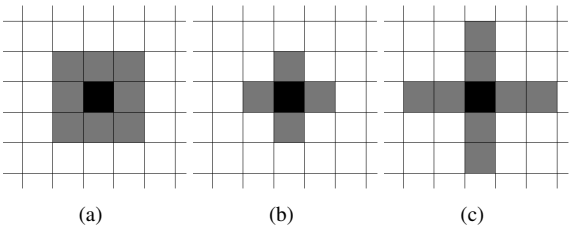


Figure 1. Examples of CA neighborhoods: a) Moore's; b) von Neumann's and; c) extended von Neumann's.

As it can be observed, the definition and behavior of the CA are simple. CA can provide an intuitive way of describing complex behavior using simple rules. CA may be considered as discrete idealizations in time and space of physical systems [35]. Due to its simplicity, CA have been used to describe models of complex systems in multiple domains. CA models have been developed in areas like chemistry [12], economics [22], medicine [10], biology and environment [13], and urban architecture [18], among many others [8]. An extension of CA models, named Lattice Gas Cellular Automata (LGCA), has been applied to the study of fluid flows. LGCA models have been also extended into Lattice Boltzmann Models (LBM) that are used as a microscopic approach for the study of fluid dynamics [33].

CA models can be combined with models described using different formalisms. For example, macroscopic quantities are usually calculated from LGCA and LBM models in order to combine them with other continuous-time models. One of the motivations of the work presented in this manuscript is to facilitate the combination of CA and continuous-time models using Modelica.

Discrete-event modeling specifications have been used to formally describe the behavior of CA, facilitating the development of models and their understanding. For instance, DEVS has been used by Zeigler [37] and Wainer [32] to describe CA models. The former provides a description of CA using Classic DEVS and Multicomponent DEVS. Models following these specifications can be implemented using tools such as DEVSJAVA [38], CoSMoS and MS4 Me™[36]. The latter introduces the Cell-DEVS formalism that is supported by CD++ [31].

On the other hand, the general-purpose, object-oriented modeling languages support the physical modeling paradigm [2]. In particular, the Modelica language [14] facilitates the object-oriented description of DAE-hybrid models, i.e.,

models composed of differential and algebraic equations, and discrete-time events. Modelica supports a declarative description of the continuous-time part of the model (i.e., equation-oriented modeling) and provides language expressions for describing discrete-time events. A detailed description of the characteristics of the language can be found in the specification of the language [14].

Modelica features have facilitated the development of libraries supporting several modeling formalisms and describing phenomena in different physical domains [15]. The main Modelica library is the Modelica Standard Library (MSL) [17] which is developed and supported by the Modelica Association. Modelica facilitates the reuse of models and model components which contribute to reduce the cost of new model development [21].

A number of Modelica libraries have been implemented for supporting discrete-event modeling formalisms, including StateCharts [6], state graphs [19], hybrid automata [20], Petri Nets [16] and extended Petri Nets [5]. The DEVSLib library was developed by the authors to facilitate the description of Parallel DEVS models in Modelica, and their combination with other Modelica models [24, 25].

Modelica has been used to describe CA models. The Game of Life is a particular example of two dimensional CA. The description in Modelica of the Conway's Game of Life is discussed by Fritzson [7]. In this implementation, the cellular space is represented using a matrix of integer numbers. The initial condition is set using a vector that contains the coordinates of the initially active cells. The behavior of the model is implemented using a function that is evaluated at discrete intervals using the Modelica *sample* operator. Two *for* loops are used in this function to iterate over all the components of the matrix, calculating the state of the neighbors (following Moore's neighborhood) and updating the state of the current component.

A new Modelica library, named CellularPDEVS, has been developed by the authors in order to facilitate the description of CA. CA in CellularPDEVS are described using the Parallel DEVS formalism, as coupled models of interconnected atomic cells. CellularPDEVS has been programmed using the DEVSLib Modelica library [25]. In this way, CA constructed using CellularPDEVS are compatible with the models described using DEVSLib, facilitating the connection between CA, Parallel DEVS and other Modelica models. The CellularPDEVS library can be freely downloaded as a part of the DESLib library [26, 4].

The structure of the manuscript is as follows. A short introduction to Parallel DEVS is presented in Section 2. The use of the Parallel DEVS as a base for CA implementation in Modelica is discussed in Section 3. The architecture and functionality of the CellularPDEVS library are described in Section 4. The construction of new CA using CellularPDEVS, as well as examples of one and two dimensional CA, are presented in Section 5. Finally, some future work ideas and conclusions are given in Sections 6 and 7, respectively.

2. Parallel DEVS

The Parallel DEVS formalism is briefly introduced in this section. Models in Parallel DEVS can be described behaviorally (named *atomic*) or structurally (named *coupled*).

2.1 Atomic Parallel DEVS Models

According to the Parallel DEVS formalism, an atomic model is the smallest component that can be used to describe the behavior of a system. It is defined by a tuple of eight elements [3, 37]:

$$Atomic =< X_M, S, Y_M, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta >$$

where:

$X_M = \{(p, v) | p \in IPorts, v \in X_p\}$ is the set of *input ports and values*.

S is the set of *sequential states*.

$Y_M = \{(p, v) | p \in OPorts, v \in Y_p\}$ is the set of *output ports and values*.

$\delta_{int} : S \longrightarrow S$ is the *internal transition* function.

$\delta_{ext} : Q \times X_M^b \longrightarrow S$ is the *external transition* function, where $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ is the *total state* set and e is the *time elapsed* since the last transition.

$\delta_{con} : Q \times X_M^b \longrightarrow S$ is the *confluent transition* function.

$\lambda : S \longrightarrow Y_M^b$ is the *output* function.

$ta : S \longrightarrow \Re_{0,\infty}^+$ is the *time advance* function.

An atomic model remains in the state $s \in S$, for a time interval $t_s = ta(s)$. After t_s is elapsed, an *internal event* is triggered and the state is changed to $s_{new} = \delta_{int}(s)$. Before that, an output can be generated using the output function and the state prior to the event ($output = \lambda(s)$).

A new internal event is scheduled to occur at time instant $t_{new} = ta(s_{new}) + time$, where $time$ is the current time, i.e., the time instant of the current event, and $ta(s_{new})$ is the duration until the next internal event scheduled as a consequence of the current event. The duration $ta(s_{new})$ is a function of the new state s_{new}.

Multiple inputs can be received simultaneously through one or several ports:

- If any input is received at time t_{ext} and $t_{ext} < t_s$ (so the inputs are received before the next internal event), an *external event* is triggered. As a consequence of the external event, the state is changed to $s_{new2} = \delta_{ext}(s, e, bag)$, where s is the current state, e is the elapsed time since the last transition ($t_{ext} - t_{last}$) and $bag \subseteq X_M$ is the set of received input messages.
- If the external input is received at time t_{ext} and $t_{ext} = t_s$, the external and the internal events are triggered simultaneously. This situation triggers a *confluent event* (that substitutes the external and internal events), and the state is changed to $s_{new3} = \delta_{con}(s, e, bag)$, being s the current state, e the elapsed time, and $bag \subseteq X_M$ the set of received inputs (similarly to the δ_{ext} function).

Also, similarly to the internal events, an output can be generated as $output = \lambda(s)$ before executing the confluent transition function.

New internal events are also scheduled after the external and confluent transitions using $ta()$. Note that the time advance function can return a zero value, generating an immediate internal event.

2.2 Coupled Parallel DEVS Models

The Parallel DEVS formalism supports the hierarchical and modular description of the model. Every model has an interface to communicate with other models.

A coupled Parallel DEVS model is a model composed of several interconnected atomic or coupled models that communicate externally using the input and output ports of the coupled model interface. It is described by the following tuple [37]:

$$Coupled =< X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC >$$

where:

$X = \{(p, v) | p \in IPorts, v \in X_p\}$ is the set of *input ports and values*.

$Y = \{(p, v) | p \in OPorts, v \in Y_p\}$ is the set of *output ports and values*.

D is the set of the *component names*.

M_d is a *DEVS model*, for each $d \in D$.

EIC is the *External Input Coupling*: connections between the inputs of the coupled model and its internal components.

EOC is the *External Output Coupling*: connections between the internal components and the outputs of the coupled model.

IC is the *Internal Coupling*: connections between the internal components.

The connection of Parallel DEVS models implies the establishment of an information transmission mechanism between the connected models. Parallel DEVS models follow a message passing communication mechanism. A model generates messages as outputs using its output function which are received by other models as external inputs. Messages can be received simultaneously through one or multiple ports. Connections between models can be in the form of 1-to-1, 1-to-many and many-to-1. Each message can transport an arbitrarily complex amount of information, depending on the particular application or experiment being studied.

3. Specification of CellularPDEVS Models

Parallel DEVS has been used to describe the CA components (i.e., the cell and the cellular space) implemented in CellularPDEVS. The formal specification of these components is presented in this section.

3.1 Specification of the Cell Models

CellularPDEVS includes two cell models: $Cell1D$ and $Cell2D$. The formal specification of the one dimensional cell, $Cell1D$, following Parallel DEVS is shown in Table 1.

The interface of $Cell1D$ is composed of:

- The "in_E" input port, used to connect with its eastern neighbor.
- The "in_W" input port, used to connect with its western neighbor.
- The "in_{ext}" input port, used to receive external inputs from outside the cellular space.
- The "out" output port, used to communicate the state of the cell.

A graphical representation of the interface of the $Cell1D$ model is shown in Figure 2.

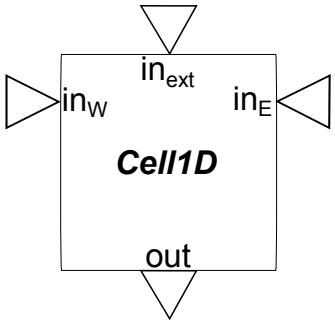


Figure 2. Interface of the $Cell1D$ model.

The state variables of $Cell1D$ are:

- $phase$ that represents if the cell is "$active$" or "$passive$". An "$active$" phase means that the state variable CS of the cell has changed in the current time step. The cell remains "$active$" until the change of the state is communicated to the neighbors. Otherwise, the phase is "$passive$".
- $sigma$ that represents the time delay until the execution of the next internal transition of the cell.
- CS that represents the current state of the cell represented by the $Cell1D$ model.
- N_E and N_W that are used to locally store the state of the neighbors, also represented by $Cell1D$ models.

The behavior of the $Cell1D$ model is as follows. An example of cell simulation is shown in Figure 3, where the evolution of the inputs, outputs and the state of the cell are shown. Initially, cells have $sigma = \infty$ and $phase =$ "$passive$" meaning that without an external input no internal transitions will be executed in the cell. Input events sent to the "in_{ext}" port are intended for initializing the cell, and have to be received at discrete time steps. The default duration of the time step is 1 second, however it can be adjusted as desired. Input events received at port "in_{ext}" update the state variable CS of the cell, set $phase =$ "$active$" and schedule an internal event at $time + 0.5$ (i.e., the middle of the current time step). In the example shown in Figure 3, the initial input event is received

Table 1. Parallel DEVS specification of the Cell1D model included in CellularPDEVS.

$$Cell1D =< X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta >$$

where:

$X_M = \{(ps, v) | p \in \{\text{“}in_E\text{”}, \text{“}in_W\text{”}, \text{“}in_{ext}\text{”}\}, v \in \mathbb{Z}\}$

$S = \{\text{“}active\text{”}, \text{“}passive\text{”}\} \times \mathbb{R}^{+}_{0,\infty} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$

$Y_M = \{(\text{‘}out\text{”}, \mathbb{Z})\}$

$\delta_{int}(phase, sigma, CS, N_E, N_W) =$

$$\begin{cases} (\text{“}passive\text{”}, 0.5, CS, N_E, N_W) & \text{if } phase == \text{“}active\text{”} \\ (\text{“}passive\text{”}, \infty, CS, N_E, N_W) & \text{if } phase == \text{“}passive\text{”} \text{ and } CS == Rule(CS, N_E, N_W) \\ (\text{“}active\text{”}, 0.5, Rule(CS, N_E, N_W), N_E, N_W) & \text{if } phase == \text{“}passive\text{”} \text{ and } CS \neq Rule(CS, N_E, N_W) \end{cases}$$

$\delta_{ext}(phase, sigma, CS, N_E, N_W, e, X^b_M) =$

$$\begin{cases} (phase, 0.5, CS, V_E, N_W) & \text{if event received in port “}in_E\text{” , whose value } V_E \in \mathbb{Z} \\ (phase, 0.5, CS, N_E, V_W) & \text{if event received in port “}in_W\text{” , whose value } V_W \in \mathbb{Z} \\ (phase, 0.5, CS, V_E, V_W) & \text{if events received in ports “}in_E\text{” and “}in_W\text{” , whose values } V_E, V_W \in \mathbb{Z} \\ (\text{“}active\text{”}, 0.5, V_{ext}, N_E, N_W) & \text{if event received in port “}in_{ext}\text{” , whose value } V_{ext} \in \mathbb{Z} \end{cases}$$

$\delta_{con}(S, e, X^b) = \delta_{int}(\delta_{ext}(S, e, X^b))$

$\lambda(phase, sigma, CS, N_E, N_W) =$

$$\begin{cases} (\text{“}out\text{”}, CS) & \text{if } phase == \text{“}active\text{”} \\ \varnothing & \text{if } phase == \text{“}passive\text{”} \end{cases}$$

$ta(phase, sigma, CS, N_E, N_W) = max(sigma, 0)$

at $time = 2s$. The scheduled internal event generates an output to send the new state to the neighbors (e.g., output at $time = 2.5s$ in the figure), fires an internal transition that sets $phase =$ “$passive$” and schedules a new internal event at $time + 0.5$ that corresponds to the next time step. The new scheduled internal event will fire a new internal transition to update the state CS of the cell using the $Rule$ function. If $CS \neq Rule(CS, N_E, N_W)$ (i.e., CS changes) then $phase =$ “$active$”, $sigma = 0.5$ (i.e., the middle of the time step) and $CS = Rule(CS, N_E, N_W)$ (e.g., in the example, $phase$ always changes to “$active$” when CS changes). Otherwise, $phase =$ “$passive$” and $sigma = \infty$ (e.g., in the example CS remains constant at $time = 3s$ and $sigma = \infty$). Each neighbor receives the update of the state in the middle of the time step as an external event. This fires an external transition that updates the locally stored neighbor state with the received value and sets $sigma = 0.5$ to schedule an internal event at the next time step (e.g., in the example, changes in the states of the neighbors are received as external inputs, and the local state variables N_E and N_W are updated with the received values).

The formal specification of the $Cell2D$ is analogous to the $Cell1D$, provided the following modifications. Additional input ports have to be included to connect with the additional neighbors (e.g., eight neighbors in the case of the Moore’s neighborhood). The state has also to be extended to locally store the states of the additional neighbors. The δ_{ext} is used to update the locally stored states with the values received with the events. This function has

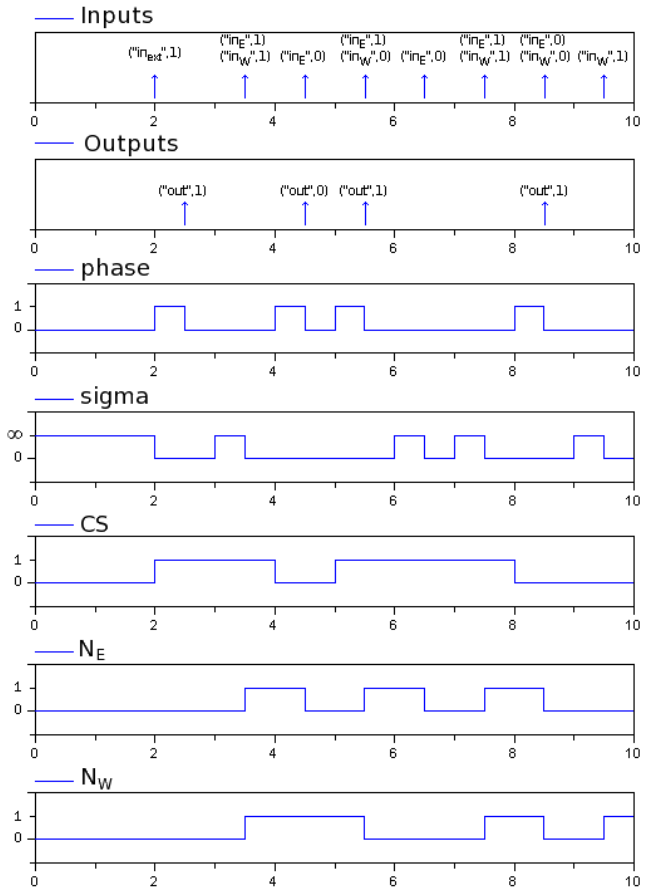


Figure 3. $Cell1D$ model execution example.

Table 2. Parallel DEVS specification of the $CellSpace1D$ model.

$$CellSpace1D =< X, Y, \{M_d | d \in D\}, EIC, EOC, IC >$$

where:
$X = \{(p, v) | p \in \{\text{“}in_1\text{”}, \ldots, \text{“}in_N\text{”}\}, v \in \mathbb{Z}\}$
$Y = \{(p, v) | p \in \{\text{“}out_1\text{”}, \ldots, \text{“}out_N\text{”}\}, v \in \mathbb{Z}\}$
$M_d = Cell1D$ for all $d \in D$, where $D = \{cell_1, \ldots, cell_N\}$
$EIC = \{(CellSpace1D, \text{“}in_i\text{”}) - (cell_i, \text{“}in_{ext}\text{”}) | i = 1, \ldots, N\}$
$EOC = \{(CellSpace1D, \text{“}out_i\text{”}) - (cell_i, \text{“}out\text{”}) | i = 1, \ldots, N\}$
$IC = \{(cell_{i-1}, \text{“}out\text{”}) - (cell_i, \text{“}in_W\text{”}) | i = 2, \ldots, N\} \cup$
$\{(cell_{i+1}, \text{“}out\text{”}) - (cell_i, \text{“}in_E\text{”}) | i = 1, \ldots, N-1\} \cup$
$\{(cell_N, \text{“}out\text{”}) - (cell_1, \text{“}in_W\text{”}), (cell_1, \text{“}out\text{”}) - (cell_N, \text{“}in_E\text{”})\}$

to be extended to allow all the possible combinations of input events from the ports of the model.

3.2 Specification of the Cellular Space Model

CellularPDEVS includes two models that represent cellular spaces: $CellSpace1D$ and $CellSpace2D$. Each cellular space is defined as a coupled Parallel DEVS model. Cellular spaces are composed of individual cells and their interconnections. The formal specification of the one dimensional cellular space, $CellSpace1D$, using Parallel DEVS is shown in Table 2.

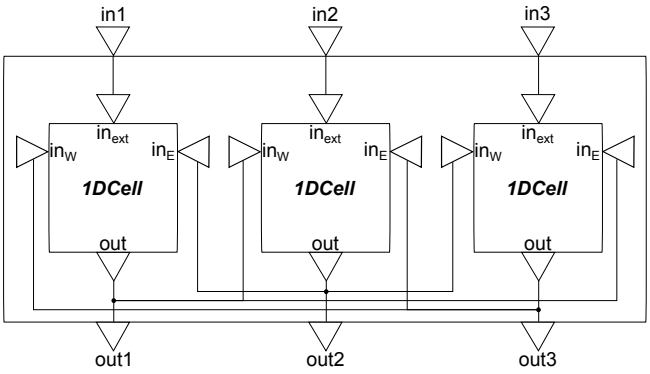


Figure 4. $CellSpace1D$ model of size 3.

The $CellSpace1D$ model is defined as an array of $Cell1D$ atomic models. The size of the array is N. An example of one dimensional CA with three cells is shown in Figure 4. Each cell in the array receives connections from its eastern neighbor (to the “in_E” port) and its western neighbor (to the “in_W” port) following the one dimensional Moore's neighborhood. The boundaries of the space are considered wrapped, so the western neighbor of the first cell of the array is the last cell of the array and vice-versa for the eastern neighbor of the last cell (cf. connections shown in Figure 4). The interface of the cellular space is composed of one input port (“in_i”) and one output port (“out_i”) for each cell in the space. These ports are connected to the “in_{ext}” and “out” ports of each cell, respectively.

The specification of the $CellSpace2D$ model is analogous to the $CellSpace1D$ provided the following modifications. The cellular space has to be defined as a two dimensional matrix of $Cell1D$ atomic models. The size of the space is $N \times N$. Each cell receives connections from its eight neighbors to its input ports. The boundaries are also wrapped considering the two dimensions of the space.

4. Architecture of CellularPDEVS

The architecture of the CellularPDEVS library is shown in Figure 5.

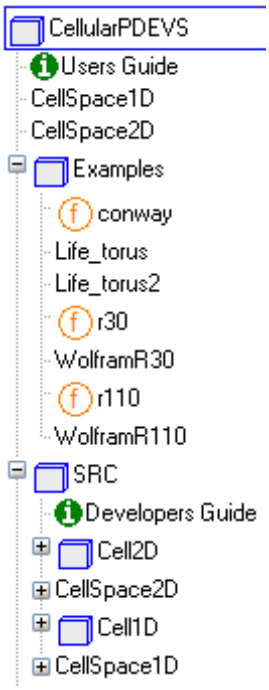


Figure 5. Architecture of the CellularPDEVS library.

The library is structured in two areas: 1) user's area and; 2) developer's area. The user's area is composed of:

- The *Users Guide* that contains the user oriented documentation.
- The *CellSpace1D* model that is used to construct new one dimensional CA.
- The *CellSpace2D* model that is used to construct new two dimensional CA.

- The *Examples* package that contains several examples of use.

The developer's area is encapsulated into the *SRC* package and contains the internal implementation of the models and the developer oriented documentation. CellularPDEVS includes two atomic DEVSLib models to represent one and two dimensional cells, named *Cell1D* and *Cell2D* respectively. Cellular spaces, named *SRC.CellSpace1D* and *SRC.CellSpace2D*, are constructed as an array or a matrix of interconnected *Cell1D* or *Cell2D* models depending on the dimension of the space.

The connections between cells are predefined into the cellular space describing the Moore's neighborhood. Since these connections between individual cells generate algebraic loops in the cellular space, a BreakLoop model from the DEVSLib library is inserted between cell connections. The BreakLoop model uses the Modelica *pre* operator to break the loop. The boundaries of the cellular space are wrapped automatically, by using the *mod* operator in the calculations of the indecees for the connections of cells.

Also, the cellular space includes the models required to generate the graphical animation of the simulation. The World, Fixed and fixedShape models, from the Multibody package of the Modelica Standard Library, have been used to generate the 1D and 2D visualizations.

Each cellular space model in CellularPDEVS includes a replaceable function, named *Rule* that needs to be redeclared in order to define the transition function for new models. The inputs of this function are the state of the current cell and its neighbors, and the output is the future state of the current cell. The prototype of this function is shown in Listing 1.

```
function Rule
  input Integer s;
  input Integer[N] neighbors;
  output Integer sout;
algorithm
end Rule;
```

Listing 1. Prototype of the CellularPDEVS transition function (`N` is the number of neighbors).

Finally, the cellular space also includes a Generator and a DUP_N models, from the DEVSLib library that are used to initialize the required cells at the beginning of the simulation. A message with $Type == 1$ is sent by the Generator model to each cell to be initialized. This message is received and managed by the external transition function of the cell which initializes the state and schedules an internal transition.

5. Modeling using CellularPDEVS

The construction of new CA using CellularPDEVS requires the description of the parameters of the cellular space (i.e., size and initial conditions of the cells) and the rule or transition function that describes the behavior of each cell. The formalism and the internal implementation of the cellular space and the cells is transparent to the user. As mentioned before, the graphical animation of the simulation is automatically generated, and it can be deactivated using a parameter of the model.

The transition function can be any Modelica function with the state of the cell and its neighbors as inputs, and the updated cell state as output (cf. the prototype of this function shown in Listing 1). All these values are represented using integer numbers.

CellularPDEVS models can be combined with any other Modelica model. The state of the cells in the automata can be observed using a variable, named *state*, included in the *CellSpace1D* and *CellSpace2D* models. Also, the state of the cells can be modified during the simulation by sending a message to the in_1 port of the desired cell. The message has to have $Type == 1$ and transport the new value for the cell. This message can be sent from any model constructed using DEVSLib. The *DUP_N* model can be used to duplicate the message if multiple cells have to be changed simultaneously.

Additionally, DEVSLib includes interfaces between continuous-time models and Parallel DEVS models which translate continuous-time signals into event trajectories (i.e., series of messages), and viceversa. These interface models allow combining the use of Parallel DEVS models developed with DEVSLib and hybrid models developed using other Modelica libraries. In this way, the behavior of a continuous Modelica model can be used to affect the state of the CA model, and viceversa.

The continuous-time to discrete-event interfaces translate continuous-time signals into event trajectories, where each event corresponds with the send of a message. Two different implementations of this interface are included in DEVSLib: quantization (*Quantizer* model) and value-crossing interfaces (*CrossUP* and *CrossDOWN* models). The quantization interface generates an event (i.e., a message) for every change in the continuous-time signal bigger than a given quantum value. The value-crossing interface generates an event every time the continuous signal crosses a given value in one direction, upwards or downwards.

The discrete-event to continuous-time interface translates the received message values into a piecewise-constant real signal. A boolean output is also included, together with the output real signal, in order to notify the reception instant of the messages. This boolean output may be useful when the received messages have the same value and consequently the reception instants cannot be inferred from the output real signal. This interface is implemented by the *DICO* model.

CellularPDEVS includes a package that contains several example model that are used to demonstrate its functionality and facilitate the construction of new models. These models have been also used to validate the library by comparison with equivalent models constructed using Golly which is an open source application for exploring CA models [9].

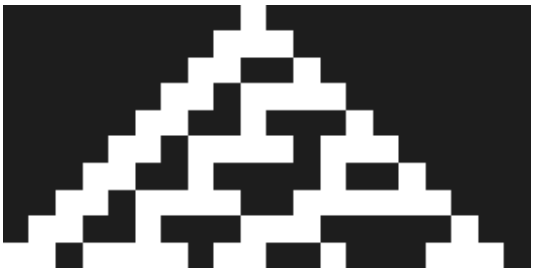
The examples included are the Wolfram's rule 30 and rule 110 [35] and two different initial conditions for the Conway's Game of Life. These models are detailed next.

5.1 Examples of One Dimensional CA

The Wolfram's rule 30 and rule 110 represent two different transition functions for one dimensional CA. These functions evaluate the state of a cell and its two adjacent neighbors and return the future state for the cell. The state of each cell is binary. The combination of possible input values and their outputs are shown in Table 3. The number of the rule, 30 and 110, defines the decimal value of the binary outputs of each function (e.g., looking at the *future state* row of the table, the binary values of the output can be interpreted as a decimal number: for the rule 30, $00011110_2 = 30_{10}$).

Table 3. Wolfram's rule 30 and rule 110.

Rule 30								
current pattern	111	110	101	100	011	010	001	000
future state	0	0	0	1	1	1	1	0
Rule 110								
current pattern	111	110	101	100	011	010	001	000
future state	0	1	1	0	1	1	1	0



(a)

(b)

Figure 6. Simulation of CellularPDEVS 1D models: a) rule 30 and; b) rule 110.

The implementation of these rules in Modelica is straight forward. The code included in CellularPDEVS for the rule 30 is shown in Listing 2. An analogous code is included in the library for the rule 110.

The models of the rule 30 and rule 110 are constructed in CellularPDEVS by extending the *CellSpace1D* model, and redeclaring the *Rule* function with the corresponding functions. The parameters of the models are the size of the cellular space size (named *Ssize*) and the initial cell (named *init_cell*), since all Wolfram rules have only one active cell at the beginning of the simulation. The simulation results of the rule 30 and rule 110 models with a space size of 20, the cell in the middle of the space as initial cell (i.e., *init_cell* = 10) and a simulation time of 10 time steps are shown in Figure 6.

```
function r30
  input Integer s;
  input Integer[2] neighbors;
  output Integer sout;
protected
  Integer[2] n = neighbors;
algorithm
  if      n[2]==1 and s==1 and n[1]==1 then
    sout := 0;
  elseif n[2]==1 and s==1 and n[1]==0 then
    sout := 0;
  elseif n[2]==1 and s==0 and n[1]==1 then
    sout := 0;
  elseif n[2]==1 and s==0 and n[1]==0 then
    sout := 1;
  elseif n[2]==0 and s==1 and n[1]==1 then
    sout := 1;
  elseif n[2]==0 and s==1 and n[1]==0 then
    sout := 1;
  elseif n[2]==0 and s==0 and n[1]==1 then
    sout := 1;
  elseif n[2]==0 and s==0 and n[1]==0 then
    sout := 0;
  end if;
end r30;
```

Listing 2. Rule 30 Modelica code.

5.2 Examples of Two Dimensional CA

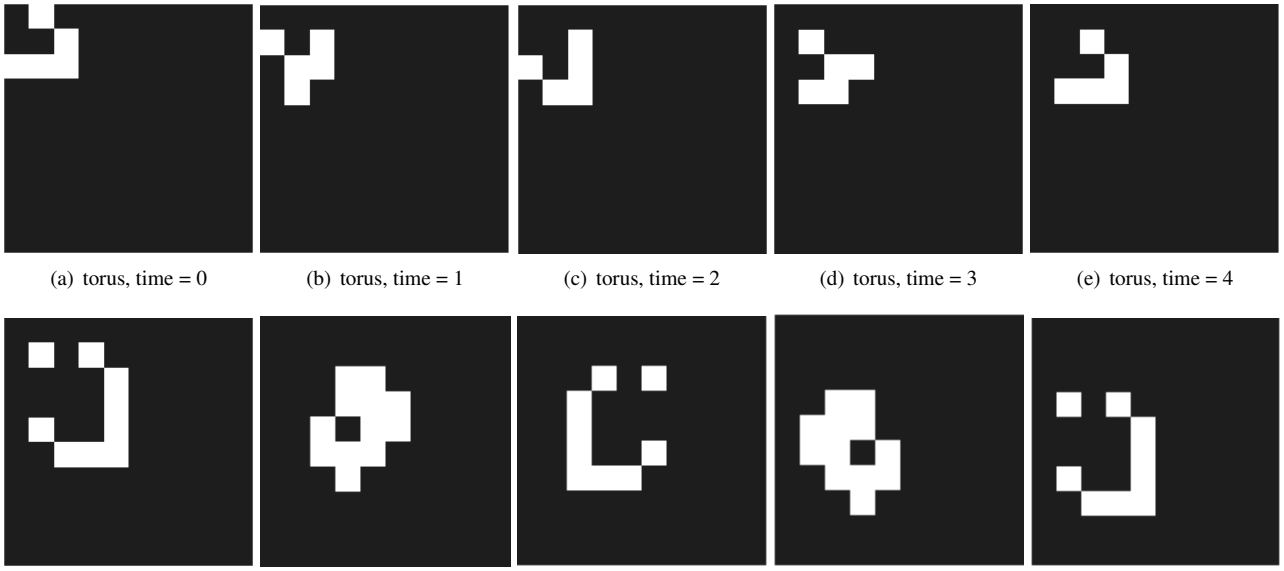
CellularPDEVS includes an implementation of the Game of Life model described by Conway. This model represents a two dimensional cell space where each cell may be alive or dead. The transition function of the model is defined by the following rules:

- A dead cell becomes alive when it has a number of alive neighbors equal to 3.
- A living cell dies when it has less than 2 or more than 3 alive neighbors.
- Otherwise, the cell remains in its current state.

```
function conway
  input Integer s;
  input Integer[8] neighbors;
  output Integer sout;
protected
  Integer[8] n = neighbors;
algorithm
  sout := s;
  if s==0 then // dead, maybe borns
    if sum(n)==3 then
      sout := 1;
    end if;
  else // alive, maybe dies
    if (sum(n)<2 or sum(n)>3) then
      sout := 0;
    end if;
  end if;
end conway;
```

Listing 3. Modelica code of the Game of Life's transition function.

The description of 2D models in CellularPDEVS is analogous to the 1D ones. The Game of Life model is con-



(a) torus, time = 0 (b) torus, time = 1 (c) torus, time = 2 (d) torus, time = 3 (e) torus, time = 4

(f) torus2, time = 0 (g) torus2, time = 1 (h) torus2, time = 2 (i) torus2, time = 3 (j) torus2, time = 4

Figure 7. Simulation of CellularPDEVS 2D models. The Conway's Game of Life.

structed by programming its transition function, as shown in Listing 3. The model extends the *CellSpace2D* model and redeclares the *Rule* function using the *conway* function shown. The initial state of the model is described as a matrix where each row represents the coordinates that correspond to the cells in the space that will be initially active (e.g., [1,2;2,3;4,4]). The first cell, (1,1), corresponds to the top-left cell of the matrix. The first index represents rows and the second represents columns (e.g., (3,5) represents the cell in the third row and the fifth column).

The first five steps of the simulation of two initial states, named *torus* and *torus2*, for the game of life model in CellularPDEVS are shown in Figure 7. The *torus* model corresponds to the initial cells: [1,2; 2,3; 3,1; 3,2; 3,3]. This model evolves in a periodical diagonal movement from the top-left area of the cellular space to the bottom-right. The *torus2* model corresponds to the initial cells: [2,2; 2,4; 3,5; 4,5; 5,5; 6,3; 6,4; 6,5; 5,2]. This model evolves in a periodical vertical movement from the top area of the cellular space to the bottom area.

6. Future Work

The models presented in this manuscript have been included in CellularPDEVS in order to validate the library and demonstrate its functionality. CellularPDEVS will be used to model more complex systems using CA such as a cement clinker cooler [1] or a PEM fuel cell [23]. The development of these models will show the applicability of library. Also, these new CA models will be used to evaluate the simulation performance of the library in comparison with the already developed Modelica models.

7. Conclusions

A new Modelica library has been developed to facilitate the description of Cellular Automata. These are discrete-time and -space models represented using a grid of individual cells, whose state is updated at discrete time steps using a predefined transition function. The library supports the description of one and two dimensional automata. The main components of the library are the cell and the cellular space. The behavior of these components has been specified using the Parallel DEVS formalism.

The behavior of each cell is specified as an atomic Parallel DEVS model and implemented using the DEVSLib library. The interface of the cell allows it to receive messages from its neighbors and from outside the cellular space. The state of the cell represented by the model is described using an integer number. Different behaviors can be described by redeclaring the rule, or transition function that defines the dynamics of the cell. The duration of the time step can be adjusted to the requirements of the simulation.

The cellular space is specified as a coupled Parallel DEVS model. It is composed of a grid of interconnected cell models. The interface of the cellular space allows to receive external messages, via its input ports, and to observe the state of the automata, via its output ports. This interface facilitates the combination of cellular automata models with other Modelica models. The boundaries of the cellular space are wrapped. It uses the Moore's neighborhood by default. However, future versions will allow the user to define the neighborhood as desired. A graphical animation of the simulation is automatically generated.

Three examples have been presented in order to demonstrate the functionality of the library and validate its models. The Wolfram's rule 30 and rule 110 elementary cellular automata as examples of one dimensional automata. The Conway's Game of Life, including two initial conditions, as an example of two dimensional automata.

References

[1] Oscar Acuña, Carla Martin-Villalba, and Alfonso Urquia. Virtual-lab of a cement clinker cooler for operator training.

In *Proceedings of the 7^{th} Vienna International Conference on Mathematical Modeling (MATHMOD)*, pages 331–336, Vienna, Austria, 2012.

[2] Karl J. Åström, Hilding Elmqvist, and Sven Erik Mattsson. Evolution of continuous-time modeling and simulation. In *Proceedings of the 12^{th} European Simulation Multiconference (ESM'98)*, pages 9–18, Manchester, UK, 1998.

[3] Alex Chung Hen Chow. Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator. *Transactions of the Society for Computer Simulation International*, 13(2):55–67, 1996.

[4] Some free modeling and simulation resources. Dpto. Informática y Automática, UNED. `http://www.euclides.dia.uned.es/`, 2013.

[5] Stefan M.O. Fabricius and E. Badreddin. Hybrid dynamic plant performance analysis supported by extensions to the Petri Net library in Modelica. In *Proceedings of the 4^{th} Asian control Conference (ASCC)*, pages 41–50, Singapore, 2002.

[6] 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.

[7] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Computer Society Pr, 2003.

[8] Niloy Ganguly, Biplab K Sikdar, Andreas Deutsch, Geoffrey Canright, and P Pal Chaudhuri. A survey on cellular automata. Technical report, 2003.

[9] Golly. `http://golly.sourceforge.net`, 2013.

[10] H. Hötzendorfer, W. Estelberger, F. Breitenecker, and S. Wassertheurer. Three-dimensional cellular automaton simulation of tumour growth in inhomogeneus oxygen environment. *Mathematical and Computer Modelling of Dynamical Systems*, 15:177–189, 2009.

[11] Andrew Ilachinski. *Cellular Automata: A Discrete Universe*. World Scientific, Singapore, 2001.

[12] Lemont B. Kier, Paul G. Seybold, and Chao-Kun Cheng. *Modeling Chemical Systems using Cellular Automata*. Springer, Dordrecht, The Netherlands, 2005.

[13] Jiri Kroc, Peter M.A. Sloot, and Alfons G. Hoekstra, editors. *Simulating Complex Systems by Cellular Automata*. Springer-Verlag, Berlin, 2010.

[14] Modelica Association. Modelica - An unified object-oriented language for physical systems modeling. Language specification version 3.1, 2012.

[15] Modelica Libraries. Modelica free and comercial libraries. `http://www.modelica.org/libraries`, 2013.

[16] 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.

[17] Modelica standard library. `http://www.modelica.org/libraries/Modelica`, February 2010.

[18] David O'Sullivan and Paul M. Torrens. Cellular models of urban systems. In S. Bandini and T. Worsch, editors, *Theoretical and Practical Issues on Cellular Automata*. Springer-Verlag, London, 2000.

[19] 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, Hamburg, Germany, 2005.

[20] Tiziano Pulecchi and Francesco Casella. HyAuLib: modelling hybrid automata in Modelica. In *Proceedings of the 6^{th} International Modelica Conference*, pages 239–246, Bielefeld, Germany, 2008.

[21] Stewart Robinson, Richard E. Nance, Ray J. Paul, Michael Pidd, and Simon J.E. Taylor. Simulation model reuse: definitions, benefits and obstacles. *Simulation Modelling Practice and Theory*, 12(7-8):479 – 494, 2004. Simulation in Operational Research.

[22] Jean-François Rouhaud. Cellular automata and consumer behaviour. *European Journal of Economic and Social Systems*, 14:37–52, 2000.

[23] Miguel A. Rubio, Alfonso Urquia, and Sebastian Dormido. Dynamic modelling of PEM fuel cells using the *fuelcell*lib Modelica library. *Mathematical and Computer Modelling of Dynamical Systems*, 16(3):165–194, 2010.

[24] Victorino Sanz. *Hybrid System Modeling Using the Parallel DEVS Formalism and the Modelica Language*. PhD thesis, E.T.S.I. Informática, UNED, Madrid, Spain, 2010.

[25] Victorino Sanz, Alfonso Urquia, François E. Cellier, and Sebastian Dormido. System modeling using the Parallel DEVS formalism and the Modelica language. *Simulation Modeling Practice and Theory*, 18(7):998–1018, 2010.

[26] Victorino Sanz, Alfonso Urquia, and Sebastian Dormido. Parallel DEVS and process-oriented modeling in Modelica. In *Proceedings of the 7^{th} International Modelica Conference*, pages 96–107, Como, Italy, 2009.

[27] Joel L. Schiff. *Cellular Automata: A Discrete View of the World*. Wiley-Interscience, New York, NY, USA, 2008.

[28] Stanislaw Ulam. Random processes and transformations. In *Sets, Numbers and Universes*. MIT Press, Cambridge, 1974.

[29] John von Neumann. The general and logical theory of automata. In *Collectd Works*, volume 5. Macmillan, New York, 1963.

[30] John von Neumann. *Theory of self-reproducing automata*. University of Illinois Press, Urbana and London, 1966.

[31] Gabriel Wainer. CD++: A toolkit to develop DEVS models. *Software: Practice and Experience*, 32(13):1261–1306, 2002.

[32] Gabriel A. Wainer. *Discrete-Event Modeling and Simulation - A Practitioner's Approach*. CRC Press, Boca Raton, FL, USA, 2009.

[33] Dieter A. Wolf-Gladrow, editor. *Lattice Gas Cellular Automata and Lattice Boltzmann Models: An Introducction*. Springer-Verlag, Berlin, 2000.

[34] Stephen Wolfram. *Cellular Automata and Complexity: Collected Papers*. Addison-Wesley, 1994.

[35] Stephen Wolfram. *A New Kind of Science*. Wolfram Media Inc., Champain, IL, USA, 2002.

[36] Benard P. Zeigler and Hessam S. Sarjoughian, editors.

Guide to Modeling and Simulation of Systems of Systems. Springer-Verlag, London, 2013.

[37] Bernard P. Zeigler, Tag Gon Kim, and Herbert Prähofer. *Theory of Modeling and Simulation*. Academic Press, Inc., Orlando, FL, USA, 2000.

[38] Bernard P. Zeigler and Hessam S. Sarjoughian. Introduction to DEVS modeling & simulation with JAVA: Developing component-based simulation models, 2003.

Models for Distributed Real-Time Simulation in a Vehicle Co-Simulator Setup

Anders Andersson[1] Peter Fritzson[2]

[1]Swedish National Road and Transportation Research Institute, Sweden, anders.andersson@vti.se
[2]IDA, Linköping University, Sweden, peter.fritzson@liu.se

Abstract

A car model in Modelica has been developed to be used in a new setup for distributed real-time simulation where a moving base car simulator is connected with a real car in a chassis dynamometer via a 500m fiber optic communication link. The new co-simulator set-up can be used in a number of configurations where hardware in the loop can be interchanged with software in the loop. The models presented in this paper are the basic blocks chosen for modeling the system in the context of a distributed real-time simulation, estimating parameters for the powertrain model, the choice of numeric solver, and the interaction with the solver for real-time properties.

Keywords: Modelica, real-time, distributed, communications link

1. Introduction

Vehicles are today becoming increasingly complex systems. An important part of a vehicle is the powertrain which converts energy in a stored form (fuel) to kinetic energy in order to move the vehicle. Recent developments to cope with increasing demands on low fuel consumption and other environmental aspects have introduced several new concepts including e.g. hybrid (combined electrical and fuel) vehicles on the market.

The new vehicles create new challenges. Questions like the following can be relevant:

- "How do we control the charging of the batteries optimally?"
- "Does an automatic gearbox change gears in the way the driver desires?"
- "Does our Human Machine Interface which should help the driver to drive more economically achieve its goal?"

One way to get answers to such questions rather quickly and in a cost efficient way is to use simulation and simulators. Simulation scenarios can be built in many different ways and one approach is to include hardware in the simulation to increase its accuracy [1].

With these benefits in mind cooperation was started between the Swedish National Road and Transport Research Institute (VTI), and Linköping University, including the Vehicular systems group at ISY and the PELAB group at IDA.

The VTI Simulator III (Figure 1) hardware at VTI has been connected via a fiber optic link to the Vehicle propulsion laboratory at Linköping University. A more detailed description of the hardware can be found in [2].

Using the VTI Simulator III it is possible to do a driving experiment, e.g. adjust the road environment or change vehicle models. Since the driving experiment is a controlled experiment, repeatable driving scenarios can be achieved. It is also possible to test safety critical situations in a controlled and safe way.

In the Vehicle propulsion laboratory it is possible to equip different cars with a chassis dynamometers setup where the dynamometers can be used for both accelerating and decelerating the vehicle. The chassis dynamometers are mobile and one person can fit them on a car by moving them around, thus enabling a fast switch between different cars. Connecting these facilities constitute a good platform for testing new powertrain solutions and also improve the fidelity of the VTI Simulator III by using a powertrain in the loop.

Even though new cars could be equipped quickly in the Vehicle propulsion laboratory there is also a need for a model of the setup with which different ideas can be tested in the simulators at VTI before using a vehicle in the chassis dynamometers setup.

Modelica [3], [4], is a modeling language which features properties such as acausal and object-oriented modeling. It has been demonstrated that it is possible to create models applicable for real-time simulation using Modelica [5], [6]. The goal of this work is to build upon an existing hardware model by splitting it into sub-systems and porting parts of it to the Modelica language. The long term aim in this work is to model the complete setup with the car and the system in the Vehicle propulsion laboratory providing a flexible framework for testing distributed simulation scenarios and to run the models in real-time together with VTI Simulator III.

This paper is organized as follows: in Section 2 we give an introduction to the different hardware facilities and the connection between them. In Section 3 we present the Modelica models for the different hardware parts and in

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, 19 April, 2013, University of Nottingham, UK.

EOOLT 2013 website:
http://www.eoolt.org/2013/

Section 4 we show performance tests on the models. The work conducted so far is then summarized in Section 5 which also includes some future work.

2. Hardware Facilities

The facilities used in this study are the VTI Simulator III and the chassis dynamometer lab at Linköping University. To control the vehicle in the chassis dynamometer lab a pedal robot has been constructed. These physical systems will be described in the following sections.

2.1 VTI Simulator III

The Swedish National Road and Transport Research Institute (VTI) is an independent Swedish research institute. The main research areas at VTI are infrastructure, traffic, transportation systems. To conduct research within these areas one important tool is vehicle simulators. The history of moving base simulators at VTI goes back to the late 1970's, and today VTI has three advanced moving base simulators.

The moving base simulators are mostly involved in behavioral studies where research questions such as "How does a driver react in this critical situation?" and "Can my invention detect if a driver is sleepy?" can be investigated.

One of these advanced moving base simulators is the VTI Simulator III which is located in Linköping. This simulator is the one used in this work. A picture of the VTI Simulator III can be seen in Figure 1.

Figure 1. The VTI moving base Simulator III in Linköping.

The motion system in VTI Simulator III has four degrees of freedom where the large outer linear motion can be used for lateral or longitudinal motion. On this motion system the driver is positioned inside a dome in a car cabin which is a production car that has been cut in half to fit the dome. The car cabin is mounted on a vibration table able to produce higher frequency noise reproducing road deformations, e.g. potholes and cracks [7].

The driver view is presented on a 120 degrees arched screen where six projectors are used to produce the front view. Small screens are used as rear view mirrors. The graphics software used is developed at VTI and includes the environments. Sound for the driver is presented using the vehicle speakers complemented with a few extra speakers resulting in a surround sound setup.

The software for controlling the simulation is developed at VTI including scenario logics, interfaces to hardware, graphics and sound, vehicle models and traffic models. The vehicle model used in most of the studies when car driving is of interest is a Fortran model. This model has been developed over several decades and its functionality has been tested in several studies over the years. Thus, the validity of the model has been confirmed repeatedly. The problem with the model is that it takes more and more time to integrate new functions and thus there is a desire to migrate the model to another language.

2.2 Vehicle Propulsion Laboratory

At the Vehicular Systems group at Linköping University, a new chassis dynamometer lab was built in 2011, see [8]. The dynamometers in this lab are mobile and can be adjusted to fit different vehicles sizes and different propulsion systems, e.g. front wheel drive or all-wheel drive. These dynamometers can provide both positive and negative torque while measuring the torque output from the equipped vehicle within 0.1 percent accuracy.

When running the system, the car is attached to the dynamometers by removing the wheels and connecting the wheel hubs to the dynamometers, see Figure 2. When all driving wheel hubs have been connected the exhaust is connected to the ventilation system. At this stage it is now possible to start the system and place a driver in the car. He can now start to drive by turning the car key and pressing the accelerator pedal.

Figure 2. Vehicle mounted at Vehicular Systems chassis dynamometer laboratory at LiU.

2.3 Chassis Dynamometer Vehicle Model

The measured torque output from the vehicle results in vehicle speed and acceleration. Therefore, the dynamometers have to simulate resistance forces equal to those experienced when driving on a road.

These resistance forces come from rolling resistance, F_{roll}, air resistance, F_{air}, and incline resistance (gravity when going uphill or downhill), F_{climb}. Thus, the incline forces are not only resistance uphill but also acceleration downhill, since the chassis dynamometers can provide

both a negative and a positive torque to the wheels. The acceleration, a, of the vehicle is calculated as

where m is the mass of the vehicle and F_{tot} are the total forces acting on the vehicle. F_{prop} is calculated from the measured torques at the wheels using

where T_i are the measured torques at the wheel hubs and r_w is the radius of the wheel.

The resistance forces are given as:

Here c_r is the rolling resistance coefficient, g is the gravitational constant, c_d is aerodynamic resistance constant, A_f is the vehicle front area, A_{air} is the density of the air, v is the speed of the vehicle, v_0 is the relative wind speed and p is the incline of the road. Values for parameters used in this study are shown in Table 1.

Table 1. Used values for the vehicle model parameters.

Parameter	*Value*
M	1401 [kg]
c_d	0.320 [-]
A_f	2.0 [m^2]
c_r	0.01 [-]
r_w	0.3 [m]

Signals measured from the chassis dynamometer are then sent to the VTI Simulator III using the UDP protocol. The list of signals is presented in Table 2.

Table 2. Data sent from the chassis dynamometers.

Signal	*Unit*	*Description*
n_i	[rpm]	Vehicle wheel speed
T_i	[Nm]	Vehicle wheel torque
v_l	[km/h]	Longitudinal velocity
v_v	[km/h]	Vertical vehicle speed
r_{road}	[m]	Road curvature radius
H	[°]	Heading, relative origin
h	[m]	Elevation of road
p	[°]	Incline
d_{TP}	[m]	Distance since start
t_{TP}	[s]	Time since start
T_i	[°C]	Dynamometer temperature
$S_{d,i}$	[-]	Dynamometer status
S	[-]	System status

2.4 Pedal Robot

The driver positioned in the VTI Simulator III can control the vehicle by pressing the accelerator or the brake pedal. The signals emitted by the simulator also have to influence the vehicle in the chassis dynamometers lab and thus a pedal robot was constructed. The pedal robot mimics the driver input in the VTI Simulator III so that the vehicle will receive the driver input. The control parameters are the accelerator pedal position and the brake pressure. The pedal robot is depicted in Figure 3.

One design choice during the construction of the pedal robot was to only include accelerator and brake pedals. Thus, only vehicles with automatic gear are considered. This design choice limits testing of vehicles but the effort needed to add manual gear change to the pedal robot was considered to be too time-consuming. This design choice limits the input signals needed by the pedal robot to accelerator pedal position and brake pedal pressure.

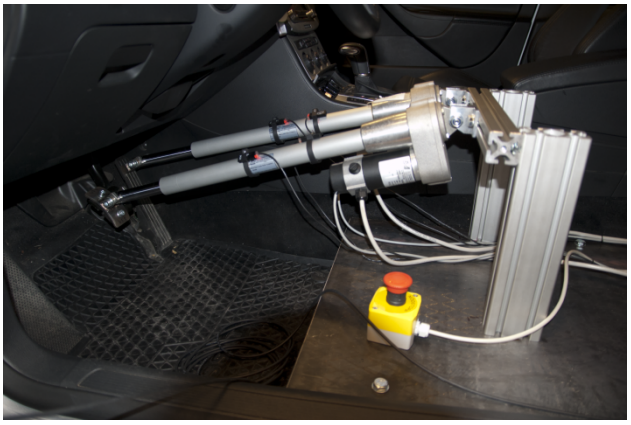


Figure 3. Pedal robot installed in the vehicle in the chassis dynamometer lab.

2.5 Network Performance

To connect the two hardware facilities together, the VTI Simulator III at VTI and the chassis dynamometers at Linköping University, it was decided to use optical fiber. Since the distance between the facilities is approximately 500 m, an optical fiber link was a viable option. The resulting network can thus be considered as a local network.

To test the network connection a round trip time test was performed. Packets which resemble the packages that are sent from the VTI Simulator III software were sent at 200 Hz which is the speed the VTI Simulator III kernel loop runs at. The result from sending one million packets is shown in Table 3.

Table 3. Statistics from the connection between facilities.

Number of packages	1 000 000
Minimum delay	0.20 ms
Maximum delay	2.17 ms
Median delay	0.22 ms
Dropped packets	none
Spikes above 0.5 ms	18

From Table 3 it can be seen that when sending one million packets no packets were lost. It can also be seen that there were very few delays longer than 0.5 ms. Based on the measured performance of the network it was decided to use the efficient UDP packet switching protocol for communication between the involved hardware setups and models during the tests.

3. Modelica Models

Modelica is a language for modeling and simulating complex physical and technical systems. The initiative to start the design of the Modelica modeling language was taken in 1996 in an international effort [15]. The main features of the language are:

- Object oriented approach
- Acausal equation-based modeling
- Hybrid (continuous-time and discrete-time) modeling

Since Modelica allows acausal modeling the model code is close to the physical equations describing the system behavior. This supports a more intuitive modeling process and provides a higher level of abstraction. Various commercial and open-source tools support the design, compilation and simulation of Modelica models. In the course of this work we have tested our models with both OpenModelica [10] and Dymola [11].

When porting the model to Modelica, an important question is what parts of the model to export to which separate subsystems. As for any large model there are alternative choices regarding how to split it into submodels. In this work the solution to this issue was more or less dictated by the hardware we want to model.

Thus the model uses the same UDP network interface as the chassis dynamometers lab, and network packets to and from the VTI Simulator III will have the same format when using a model. The resulting model partitioning consists of a Modelica car model used in the VTI Simulator III and of another Modelica model of the chassis dynamometers setup.

3.1 Powertrain Model

The powertrain model is split into two parts, the engine and the gearbox. To model the engine in the Fortran car model in the VTI Simulator III an engine map is used. This engine map consists of a 12 by 12 matrix where the engine rotational speed and throttle are taken as input and the output is engine torque.

Using such a matrix has been sufficient for many performed studies while still being simple. Thus, such an engine model is implemented in Modelica using a `Modelica.Blocks.Tables.CombiTable2D` model from the Modelica standard library, MSL, with smoothness set to `Modelica.Blocks.Types.Smoothness.LinearSegments`. Instead of using throttle as input to the engine map it was changed to pedal position resulting in

Here is the output torque from the engine, is the accelerator pedal position scaled between zero to one and is the engine rotational speed.

The engine rotational speed is also limited in the model to prevent the engine to infinitely increase rotational speed.

Looking at the gearbox inside the Fortran car model it is constructed using for-loops and `break` statements. The possibility to translate these parts, loops with `break` statements, which are quite nonphysical, to a Modelica model was investigated, but it was decided to create a new basic model instead. The created model uses the following equations between the gearbox and the wheel hubs:

Here $\dot{E}_{clutch}$ is the rotational speed at the clutch, $\dot{E}_{fl}$ and $\dot{E}_{fr}$ are the rotational speeds at the wheel hubs, i_{gear} is the gear ratio from the gearbox and final drive, T_{clutch} is the torque at the clutch and T_{fl} and T_{fr} are the torques at the wheels.

The clutch model's rotational speed depends on the clutch position. The following equations are used for the clutch:

Here is the inertia in the engine, is a modified clutch adjusted dead zones in the clutch and is a first order response time.

This model for the gearbox models a manual gear and the logic for automatic gear changes has to be applied. This would mean that instead of having clutch pedal and gear stick handled by a driver, physical or modeled, the clutch and gear signals are handled by logics with accelerator pedal as input from the driver.

3.2 Estimating Powertrain Model Parameters

In the chassis dynamometer lab there exist signals measuring the responses at the wheels, e.g. engine rotational speed and torques. To add other necessary signals for the parameterization of a powertrain model, an OBD II sensor was used. The OBD II sensor is capable of

logging 5 parameters at a speed of 2-4 Hz which might be too slow for dynamic testing, but for static tests it was deemed sufficient. As the two systems used for measuring data both save time stamps a time synchronization using the Network Time Protocol [12] was performed before the measurements.

Starting with estimating the gear ratios, the chassis dynamometer setup wheel rotational speed and time stamps were logged. From the OBD II sensor engine rotational speed and time stamps were logged. For every gear the driver started at a low speed which was maintained for approximately one minute. The driver then increased speed to another stationary speed. This procedure of increasing speed was repeated two to three times giving a measurement of the gear ratios with both low and high engine rotational speeds.

The measurements from the chassis dynamometers are made at a higher frequency. Thus, to achieve equal positions in time, linear interpolation is used to get logged wheel rotational speed at the same time instances as the engine rotational speed. A least square approximation is then used to estimate the gear ratios and the estimated ratios are shown in Table 4.

Table 4. Measured gear ratios from car mounted in the chassis dynamometer lab.

Gear	*Ratio*
1	16.70
2	10.08
3	6.79
4	4.97
5	3.79
6	3.06

In Figure 4 the relation between the engine rotational speed and the wheel rotational speed using measured gear ratio for gear one is shown.

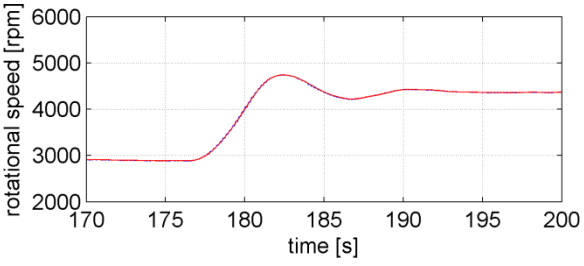


Figure 4. Comparison of engine rotational speed, blue curve, and vehicle wheel speeds multiplied with the gear ratio, red curve, at gear one.

We continue by measuring a static engine map. In this setup we use the OBD II sensor to measure time, engine rotational speed, and accelerator pedal position. Before starting any measurements the engine was run at high load to reduce variations in temperature during the measurements. Measurements for each operating point were done during approximately 30 seconds when the engine torque output had stabilized. The resulting engine map is shown in Figure 5.

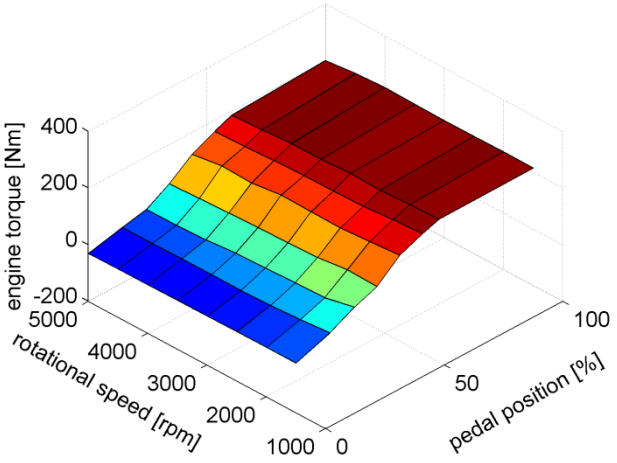


Figure 5. Measured engine map from a car mounted in the chassis dynamometer lab.

In the used cabin in the VTI Simulator III the red zone of the engine rotational speed starts at 6400 rpm with a stop at 7000 rpm. To take this into account the engine map is modified by extrapolating the engine map from 5000 rpm to 6400 rpm and after that linearly reduce output torque to 7000 rpm. At 7000 rpm the output engine torque is independent of pedal position and the engine torque where the driver has released the accelerator pedal is used for every pedal position. In this case -35.5 Nm.

3.3 Pedal Robot Model

We considered the performance of the pedal robot to be accurate and fast enough to neglect the effects from it. Thus, the pedal robot has not been modeled and instead the pedal signals are sent directly to the chassis dynamometers model.

3.4 Model of the Chassis Dynamometer Lab

The complete model of the chassis dynamometer lab consists of three parts. One part handles the longitudinal vehicle model used to calculate the vehicle speed as described earlier in the hardware section. The second part is the vehicle powertrain which from driver pedal input models wheel torque and rotational speed responses as shown. The third part is the brake dynamics. To simulate this setup the complete model had to be extended with a driver model. A simulation of the complete setup is shown in Figure 6 where a calm acceleration is performed.

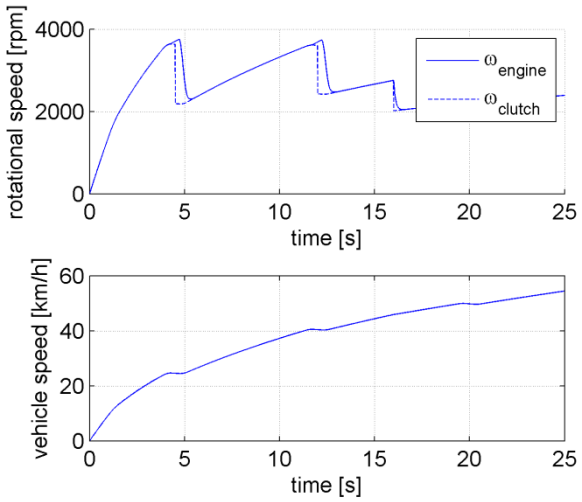


Figure 6. A simulation of the complete chassis dynamometers lab model during a calm acceleration.

The minimum requirement of the complete model is that it has to run at least at 100 Hz because the chassis dynamometers send torque and rotational speed data at this frequency. These signals are important and thus we want the model to send these signals in the same way the hardware would do.

The final chassis dynamometer model has 63 equations where 40 of these are trivial equations. The final model has 8 continuous states.

3.5 Simulator Car Model

In the VTI Simulator III simulator environment different vehicle models can be used. One of these models is a ten degrees of freedom Modelica car model, see [13]. The model has been compiled from Dymola to Simulink in Matlab 7.5 where it was further compiled to be used in an xPC-Target 3.3 environment which is a real-time environment. For further information about xPC-Target see [14].

In the Modelica car model the parts regarding the powertrain have been modified to have the same connections as to the chassis dynamometers model. Since the connections are the same it is possible to use the estimated powertrain model without adjustments in both the Modelica car model and the chassis dynamometers model. The connections also make it possible to run the simulator connected to the chassis dynamometers model in the same way as if it would be connected to the hardware in the chassis dynamometers lab.

3.6 Complete Simulator Setup

The complete simulator setup consists of several components of hardware and Modelica models. An overview of these components is shown in Figure 7.

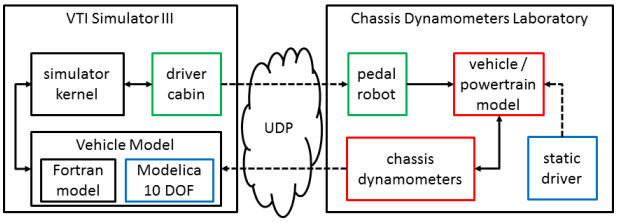


Figure 7. An overview of the components in the simulator environment. Green boxes picture hardware components and blue boxes picture Modelica model components. Red boxes picture components where either hardware or a Modelica model can be used.

Here it can be seen that it is possible to combine these components in different ways. Examples of combinations are:

- VTI Simulator III connected to the pedal robot and the chassis dynamometers.
- VTI Simulator III with the Modelica powertrain model included in the 10 DOF Modelica car model.

It should be noted that the chassis dynamometers and connected vehicle either both are in hardware or software as it is not possible to connect a Modelica powertrain model to the chassis dynamometers.

One component shown in Figure 7 which has not been discussed much is the static driver. By static driver we here mean that the driver has a predefined way of driving, e.g. change gear from gear 1 to 2 at time 5 s. Another typical term used for this kind of component is a drive cycle. Thus, this Modelica model relies heavily on the `time` variable and on `if` statements for controlling driver output such as the accelerator pedal, the clutch pedal, the brake pedal and the gear.

4. Results

For the models we have created there are many aspects to investigate. Our main concern during the initial stages has been the feasibility of real-time implementation. We primarily consider two questions:

- How accurately different solvers simulate the model?
- Will the model manage desired time steps?

In the following sections we discuss how our models perform with respect to these questions.

4.1 Performance of the Chassis Dynamometers Model

We start here by looking at the first question: "How accurately different solvers simulate the model?". As a baseline for comparison the DASSL solver has been used which is compared to the Euler forward solver. Since some signals from the chassis dynamometers are sent at 100 Hz this sets the minimal required speed for the model.

Figure 8 shows the difference in acceleration when using Euler forward or DASSL during the acceleration maneuver shown in Figure 6. Settings used were a tolerance of 1e-6 and 2500 intervals. Simulation environment used was OpenModelica.

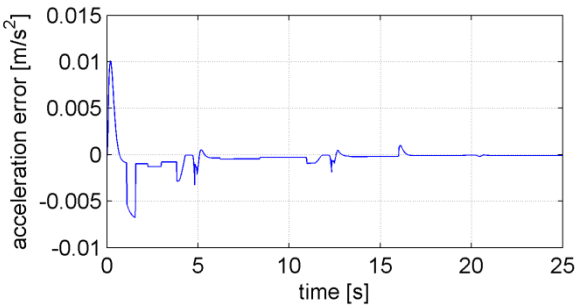


Figure 8. Difference in acceleration when simulations are using Euler forward or DASSL as solvers during an acceleration maneuver.

Here it can be seen that the difference between using DASSL and Euler forward is comparably small. Since the difference between DASSL and Euler forward is so small and we want to keep the setup as simple as possible Euler forward has been chosen for real-time simulation.

The next concern is if it will be possible to obtain 100 Hz performance during a real-time simulation. To give an estimate of this the profiler in OpenModelica, [15], has been used. The results from a run with Euler forward with a time step of 0.01 s are shown in Table 5.

Table 5. Time measurements from model simulation.

Task	*Time*	*Fraction*
Pre-Initialization	0.000464	1.12%
Initialization	0.000118	0.29%
Event-handling	0.000671	1.62%
Creating output file	0.030664	74.24%
Linearization	0.000000	0.00%
Time steps	0.004043	9.79%
Overhead	0.003650	8.84%
Unknown	0.00216	4.11%
Total simulation time	0.041306	100.00%

In Table 5 we can see that event handling and the time steps take approximately 0.0047 s which means that it should be possible to run the model in real-time using Euler forward with a time step of 0.01 s.

5. Conclusion

The established connection between the VTI Simulator III and the chassis dynamometer lab has been developed and this setup shows promising results. For applications it will be interesting to perform more detailed investigations regarding modeling and performance issues.

In this paper we have described and presented our first results on the technical side. A Modelica car model of appropriate complexity has been adjusted for use in both OpenModelica and Dymola. A chassis dynamometer model has been developed which runs in both OpenModelica and Dymola.

The performance profiler in OpenModelica indicates that it is possible to run the model in real-time. This shows that it is promising to continue this work to achieve a real-time simulation with the VTI Simulator III together with a Modelica model of the chassis dynamometer setup.

5.1 Future Work

One main future work is to run the parameterized models in real-time together with the hardware. This should be investigated using the different possible combinations of hardware and software. Additionally, some parts of the model should be improved. For instance, the models for the pedal robot and the brake dynamics have been over-simplified and can be enhanced. A model from accelerator pedal to throttle input could also be further investigated.

Another option to investigate is the use of FMI for real-time simulation [16]. FMI is a standard for exporting a compiled model to C-code packaged into a so-called FMU (Functional Mockup Unit) which can be imported for simulation within another tool.

The chassis dynamometer model has already been compiled to a FMU using OpenModelica. A possible future work is to integrate this FMU with a small cross platform application. The intention is to run the model on a stripped Linux computer through this small cross platform application.

Acknowledgements

The authors would like to thank Tobias Lindell and Per Öberg for the help with measurements in the chassis dynamometers lab. The authors would also like to thank Lena Buffoni for valuable comments and help with the manuscript.

Partial support for this work has been received from SSF in the project HiPo, and from Vinnova in the project RTSIM and in the ITEA2 project MODRIO.

References

[1] X. Hu and E. Azarnasab. From virtual to real – A progressive simulation-based design framework. In *Discrete-Event Modeling and Simulation*, *CRC Press,* 2010.

[2] A. Andersson, P. Nyberg, H. Sehammar, and P. Öberg. Vehicle Powertrain Test Bench Co-Simulation with a Moving Base Simulator Using a Pedal Robot. In SAE World Congress, number 2013-01-0410, Detroit, USA, 2013.

[3] Modelica Association. Modelica Language Specification 3.3, www.modelica.org, May 2012.

[4] Peter Fritzson. *Principles of Object Oriented Modeling and Simulation with Modelica 2.1*, 940 pages, ISBN 0-471-471631, Wiley-IEEE Press. January 2004.

[5] Elmqvist, H., Mattsson, S., & Olsson, H. Real-time simulation of detailed vehicle and powertrain dynamics. *SAE SP, 2004*.

[6] Otter, M., Schlegel, C., & Elmqvist, H. Modeling and realtime simulation of an automatic gearbox using Modelica. *Proceedings of ESS,* 1997.

[7] A. Bolling, J. Jansson, M. Hjort, M. Lidström, et al. An approach for realistic simulation of real road condition in a moving base driving simulator. *ASME/Journal of Computing and Information Science in Engineering*, 11(4), 2011.

[8] P. Öberg, P. Nyberg, and L. Nielsen. A new chassis dynamometer laboratory for vehicle research. In SAE World Congress, number 2013-01-0402, Detroit, USA, 2013.
[9] S. Mattson, H. Elmqvist and M. Otter. Physical system modeling with Modelica. *Control Engineering Practice*, 6(4), 501–510, 1998.
[10] Open Source Modelica Consortium. *OpenModelica Users Guide version 1.9.0beta*, February 2013. www.openmodelica.org.
[11] Dassault Systems. Dymola Users Guide, version 7.1, February 2013. www.dymola.com.
[12] D. L. Mills. Internet Time Synchronization: The Network Time Protocol, IEEE Transactions on Communications, vol. 39, no. 10, 1991.
[13] J. G. Fernández. *A Vehicle Dynamics Model for Driving Simulators.* Master's thesis at Chalmers University of Technology, Göteborg, Sweden, 2012.
[14] P. J. Mosterman, S. Prabhu, A. Dowd, J. Glass, T. Erkkinen, J. Kluza, R. Shenoy. Embedded Real-Time Control via MATLAB, Simulink, and xPC Target.
[15] M. Huhn, M. Sjölund, W. Chen, C. Schulze & P. Fritzson. Tool Support for Modelica Real-time Models In *Proceedings of the 8th Modelica Conference*, Dresden, Germany, March 20-22, 2011.
[16] T. Blochwitz, M. Otter, & M. Arnold. The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In *Proceedings of the 8th Modelica Conference,* (pp. 105–114), 2011.

Appendix

Modelica models for most of the simulated system.

```
model ChassisDynamometerSystem
  StaticDriver driver;
  ChassisDynamometerVehicleModel
   chassis_dynamometer_vehicle_model;
  volvos40.volvos40powertrain powertrain;
equation
  powertrain.throttle = driver.throttle;
  powertrain.clutch = driver.clutch;
  powertrain.gear = driver.gear;
  powertrain.long_vel =
   chassis_dynamometer_vehicle_model.vl;
  connect(powertrain.fl,
    chassis_dynamometer_vehicle_model.fl);
  connect(powertrain.fr,
    chassis_dynamometer_vehicle_model.fr);
end ChassisDynamometerSystem;
```

```
model StaticDriver "driver with
    pre-defined output"
  output Real throttle
   "throttle position scaled [0.0-1.0]";
  output Real clutch
   "clutch positio scaled [0.0-1.0]";
  output Real brake "brake pressure";
  output Integer gear "chosen gear";
  output Real stw_ang
   "steering wheel angle";
protected
  constant Real pi=Modelica.Constants.pi;
  constant Real stwamp = (110 * pi) / 180
    "amplitude of the swd manouvre";
equation
  der(throttle) = if time < 17 then 10 *
    (0.5 - throttle) else 10 * (0.3 -
    throttle);
  clutch = if abs(time - 5) < 1 then
    1 - max(0, min(1, abs(time - 5)))
    elseif abs(time - 10) < 1 then
      1 - max(0, min(1, abs(time - 10)))
    elseif abs(time - 20) < 1 then
      1 - max(0, min(1, abs(time - 20)))
    else 0;
  brake = if time < 20 then 0
    elseif time < 30 then 15000
    elseif time < 40 then 0
    elseif time < 55 then 15000
    else 0;
  gear = if time < 5 then 1
    elseif time < 10 then 2
    elseif time < 20 then 3
    elseif time < 35 then 4
    else 3;
  stw_ang = if time < 10 then 0
    elseif time < 10 + (1 / 0.7 * 3) / 4
      then stwamp * sin(2 * pi * 0.7 *
       (time - 10))
    elseif time < 10 + (1 / 0.7 * 3) / 4 +
      0.5 then -stwamp
    elseif time < 10 + (1 / 0.7 * 4) / 4 +
      0.5 then stwamp * sin(2 * pi * 0.7 *
       (time - 10 - 0.5))
    else 0;

end StaticDriver;
```

```
model ChassisDynamometerVehicleModel
  "Vehicle model used in the Chassis
   Dynamometer setup at LiU"
  package Interfaces =
Modelica.Mechanics.Rotational.Interfaces;
  Interfaces.Flange_a fl "mechanical
    connection to front left wheel";
  Interfaces.Flange_a fr "mechanical
    connection to front right wheel";
  Interfaces.Flange_b rl "mechanical
    connection to rear left wheel";
  Interfaces.Flange_b rr "mechanical
    connection to rear right wheel";
  Modelica.SIunits.Acceleration a(start =
    0) "vehicle acceleration";
  Modelica.SIunits.Velocity v(start = 0)
   "vehicle speed";
  output Real[4] n "wheel rotational
    speeds";
  output Real[4] M "wheel torque";
  output Real vl "vehicle longitudinal
    speed";
  output Real vv "vehicle lateral speed";
  output Real rroad "road curvature
    radius";
  output Real H "vehicle heading";
  output Real h "elevation of road";
  output Real p "incline";
```

```
  output Real d_TP "distance since start";
  output Modelica.SIunits.Time t_TP "time
    since start";
  output Modelica.SIunits.Temperature[4] T
    "dynamometer temperature";
protected
  package SI = Modelica.SIunits;
  constant SI.Mass m = 1401 "vehicle
    mass";
  constant SI.CoefficientOfFriction c_d =
    0.32
    "aerodynamic resistance coefficient";
  constant SI.Area A_f = 2.0 "vehicle
    front area";
  constant SI.CoefficientOfFriction c_r =
     0.001 "rolling friction coefficient";
  constant SI.Length r_w = 0.3 "wheel
    radius";
  constant SI.Acceleration g =
    Modelica.Constants.g_n "gravitational
      constant";
  constant SI.Density rho_air = 1.202 "air
    density at an altitude of 200m";
  Real Ftot "total amount of forces acting
    on the vehicle";
  Real Fprop "propulsion forces";
  Real Froll "rolling resistance forces";
  Real Fair "air resistance forces";
  Real Fclimb "vehicle incline forces";
equation
  a = der(v);
  Ftot = m * a;
  Ftot = Fprop - Froll - Fair - Fclimb;
  Fprop = -(fl.tau + fr.tau + rl.tau +
    rr.tau) / r_w;
  Froll = c_r * m * g;
  Fair = (c_d * A_f * rho_air * v * v)/2;
  Fclimb = 0.0;
  der(fl.phi) = v / r_w;
  der(fr.phi) = v / r_w;
  der(rl.phi) = v / r_w;
  der(rr.phi) = v / r_w;
  //Output
  n[1] = v / r_w;
  n[2] = v / r_w;
  n[3] = v / r_w;
  n[4] = v / r_w;
  M[1] = fl.tau;
  M[2] = fr.tau;
  M[3] = rl.tau;
  M[4] = rr.tau;
  vl = v;
  vv = 0.0 "dummy value";
  rroad = 0.0 "dummy value";
  H = 0.0 "dummy value";
  h = 0.0 "dummy value";
  p = 0.0 "dummy value";
  der(d_TP) = v;
  t_TP = time;
  T[1] = 300.0 "dummy value";
  T[2] = 300.0 "dummy value";
  T[3] = 300.0 "dummy value";
  T[4] = 300.0 "dummy value";
end ChassisDynamometerVehicleModel;
```