Proceedings of the

3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools

Oslo, Norway, October 3, 2010,
in conjunction with MODELS 2010

Editors
Peter Fritzson
Edward Lee
François Cellier
David Broman

Sponsored by PELAB, Department of Computer and Information Science, Linköping University

EOOLT

2010

Copyright

Series: Linköping Electronic Conference Proceedings
Number 47
ISSN (print): 1650-3686
ISSN (online): 1650-3740
http://www.ep.liu.se/ecp/047/

Printed by LiU-Tryck, Linköping, 2010

Table of Contents

Preface

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

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

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

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

On behalf of the program committee, the Program Chairs would like to thank all those who submitted papers to EOOLT'2010. Many thanks to the program committee for reviewing the papers. The venue for EOOLT'2010 was Oslo, Norway, in conjunction with the MODELS'2010 conference.

Linköping, September 2010

Peter Fritzson
Edward Lee
François Cellier
David Broman

Program Chairs

Peter Fritzson, Chair	Linköping University, Linköping, Sweden
Edward Lee, Co-Chair	U.C. Berkeley, USA
François Cellier, Co-Chair	ETH, Zurich, Switzerland
David Broman, Co-Chair	Linköping University, Linköping, Sweden

Program Committee

Peter Fritzson, Chair	Linköping University, Linköping, Sweden
Edward Lee, Co-Chair	U.C. Berkeley, USA
François Cellier, Co-Chair	ETH, Zurich, Switzerland
David Broman, Co-Chair	Linköping University, Linköping, Sweden
Bernhard Bachmann	University of Applied Sciences, Bielefeld, Germany
Bert van Beek	Eindhoven University of Technology, Netherlands
Felix Breitenecker	Technical University of Vienna, Vienna, Austria
Jan Broenink	University of Twente, Netherlands
Peter Bunus	Linköping University, Linköping, Sweden
Francesco Casella	Politecnico di Milano, Italy
Hilding Elmqvist	Dassault Systèmes, Lund, Sweden
Olaf Enge-Rosenblatt	Fraunhofer Institute for Integrated Circuits, Dresden, Germany
Petter Krus	Linköping University, Linköping, Sweden
Sven-Erik Mattsson	Dassault Systèmes, Lund, Sweden
Jacob Mauss	QTronic GmbH, Berlin, Germany
Pieter Mosterman	MathWorks, Inc., Natick, MA, USA
Toby Myers	Griffith University, Brisbane, Australia
Henrik Nilsson	University of Nottingham, Nottingham, United Kingdom
Dionisio de Niz Villasensor	Carnegie Mellon University, Pittsburgh, USA
Hans Olsson	Dassault Systèmes, Lund, Sweden
Martin Otter	DLR Oberpfaffenhofen, Germany
Chris Paredis	Georgia Institute of Technology, Atlanta, Georgia, USA
Peter Pepper	TU Berlin, Berlin, Germany
Adrian Pop	Linköping University, Linköping, Sweden
Nicolas Rouquette	NASA Jet Propulsion Laboratory, USA
Peter Schwarz	Fraunhofer Inst. for Integrated Circuits, Dresden, Germany
Christian Sonntag	TU Dortmund, Dortmund, Germany
Martin Törngren	KTH, Stockholm, Sweden
Alfonso Urquía	National University for Distance Education, Madrid, Spain
Hans Vangheluwe	McGill University, Montreal, Canada
Dirk Zimmer	DLR Oberpfaffenhofen, Germany
Johan Åkesson	Lund University, Lund, Sweden

Workshop Organization

Peter Fritzson	Linköping University, Linköping, Sweden
David Broman	Linköping University, Linköping, Sweden
Edward Lee	U.C. Berkeley, USA
François Cellier	ETH, Zurich, Switzerland

Execution of UML State Machines Using Modelica

Wladimir Schamai[1], Uwe Pohlmann[2], Peter Fritzson[3], Christiaan J.J. Paredis[4], Philipp Helle[1], Carsten Strobel[1]

[1]EADS Innovation Works, Germany
[2]University of Paderborn, Department of Computer Science, Germany
[3] Linköping University, PELAB – Programming Environment Lab, Sweden
[4]Georgia Institute of Technology, Atlanta, USA

Abstract

ModelicaML is a UML profile for the creation of executable models. ModelicaML supports the Model-Based Systems Engineering (MBSE) paradigm and combines the power of the OMG UML standardized graphical notation for systems and software modeling, and the simulation power of Modelica. This addresses the increasing need for precise integrated modeling of products containing both software and hardware. This paper focuses on the implementation of executable UML state machines in ModelicaML and demonstrates that using Modelica as an action language enables the integrated modeling and simulation of continuous-time and reactive or event-based system dynamics. More specifically, this paper highlights issues that are identified in the UML specification and that are experienced with typical executable implementations of UML state machines. The issues identified are resolved and rationales for design decisions taken are discussed.

Keywords UML, Modelica, ModelicaML, Execution Semantics, State Machine, Statechart

1 Introduction

UML [2], SysML [4] and Modelica [1] are object-oriented modeling languages. They provide means to represent a system as objects and to describe its internal structure and behavior. UML-based languages facilitate the capturing of information relevant to system requirements, design, or test data by means of graphical formalisms, crosscutting constructs and views (diagrams) on the model-data. Modelica is defined as a textual language with standardized graphical annotations for icons and diagrams, and is designed for the simulation of system-dynamic behavior.

3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, October, 2010, Oslo, Norway.

EOOLT 2010 website: http://www.eoolt.org/2010/

1.1 Motivation

By integrating UML and Modelica the strength of UML in graphical and descriptive modeling is complemented with the Modelica formal executable modeling for system dynamic simulation. Conversely, Modelica will benefit from using the selected subset of the UML-based graphical notation (visual formalisms) for editing, visualizing and maintaining Modelica models.

Graphical modeling, as promoted by the OMG [12], promises to be more effective and efficient regarding editing, human-reader perception of models, and maintaining models compared to a traditional textual representation. A unified standardized graphical notation for systems modeling and simulation will facilitate the common understanding of models for all parties involved in the development of systems (i.e., system engineers, designers, and testers; software developers, customers or other stakeholders).

From a simulation perspective, the behavior described in the UML state machine is typically translated into and thereby limited to time-discrete or event-based simulations. Modelica enables mathematical modeling of hybrid (continuous-time and discrete-time dynamic description) simulation. By integrating UML and Modelica, UML-based modeling will become applicable to the physical-system modeling domain, and UML models will become executable while covering simulation of hardware and software, with integrated continuous-time and event-based or time-discrete behavior. Furthermore, translating UML state machines into executable Modelica code enables engineers to use a common set of formalisms for behavior modeling and enables modeling of software parts (i.e., discrete or event-based behavior) to be simulated together with physical behavior (which is typically continuous-time behavior) in an integrated way.

One of the ModelicaML design goals is to provide the modeler with precise and clear execution semantics. In terms of UML state machines this implies that semantic variation points or ambiguities of the UML specification have to resolved.

The main contribution of this paper is a discussion of issues that were identified when implementing UML state machines in ModelicaML. Moreover, proposals for the resolution of these issues are presented. The issues identified or design decisions taken are not specific to the ModelicaML state machines implementation. The questions addressed in this paper will most likely be raised by anyone who intends to generate executable code from UML state machines.

1.2 Paper Structure

The rest of this paper is structured as follows: Chapter 2 provides a brief introduction to Modelica and ModelicaML and gives an overview of related research work. Chapter 3 describes how state machines are used in ModelicaML and highlights which UML state machines concepts are supported in ModelicaML so far. Chapter 4 discusses the identified issues and explains both resolution and implementation in ModelicaML. Chapter 5 provides a conclusion.

2 Background and Related Work

2.1 The Modelica Language

Modelica is an object-oriented equation-based modeling language that is primarily aimed at physical systems. The model behavior is based on ordinary and differential algebraic equation (OAE and DAE) systems combined with discrete events, so-called hybrid DAEs. Such models are ideally suited for representing physical behavior and the exchange of energy, signals, or other continuous-time or discrete-time interactions between system components.

2.2 ModelicaML – UML Profile for Modelica

This paper presents the further development of the Modelica Graphical Modeling Language (ModelicaML [15]), a UML profile for Modelica. The main purpose of ModelicaML is to enable an efficient and effective way to create, visualize and maintain combined UML and Modelica models. ModelicaML is defined as a graphical notation that facilitates different views (e.g., composition, inheritance, behavior) on system models. It is based on a subset of UML and reuses some concepts from SysML. ModelicaML is designed for Modelica code generation from graphical models. Since the ModelicaML profile is an extension of the UML meta-model it can be used as an extension for both UML and SysML[1]. The tools used for modeling with ModelicaML and generating Modelica code can be downloaded from [15].

[1] SysML itself is also a UML Profile. All ModelicaML stereotypes that extend UML meta-classes are also applicable to the corresponding SysML elements.

2.3 Related Work

In previous work, researchers have already identified the need to integrate UML/SysML and Modelica, and have partially implemented such an integration. For example, in [7] the basic mapping of the structural constructs from Modelica to SysML is identified. The authors also point out that the SysML Parametrics concept is not sufficient for modeling the equation-based behavior of a class. In contrast, [9] leverages the SysML Parametrics concept for the integration of continuous-time behavior into SysML models, whereas [8] presents a concept to use SysML to integrate models of continuous-time dynamic system behavior with SysML information models representing systems engineering problems and provides rules for the graph-based bidirectional transformation of SysML and Modelica models.

In Modelica only one type of diagram is defined: the *connection diagram* that presents the structure of a class and shows class-components and connections between them. Modelica does not provide any graphical notation to describe the behavior of a class. In [13] an approach for using the UML-based notation of a subset of state machines and activity diagrams for modeling the behavior of a Modelica class is presented.

Regarding state machines, a list of general challenges with respect to regarding statecharts is presented in [10] and a summary on existing statechart variants is provided. In [11] the fact is stressed that different statechart variants are not compatible even though the syntax (graphical notation) is the same. It is also pointed out that the execution semantics strongly depend on the implementation decisions taken, which are not standardized in UML.

Few implementations of the translation of statecharts into Modelica exist ([6], [5][2]). However, none implements a comprehensive set of state machines concepts as defined in the UML specification.

The main focus of this paper is the resolution of issues related to the execution semantics of UML state machines. A detailed description of the execution semantics of the implementation of UML state machines in ModelicaML and additional extensions are provided in [15] and are out of the scope of this paper.

3 State Machines in ModelicaML

3.1 Simple Example

Assume one would like to simulate the behavior defined by the state machines depicted in Figure 1 using Modelica. This state machine defines part of the behavior of the

[2] StateGraph [5] uses a different graphical notation compared to the UML notation for state machines.

class SimpleStateMachine. In UML, this class is referred to as the *context* of StateMachine_0.

The UML graphical notation for state machines consists of rectangles with rounded corners representing *states*, and edges representing *transitions* between states. Transitions can only be executed if appropriate *triggers* occur and if the associated *guard* condition (e.g., **`[t>1 and x<3]`**) evaluates to true. If no triggers are defined then an empty trigger, which is always activated, is assumed. In addition, transitions can have *effects* (e.g., **`/x:=1`**).

The UML semantics define that a state machine can only be in one of the (simple) states in a region[3] at the same instance of time. The filled circle and its outgoing transition mean that, by default, (i.e. when the execution is started) the state machine is in its initial state, *State_0* (i.e., when the execution is started). The expected execution behavior is as follows:

- When the execution is started the state machine is in its initial state, *State_0*.
- As soon as the guard condition, **`[t>1 and x<3]`**, is true, *State_0* is exited, the transition effect, **`x := 1`**, is executed and *State_1* is entered. The state machine is again in a stable configuration that is referred to as an *active configuration*.
- As soon as the guard condition, **`t>1.5 and x>0`**, becomes true, *State_1* is exited, the effect, **`x:=2`**, is executed and *State_2* is entered.
- As soon as the condition **`x>1`** becomes true, *State_2* is exited, the transition effect, **`x:=3`**, is executed and *State_0* is entered.

Figure 2 shows Modelica code that performs the behavior described above. Figure 3 presents the visualized simulation results.

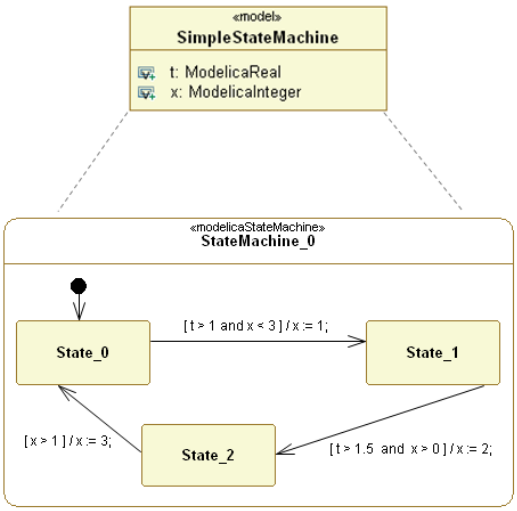

Figure 1: State machine defines part of class behavior

```
model SimpleStateMachine
  Boolean State_0 "State_0 representation";
  Boolean State_1 "State_1 representation";
  Boolean State_2 "State_2 representation";
  Integer x "discrete variable";
  Real t "continuous variable";
 equation //Code for continuous integration of t
    der(t)=time;

 algorithm //Code for StateMachine_0
  when initial() then
      State_0:=true "Activation of the initial state";
  end when;
  //Transition from State_0 to State_1
  if pre(State_0) and t > 1 and x < 3 then
      State_0:=false "Deactivation of state";
      x:=1 "Transistion effect";
      State_1:=true "Activation of state";
  //Transition from State_1 to State_2
  elseif pre(State_1) and t > 1.5 and x > 0 then
      State_1:=false "Deactivation of state";
      x:=2 "Transistion effect";
      State_2:=true "Activation of state";
  //Transition from State_2 to State_0
  elseif pre(State_2) and x > 1 then
      State_2:=false "Deactivation of state";
      x:=3 "Transistion effect";
      State_0:=true "Activation of state";
  end if;
end SimpleStateMachine;
```

Figure 2: Corresponding Modelica code

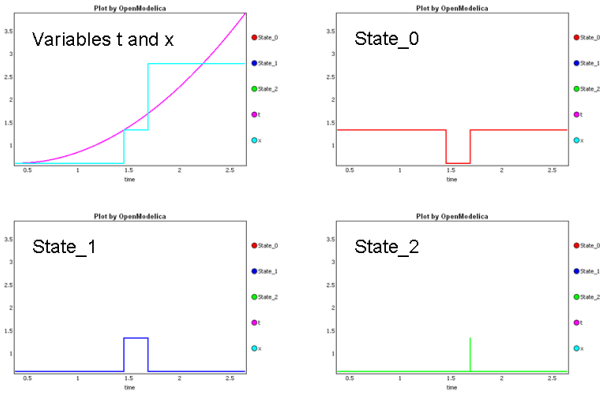

Figure 3: Simulation results

[3] If a composite state or multiple regions are defined for a state machine, then it means that the state machine is in multiple (simple) states at the same time.

3.2 State Machines in ModelicaML

UML2 defines two types of state machines: behavior state machines and protocol state machines. Behavior state machines are used to model parts of class behavior. ModelicaML state machines are derived from UML behavior state machines. Compared to behavior state machines, protocol state machines are limited in terms of expressiveness and are tailored to the need to express protocols or to define the lifecycle of objects. Since this is not the main intended area of application for ModelicaML, protocol state machines are not taken into account. Consequently, none of the chapters of the UML specification that address the protocol state machines are considered.

State machines are used in ModelicaML to model parts of class behavior. A behavioral class (i.e., Modelica class, model or block) can have 0..* state machines as well as 0..* other behaviors (e.g. equation or algorithm sections). This is different from a typical UML application where usually only one state machine is used to represent the *classifierBehavior*. In ModelicaML it is possible to define multiple state machines for one class which are executed in parallel. This possibility allows the modeler to structure the behavior by separating it into individual state machines. When multiple state machines are defined for one class they are translated into separate algorithm sections in the generated Modelica code. This implies that they cannot set the same class variables (e.g., in entry/do/exit or transition effects) because it would result in an over-determined system.

UML state machines are typically used to model the reactive (event-based) behavior of objects. Usually an event queue is implemented that collects all previously generated events which are then dispatched one after the other (the order is not fully specified by UML) to the state machine and may cause state machine reactions. Strictly speaking, a UML state machine reacts (is evaluated) only when events are taken from the event queue and dispatched to the state machine.

ModelicaML uses Modelica as the execution (action) language. In contrast to the typical implementations of UML state machines, the Modelica code for a ModelicaML state machine is evaluated continuously, namely, after each continuous-time integration step and, if there are event iterations, at each event iteration. Event iterations concept is defined by Modelica ([1], p.25) as follows: "*A new event is triggered if at least for one variable v "pre(v) <> v" after the active model equations are evaluated at an event instant. In this case, the model is at once re-evaluated. This evaluation sequence is called "event iteration". The integration is restarted, if for all v used in pre-operators the following condition holds: "pre(v) == v"*.". The definition of pre(v) is the following: "*Returns the "left limit" $y(t^{pre})$ of variable y(t) at a time instant t. At an event instant, $y(t^{pre})$ is the value of y after the last event iteration at time instant t ...*" (see [1], p.24).

Furthermore, the following definition is essential to understand the execution of Modelica code (see Modelica specification [1], p.84): "*Modelica is based on the synchronous data flow principle and the single assignment rule, which are defined in the following way:*

- *All variables keep their actual values until these values are explicitly changed. Variable values can be accessed at any time instant during continuous integration and at event instants.*
- *At every time instant, during continuous integration and at event instants, the active equations express relations between variables which have to be fulfilled concurrently (equations are not active if the corresponding if-branch, when-clause or block in which the equation is present is not active).*
- *Computation and communication at an event instant does not take time. [If computation or communication time has to be simulated, this property has to be explicitly modeled].*
- *The total number of equations is identical to the total number of unknown variables (= single assignment rule)."*

3.2.1 Transformation of State Machines to Modelica Code

Two main questions need to be considered when translating a ModelicaML state machine into Modelica:

- The first question is whether a library should be used or whether a code generator should be implemented. One advantage of using a library is that the execution semantics can be understood simply by inspecting the library classes. In order to understand the execution semantics of generated code one also needs to understand the code generation rules.
- The behavior of a ModelicaML state machine can be expressed by using Modelica algorithmic code or by using equations. Note that the statements inside an algorithm section in Modelica are executed exactly in the sequence they are defined. This is different from the equation sections which are declarative so that the order of equations is independent from the order of their evaluation. The Modelica StateGraph library [5] uses equations to express behavior that is similar to the UML state machine behavior.

In ModelicaML, the behavior of one state machine is translated into algorithmic code that is generated into one algorithm section of the containing class[4]. The rationale

[4] A Modelica class can have 0..* algorithm sections.

for the decision to implement a specific code generator instead of implementing a library and to use algorithm statements instead of equations is the following:

- The behavior expressed by a state machine is always causal. There is no need to use the acausal modeling capability of Modelica. By using algorithm (with predefined causality) no sorting of equations is required.
- Furthermore, for the implementation of inter-level transitions, i.e. transitions which cross states hierarchy borders, the deactivation and activation of states and the execution sequence of associated actions (exit/entry action of states or state transitions effects) has to be performed in an explicitly defined order. This is hard to achieve when using equations that are sorted based on their data dependencies.

3.2.2 Combining Continuous-Time and Discrete-Time Behavior

The fact that a ModelicaML state machine is translated into algorithmic Modelica code implies that all actions (transition effects, or entry/do/exit actions of states) can only be defined using algorithmic code. Hence, it is not possible to insert equations into transition effects or entry/do/exit actions of states. However, it is possible to relate the activation of particular equations based on the activation or deactivation of state machine states. This is supported by the dedicated `IsInState()`-macro in ModelicaML. Vice versa, state machines can react on the status of any continuous-time variable.

3.2.3 Event Processing (Run-To-Completion Semantics Applicability)

UML, [2] p. 565, defines the *run-to-completion* semantics for processing events. When an event is dispatched to a state machine, the state machine must process all actions associated with the reaction to this event before reacting to further events (which might possibly be generated by the transitions taken). This definition implies that, even if events occur simultaneously, they are still processed sequentially. In practice, this requires an implementation of an event queue that ultimately prevents events from being processed in parallel. This can lead to ambiguous execution semantics as is pointed out in section 4.2.

The problem with event queues, as discussed in section 4.2, does not exist in ModelicaML.If events occur simultaneously (at the same simulated time instant or event iteration) in ModelicaML state machines they are processed (i.e. consumed) in parallel in the next evaluation of the state machine.

However, the definition of the *run-to-completion* semantics is still applicable to Modelica and, thus, to ModelicaML state machines in the sense that when an event has occurred a state machine first finishes its reactions to this event before processing events that are generated during these reactions.

4 State Machines Execution Semantics Issues Discussion

4.1 Issues with Instantaneous States: Deadlocks (Infinite Looping)

In Modelica the global variable *time* represents the simulated real time. Computations at event iterations do not consume simulated time. Hence, states can be entered and exited at the same simulated point in time. For instance, Figure 3 shows State_2 being entered and exited at the same simulated point in time. However, in ModelicaML, a state cannot be entered and exited during the same event iteration cycle, i.e., variables that represent states cannot be set and unset in the same event iteration cycle. This is ensured by using the *pre(state activation status)* function in the conditions of state transition code. This enforces that the entire behavior first reacts to an event before reacting to the events that are generated during event iterations (i.e. re-evaluation of the equation system).

When instantaneous states are allowed it is possible to model deadlocks that lead to an infinite loop at the same simulated point in time. Consider Figure 4 (left): The behavior will loop infinitely without advancing the simulated time and a simulation tool will stop the simulation and report an issue.

If such a behavior is intended and the state machine should loop continuously and execute actions, the modeler can break the infinite looping by adding a time delay[5] to one of the involved transitions (see the right state machine in Figure 4). In doing so the simulation time is advanced and the tool will continue simulating.

In large models deadlocks can exist which are not as obvious as in the simple example depicted in Figure 4. Infinite looping is often not modeled on purpose and is hard to prevent or to detect. This issue is subject to future research in ModelicaML.

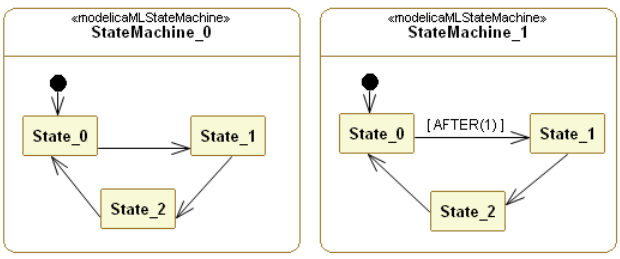

Figure 4: Deadlocks (infinite looping) example

[5] AFTER(expression) is a ModelicaML macro. It is expanded to the guard condition `state_local_timer > expression`.

4.2 Issues With Concurrency When Using Event Queues

Consider the state machine in Figure 5 modeled in IBM Rational Rhapsody [14]. The events *ev1* and *ev2* are generated simultaneously at the same time instant when entering the initial states of both regions. However, it is not obvious to the modeler in which order the generated events are dispatched to the state machine.

When the simulation is started Rhapsody shows that the events are generated and put into the event queue in the following order: *ev1, ev2*. When restarting the simulation the order of events in the queue will always be the same. Obviously, there is a mechanism that determines the order of region executions based on the model data.

Next, these events are dispatched to the state machine one after the other. First the event *ev1* is dispatched and the transition to *state_1* is executed, then the event *ev2* is dispatched and the transition from *state_1* to *state_2* is executed, as shown in Figure 6.

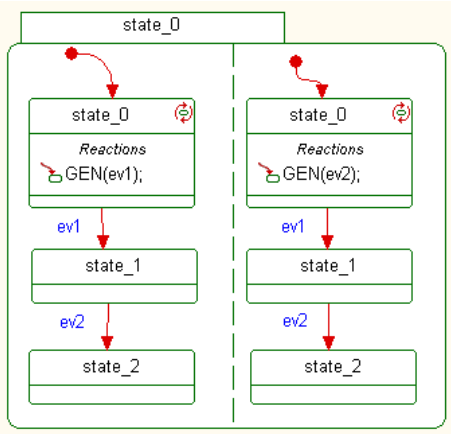

Figure 5: Events queue issue 1

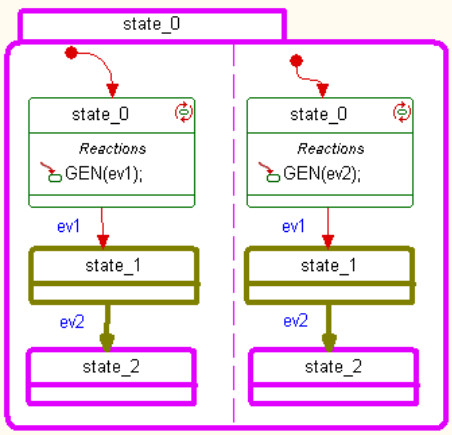

Figure 6: Events queue issue 1 simulation. The state machine ends up in state_2 in both regions.

According to this behavior the occurrence of *ev2* is delayed. However, with the state machine in *state_1* no *ev2* occurs. The event *ev2* occurs when the state machine is in *state_0*. This behavior seems to be similar to the concept of *deferred events* described in the UML specification ([2], p.554): "*An event that does not trigger any transitions in the current state, will not be dispatched if its type matches one of the types in the deferred event set of that state. Instead, it remains in the event pool while another non-deferred event is dispatched instead. This situation persists until a state is reached where either the event is no longer deferred or where the event triggers a transition.*".

Consider Figure 7. It shows a slightly modified state machine. The event e*v2* is generated inside the left region and the *ev1* is generated inside the right region. The order of events in the queue is now reversed: *ev2, ev1*.

Figure 8 shows the simulation result. In contrast to the assumption above, the event *ev2* is not deferred. It is dispatched to the state machine and discarded after the transition to *state_1* is taken. The state machine finally stays in *state_1*, which is a different behavior than in Figure 6.

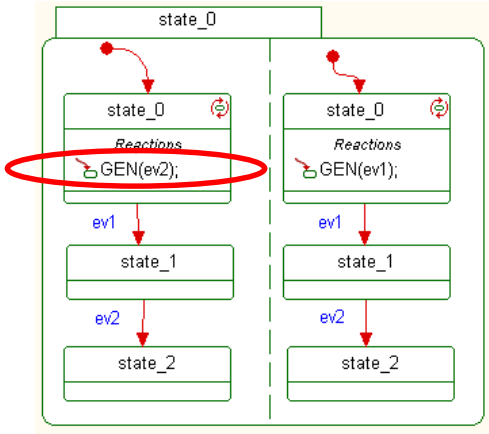

Figure 7: Events queue issue 2

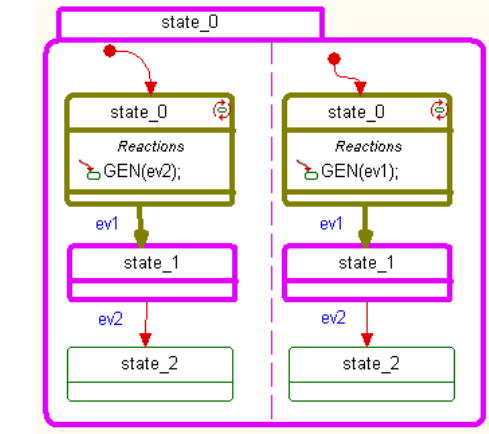

Figure 8: Events queue issue 2 simulation. The state machine ends up in state_1 in both regions.

Along with the fact that the modeler cannot control the execution order of the parallel regions (this issue is ad-

dressed in section 4.3) and, thus, the order in which events are generated, the main issue here is that it leads to behavior that is unpredictable and cannot be expected from the modeler's perspective.

Figure 9 shows the same model in ModelicaML. In contrast to the examples above, regardless of whether the event *ev2* is generated in the right region or in the left region, the execution behavior is the same – the state machine always ends up in *state_1* in both regions.

This is because the events that occur simultaneously are processed (i.e., dispatched and consumed) in parallel during the next state machine evaluation cycle. When the transitions from the states *state_0* to *state_1* in both regions are executed, event *ev2* is also consumed and the state machine stays in both states *state_1* because with the state machines in states *state_1* in both regions no event *ev2* is generated.

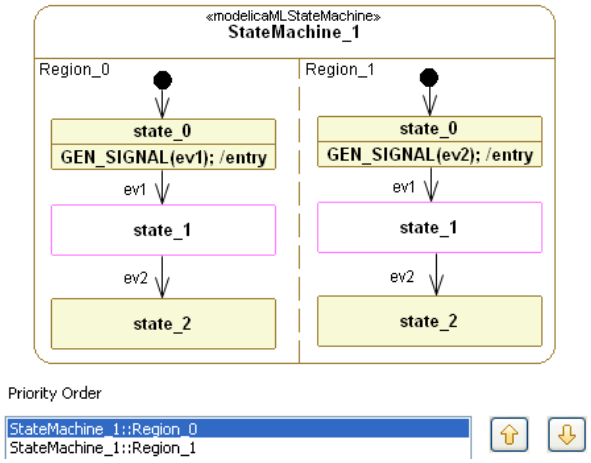

Figure 9: Same model in ModelicaML. The state machine ends up in state_1 in both regions.

4.3 Issue with Concurrent Execution in Regions

Regions are executed in parallel, i.e. at the same simulated time instant. However, the corresponding Modelica code in the algorithm section is still sequential (procedural). To ensure determinism and make the semantics explicit to the modeler, in ModelicaML each region is automatically given a priority relative to its neighboring regions (a lower priority number implies higher execution order priority). This may be necessary when there are actions that set the same variables in multiple parallel regions[6], as illustrated in Figure 10.

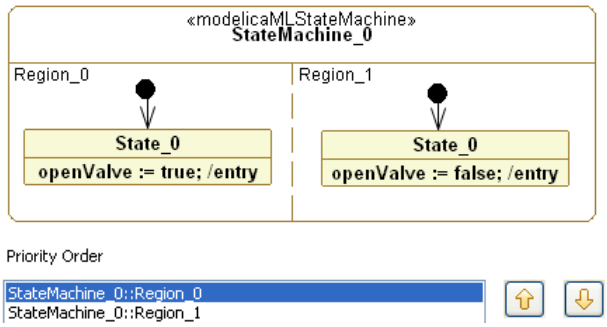

Figure 10: Definition of priority for parallel regions

Since *Region_0* is given a higher execution priority, it will be executed prior to *Region_1*, which has lower priority. The result is that `openValve` is set to `false`. Note that if *State_0* in *Region_0* was a composite state, then its internal behavior would also be executed before the behavior of *State_0* in *Region_1*.

Priorities are set by default by the ModelicaML modeling tool. The modeler can change priorities and, in doing so, define the execution sequence explicitly to ensure the intended behavior.

The regions priority definition is also used for exiting and entering composite states as well as inter-level transitions as discussed in sections 4.5 and 4.6.

4.4 Issues with Conflicting Transitions

When a state has multiple outgoing transitions and trigger and guard conditions overlap, i.e. they can be true at the same time, then the transitions are said to be in conflict ([2], p.566): "*Two transitions are said to conflict if they both exit the same state, or, more precisely, that the intersection of the set of states they exit is non-empty. Only transitions that occur in mutually orthogonal regions may be fired simultaneously.*" When triggers are defined for a transition, then there is no issue with overlapping guard conditions because simultaneous events are not processed in parallel in UML. This is different in ModelicaML because events are processed in parallel and can overlap. However, it is not clear from the UML specification what should happen if conflicting transitions do not have any triggers but only have guard conditions defined, that can evaluate to true at the same time. This issue is addressed in section 4.4.1

Furthermore, in case the conflicting transitions are at different hierarchy levels, UML defines the following: "*In situations where there are conflicting transitions, the selection of which transitions will fire is based in part on an implicit priority. These priorities resolve some transition conflicts, but not all of them. The priorities of conflicting transitions are based on their relative position in the state hierarchy.*" This issue is described in section 4.4.2.

[6] This example is artificial and is meant for illustration purposes only. Normally, modeling such behaviour will probably be avoided.

4.4.1 Priorities for State-Outgoing Transitions

Consider the state machine in Figure 11. If *x* and *y* are greater than *2* at the same time both guard conditions evaluate to true. In ModelicaML, which transition will be taken then is determined by the transition execution priority defined by the modeler.

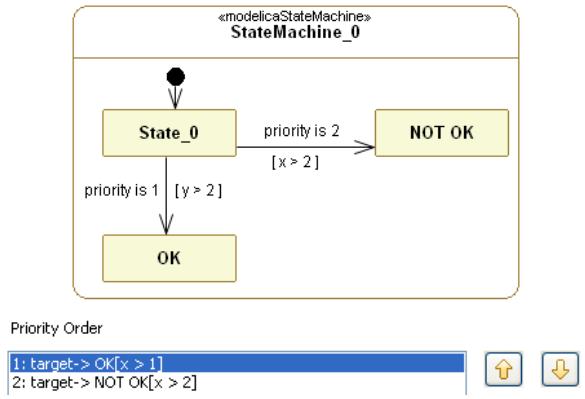

Figure 11: Priorities definition for state-outgoing transitions

As for regions (discussed in section 4.3), conflicting transitions coming out of a state are prioritized in ModelicaML. Priorities for transitions are set by default by the modeling tool. The modeler can change priorities and thereby ensure deterministic behavior.

4.4.2 Priority Schema for Conflicting Transitions at Different State Hierarchy Levels

UML defines the priority schema for conflicting transitions that are at different levels as follows (see p.567): "… *By definition, a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states."* No rationale is documented for this decision as pointed out in [11].

Consider the state machine in Figure 12. What should happen when *x* and *y* are greater than *2* at the same time? This case is not addressed in the UML specification because no triggers are defined for these transitions. One possible answer could be: Transition to state *NOT OK* is taken. Another answer could be: Transition to state *NOT OK* is taken and then transition to state *OK* is taken. Yet another answer could be: Transition to state *OK* is taken and since *State_0* is deactivated no further reaction inside *State_0* is to be expected. The latter is implemented in ModelicaML.

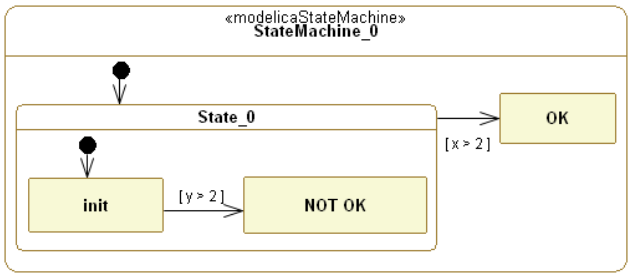

Figure 12: Transition at higher levels have higher priority

In ModelicaML the priority scheme is different from UML. In ModelicaML, outgoing transitions of a composite state have higher execution priority than transitions inside the composite state. The rationale for this decision is as follows:

- This semantics is more intuitive and clear. For example, if states are described using sub-state machines (presented in their own diagrams), the modeler cannot know if a particular transition is taken unless he or she has inspected all lower-level sub-state machines diagrams. The UML priority scheme can also lead to behavior where the composite state is never exited because events are consumed at some level further down the hierarchy of the composite state.
- The ModelicaML priority scheme reduces the complexity of the code generator. For example, as already mentioned above, event queues are not used in Modelica. To avoid that transitions of a composite state consume events that are already consumed inside the composite states, events need to be marked or removed from the queue. Such an implementation would drastically increase the size and complexity of the code generator as well as of the generated Modelica code.

4.5 Issues with Inter-Level Transitions

This section discusses issues concerning the execution order of actions as a result of transition executions. Actions can be *entry/do/exit* actions of states or *effect* actions of transitions. The order in which actions are executed is important when actions are dependable, i.e. if different actions set or read the same variables.

Consider the state machine in Figure 13. Assume that each state has entry and exit actions, and that *Region_0* is always given the highest priority and *Region_x* the lowest.

When the state machine is in state *a* and *cond1* is true, the question is in which order the states are activated and the respective entry actions are executed. This case is not addressed in the UML specification.

From the general UML state machine semantics definition we can deduce that sub-states cannot be activated as long as their containing (i.e. composite) state is not activated. For example, it is clear that the state *b* has to be

activated before the states *c* and *i* can become active. Furthermore, we can argue that since the modeler explicitly created an inter-level transition the state *e2* should be activated first, i.e. before states in neighboring regions at the same (i.e. *f*) or at a higher state hierarchy level (i.e. *g, h, or i*). Hence, when the inter-level transition from state *a* to state *e2* is taken the partial states activation sequence and its resulting order of entry-actions execution should be: *b, c, d, e2*. However, in which sequence shall the states *f, g, h* and *i* be activated? Possible answers are: *i, h, g, f,* or *i, g, h, f* , or *f, g, h, i*, or *f, h, g, i*.

In ModelicaML, this issue is resolved as follows: First all states containing the target state and the target state itself are activated. Next, based on the region execution priority, the initial states from neighboring regions are activated. Since the priority in this example is defined for regions from left (highest) to right (lowest) for each composite state, the activation order would be *b, c, d, e2, f, g, h, i*. Vice versa, if the region priority would be defined the other way around (from right to left) the activation order would be *b, c, d, e2, i, h, g, f*.

A similar issue exists regarding the transition to state *a* when the state machine is in state *e2* and when *cond2* is true. Here the states deactivation and exit actions execution order are involved. The resulting deactivation sequence in ModelicaML would be: *e2, f, d, g* and *h* (based on the region priority definition), c, *i* and *b*.

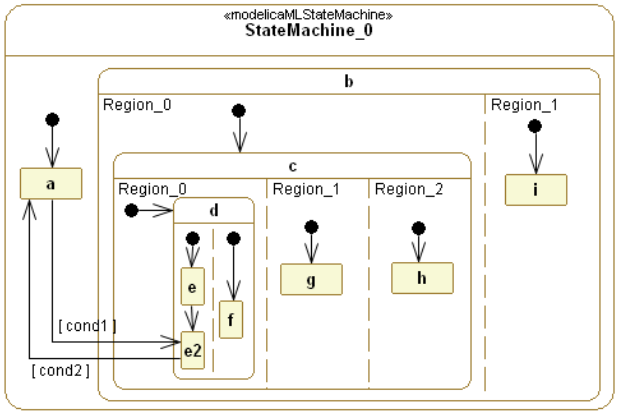

Figure 13: Inter-level transition example

4.6 Issues with Fork and Join

Like section 4.5, this section addresses issues regarding the activation and deactivation of states. However, in this case, UML fork and join constructs are regarded. Consider the state machine on Figure 14. With the state machine in state *a* the questions are:

- In which sequence are states *b, c, d, e,* and *f* activated when the transitions (fork construct) from state *a* is executed?
- In which sequence are states *b, c, d, e,* and *f* deactivated when the transitions (join construct) to state *g* are executed?

This case is also not addressed in the UML specification. In ModelicaML, first the parent states of the transition target-states are activated. Then the target states themselves are activated based on the fork-outgoing transition priority. Next the initial states in the neighboring regions are activated based on the region priority definition.

Again assume that each state has entry and exit actions, and that *Region_0* is always given the highest priority and *Region_x* the lowest. The resulting states activation sequence (and respective execution of entry actions) for the fork construct would be: *b, d* and *e* (based on the fork-outgoing transitions priority), *c* and *f* (based on their region priority). The resulting deactivation for the join construct would be: *d* and *e* (based on the join-incoming transitions priority), *c* and *f* (based on their region priority definition), *b*. In any case, the modeler can define the execution order explicitly.

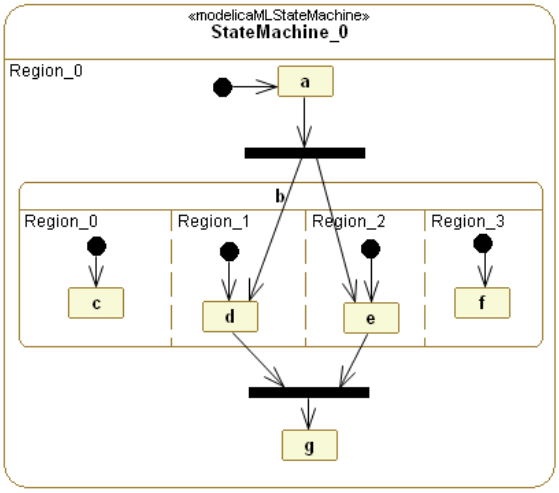

Figure 14: Fork and join example

5 Conclusion

This paper presents a proof of concept for the translation of UML state machines into executable Modelica code. It presents how the executable semantics of UML state machines are defined in ModelicaML and how state machines are used to model parts of class behavior in ModelicaML. The ModelicaML prototypes can be downloaded from [15].

Furthermore, this paper highlights issues that should be addressed in the UML specification and makes proposals on how to resolve them. Section 4.2 questions the use of an event queue that prevents simultaneous events from being processed in parallel. When procedural code is used for the implementation of state machines execution, sec-

tion 4.3 makes a proposal to include priority for regions. A regions priority also supports the definition of states activation or deactivation order in case of inter-level state transitions (section 4.5) or fork/join constructs (section 4.6). Section 4.4.1 introduces execution priority for conflicting state-outgoing transitions in order to allow the modeler to control the execution and to ensure that the state machine behaves as intended.

References

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

[2] OMG. OMG Unified Modeling Language ™ (OMG UML). Superstructure Version 2.2, February 2009.

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

[4] OMG. OMG Systems Modeling Language (OMG SysML™), Version 1.1, November 2008.

[5] Martin Otter, Martin Malmheden, Hilding Elmqvist, Sven Erik Mattsson, Charlotta Johnsson. A New Formalism for Modeling of Reactive and Hybrid Systems. Proceedings of the 7th International Modelica Conference, Como, Italy. September 20-22, 2009.

[6] Ferreira J. A. and Estima de Oliveira J. P., Modelling Hybrid Systems Using Statecharts And Modelica. Department of Mechanical Engineering, University of Aveiro, 3810 Aveiro (PORTUGAL), Department of Electronic Engineering, University of Aveiro, 3810 Aveiro (PORTUGAL)

[7] Adrian Pop, David Akhvlediani, Peter Fritzson. Towards Unified Systems Modeling with the ModelicaML UML Profile. International Workshop on Equation-Based Object-Oriented Languages and Tools. Berlin, Germany, Linköping University Electronic Press, www.ep.liu.se, 2007

[8] Thomas Johnson, Christian Paredis, Roger Burkhart. Integrating Models and Simulations of Continuous Dynamics into SysML. www.omgsysml.org

[9] Johnson, T. A. Integrating Models and Simulations of Continuous Dynamic System Behavior into SysML. M.S. Thesis, G.W. Woodruff School of Mechanical Engineering, Georgia Institute of Technology. Atlanta, GA. 2008

[10] M. von der Beeck. A Comparison of Statecharts Variants. In Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS 863, pages 128-148. Springer, 1994.

[11] Michelle L. Crane and Juergen Dingel. UML vs. Classical vs. Rhapsody Statecharts: Not All Models are Created Equal School of Computing, Queen's University Kingston, Ontario, Canada

[12] Object Management Group (OMG). www.omg.org

[13] Wladimir Schamai, Peter Fritzson, Chris Paredis, Adrian Pop. Towards Unified System Modeling and Simulation with ModelicaML: Modeling of Executable Behavior Using Graphical Notations. Proceedings of the 7th International Modelica Conference, Como, Italy. September 20-22, 2009

[14] IBM® Rational® Rhapsody® Designer for Systems Engineers, http://www-01.ibm.com/software/rational/products/rhapsody/designer/

[15] ModelicaML - A UML Profile for Modelica. www.openmodelica.org/index.php/developer/tools/134

Modal Models in Ptolemy *

Edward A. Lee and Stavros Tripakis
University of California, Berkeley
eal@eecs.berkeley.edu, stavros@eecs.berkeley.edu

Abstract

Ptolemy is an open-source and extensible modeling and simulation framework. It offers heterogeneous modeling capabilities by allowing different models of computation to be composed hierarchically in an arbitrary fashion. This paper describes modal models, which allow to hierarchically compose finite-state machines with other models of computation, both untimed and timed. The semantics of modal models in Ptolemy are defined in a modular manner.

Keywords Hierarchy, State machines, Modes, Heterogeneity, Modularity, Modeling, Semantics, Simulation, Cyber-physical systems.

1. Introduction

Cyber-physical systems (CPS) consist of digital computers interacting among themselves and with physical processes. CPS applications are emerging at high rates today in many domains, including energy, environment, healthcare, transportation, etc.

Designing CPS is a non-trivial task, as these systems manifest non-trivial dynamics, complex interactions, dynamic behavior, and a large number of components. The design complexity is increased by the inherent *heterogeneity* in modeling such systems: parts of the system are digital, others are analog; parts are timed, others untimed; parts are discrete-time, others are continuous-time; parts are synchronous, others are asynchronous; and so on. This inherent heterogeneity implies a need for *heterogeneous modeling*. By the latter we mean a method and associated tools, that provide designers with a way of combining different *models of computation*, in an unambiguous way, in a single model of a system. A model of computation (MoC) here refers to a language or class of languages with a common syntax and semantics. Different MoCs realize different modeling paradigms, each being more or less suitable for capturing different parts of the system.[1]

A number of modeling languages exist today, realizing different MoCs. Many of these languages are gaining acceptance in the industry, in so-called *model-based design* methodologies. Examples are UML/SysML, Matlab/Simulink/Stateflow, AADL, Modelica, LabVIEW, and others. These types of languages are raising the level of abstraction in designing CPS, by offering mechanisms to capture concurrency, interaction, and time behavior, all of which are essential concepts in CPS. Moreover, verification and code generation tools exist for many of these languages, allowing to go beyond simple modeling and simulation, and facilitating the process of going from high-level models to low-level implementations.

Despite these advances, however, no universally accepted solution exists for heterogeneous modeling. In fact, integration of modeling languages and tools is still a common theme in many research or industrial projects, as well as products (e.g., co-simulation environments) despite the fact that such solutions are often cumbersome to use and unsatisfactory, at best.

One of the longest efforts attacking the heterogeneous modeling problem is the Ptolemy project [15, 25]. Ptolemy follows the *actor-oriented* paradigm, where a system consists of a set of *actors*, which can be seen as processes executing concurrently and communicating using some mechanism. In Ptolemy, the exact manner in which actors execute (e.g., by interleaving, in lock-step, or in some other or-

* Ptolemy in this document refers to Ptolemy II, see http://ptolemy.eecs.berkeley.edu. **NOTE:** If you are reading this document on screen (vs. on paper) and you have a network connection, then you can click on the figures showing Ptolemy models to execute and experiment with those models on line. There is no need to pre-install Ptolemy or any other software. The models that are provided online are summarized at http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII8.0/jnlp-books/doc/books/design/modal/index.htm.

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #CCR-0225610 (ITR), #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312 and AF-TRUST #FA9550-06-1-0244), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC) and the following companies: Bosch, National Instruments, Thales, and Toyota.

3rd International Workshop on Equation-Based Object-Oriented Languages and Tools. October, 2010, Oslo, Norway.

EOOLT 2010 website: http://www.eoolt.org/2010/

[1] We should emphasize the importance of syntax, in addition to semantics, in choosing a MoC. State machines, for example, can be given a discrete-time semantics, and so can a synchronous language such as Lustre [19]. Even though the two have the same semantics, their syntax (in the broad sense) is very different, which makes them suitable for different classes of applications.

der) and the exact manner in which they communicate (e.g., through message passing or shared variables) are not fixed once and for all: they are defined by an MoC. The implementation of an MoC in Ptolemy is called a *domain* and is realized by a *director*, which semantically is a composition operator. Currently, Ptolemy includes a number of different domains and corresponding directors, including, *finite-state machines* (FSM), *synchronous data flow* (SDF) [26], *synchronous reactive* (SR) [11, 19, 14], and *discrete event* (DE) [7, 33, 9, 37, 23, 30, 8].

Among the important characteristics of Ptolemy is that (a) it is open-source (and free); and (b) it is architected to be easily extensible. Thanks to these features, new domains and new actors are being added to the tool by different groups, depending on their specific interests.

Another essential feature of Ptolemy is that domains can be combined hierarchically, in an arbitrary fashion. For example, the model in Figure 1 combines SDF with hierarchical FSMs; the model in Figure 3 combines DE with FSMs. This is the fundamental mechanism that Ptolemy provides to deal with the heterogeneous modeling problem. It allows designers to build models where different parts are described in different MoCs, in a well-structured manner [15].

In this paper, we are particularly interested in one aspect of heterogeneous modeling in Ptolemy, namely, *modal models*. Modal models are hierarchical models where the top level model consists of an FSM, the states of which are *refined* into other models (possibly from different domains). Modal models are suitable for a number of applications. They are especially useful in describing event-driven and modal behavior, where the system's operation changes dynamically by switching among a finite set of modes. Such changes may be triggered by user inputs, sensor data, hardware failures, or other types of events, and are essential in fault management, adaptivity, and reconfigurability (see, for instance, [36, 35]). A modal model is an explicit representation of this type of behaviors and the rules that govern transitions between behaviors.

The main contribution of this paper is to provide a formal semantics of Ptolemy modal models. In the process, we also give a modular and formal framework for Ptolemy in general, which is an additional contribution. We do not formalize all the domains of Ptolemy, however, as this is beyond the scope of this work.

The paper is organized as follows. Section 2 briefly reviews the visual syntax of Ptolemy through an example. In Section 3 we provide a formalization of the abstract semantics of Ptolemy. In Section 4 we provide the formal semantics of Ptolemy modal models. In Section 5 we discuss possible alternatives and justify our choices. Section 6 discusses related work. Section 7 concludes the paper.

2. Visual Syntax

Ptolemy models are hierarchical. They can be built using a *visual syntax*, an example of which is given in Figure 1. This example contains, at the top level of the hierarchy, a model with five actors, `Temperature Model`, `Bernoulli`, `ModalModel`, `SequencePlotter` and `SequencePlotter2`. The SDF domain is used at this level, as indicated by the use of `SDF Director`, explained below. `Temperature Model` and `ModalModel` are *composite actors*: they are *refined* into other models, at a lower level of the hierarchy.

The refinement of `ModalModel` is an FSM with two *locations*,[2] `normal` and `faulty`. This FSM is hierarchical: each of its locations is refined into a new FSM, as shown in the figure. In Ptolemy, FSMs use implicitly the FSM domain. This is why no director is shown in the FSM models. The FSM director is implied. Note that, although in this example the location refinements are FSMs, this need not be the case: they can be models using any of the Ptolemy domains (e.g., see example of Figure 3).

The refinement of `Temperature Model` is shown in Figure 2. This refinement does not specify a domain (it contains no director). In such a case, the refinement uses implicitly the same domain as its parent, that is, in this case, the SDF domain. Since this model mixes SDF and FSM, it is an example of a heterogeneous model.

The visual syntax of Ptolemy contains other elements, which we briefly describe next. For details, the reader is referred to [27, 24] and the Ptolemy documentation [1]. Each actor contains a set of *ports*, used for communication with other actors. Ports are explicitly shown in the internal model of a composite actor: for instance, `fault` is an input port and `heat` is an output port of `ModalModel`. A port may be an input, an output, both, or neither. *Parameters* can also be defined: for instance, `heatingRate` is a parameter of the top-level model of Figure 1, set initially to 0.1 (the value of parameters can be modified dynamically during execution).

FSMs in Ptolemy consist of a finite set of locations, one of which is the initial location, and some of which may be labeled as final locations. Initial locations are indicated by a bold outline; the initial locations of the FSMs in Figure 1 are `normal` and `heating`. A transition links a source location to a destination location. A transition is annotated with a *guard*, a number of *output actions* and a number of *set actions*. Guards are expressions written in the Ptolemy expression language. Actions are written in the Ptolemy action language. Guards of two or more outgoing transitions of the same location need not be disjoint, in which case the FSM is non-deterministic. The user can indicate this, in which case transitions are visually rendered in red. *Default* transitions, indicated with dashed lines, are to be taken when no other transitions are enabled, i.e., their guard is the negation of the disjunction of the guards of all other transitions outgoing from the same source location. *Reset* transitions, indicated with open arrowheads, result in the refinement of the destination state being reset to its initial condition. *Preemptive* transitions, indicated by a red circle at the start of the transition, may prevent the execution of the current state refinement, when the guard evaluates to true.

[2] For the visual syntax, we use the term *location* instead of *state*, in order to distinguish it from the semantical concept of state (Section 3).

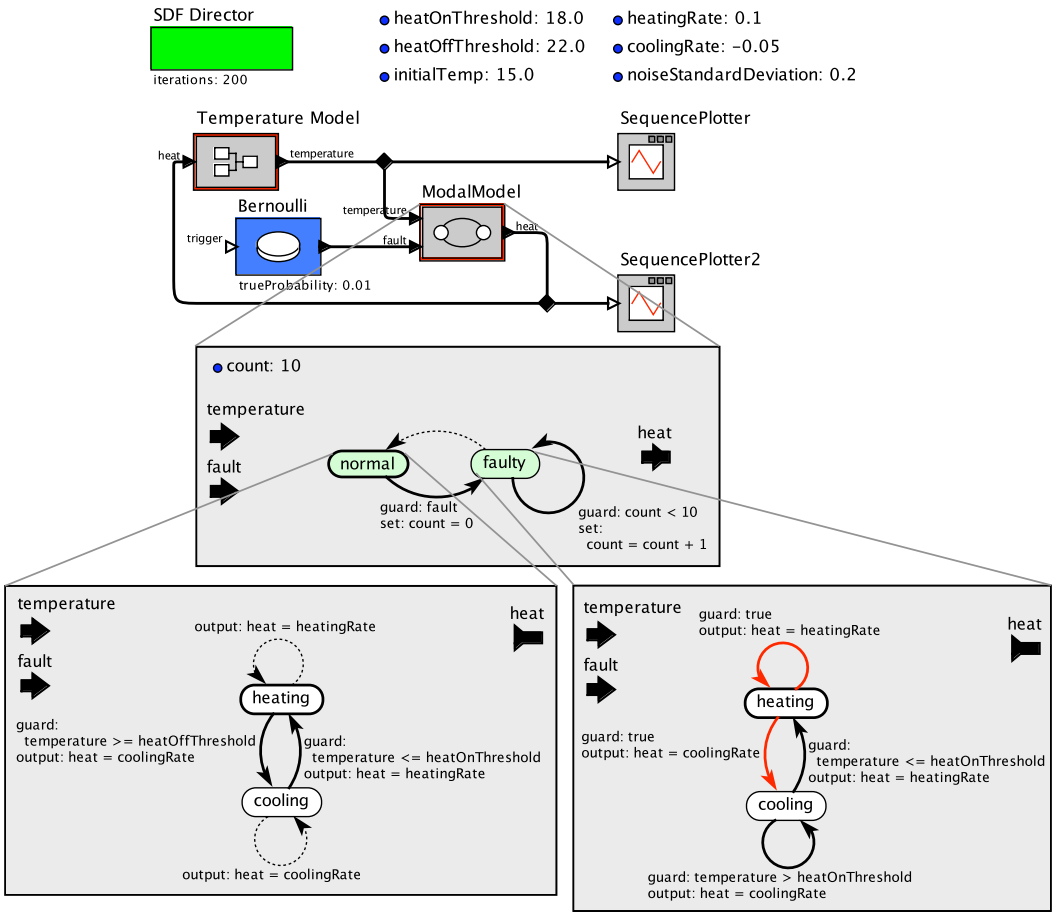

Figure 1. A hierarchical Ptolemy model.

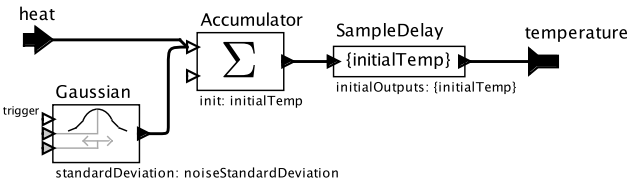

Figure 2. The `Temperature Model` composite actor of Figure 1.

3. Ptolemy Abstract Semantics

The semantics and execution of a Ptolemy model is defined by means of so-called *abstract semantics*. The same mechanism is used to ensure compositionality of Ptolemy domains. Mathematically, a Ptolemy model can be viewed as an abstract state machine, with a set of states, inputs and outputs. The abstract semantics defines the transitions of this machine, that is, how its state and outputs evolve according to the inputs.

Evolution can be seen as being *untimed*, that is, a sequence of transitions, or *timed*, that is, a sequence of transitions annotated by some timing information (e.g., a time delay since the previous transition). It is interesting to note that time is mostly external to the definition of the abstract state machine, that is, the times in which transitions are taken are primarily decided by the environment of the machine. However, *timed*, or *proactive*, machines can also be defined, by providing means to impose constraints on these times. We do this using *timers* (see Section 3.1.2). In the absence of any such constraints, the machine is *untimed*, or *reactive*.

From an implementation point of view, the abstract semantics is essentially a set of methods (in the object-oriented programming sense) that every actor in a model implements. Composite actors also implement these methods, through their director. By implementing these methods in different ways, the various types of Ptolemy directors realize different models of computation. In the Java implementation of Ptolemy the abstract semantic methods form a Java interface that actor and director classes implement. This interface includes the following methods:[3]

- `initialize`: it defines the initial state of the machine.
- `fire`: it computes the outputs at a given point in time, based on the inputs and state at that point in time.
- `postfire`: it updates the state.

A formalization of the abstract semantics of Ptolemy is provided next.

[3] For the purposes of this discussion, we omit some methods, e.g., `prefire`, etc. We also ignore implementation of communication between actors. See [1] for details.

3.1 A formalization of abstract semantics of actors

3.1.1 Untimed actors

An untimed actor is formalized as a tuple

$$(S, s_0, I, O, F, P).$$

The actor has a set of states S, a set of input values I, and a set of output values O. The `initialize` method of the actor is formalized as an initial state $s_0 \in S$. The `fire` and `postfire` methods of the actor are formalized as two functions F and P, respectively, of the following type:

$$F : S \times I \times \mathbb{N} \to O$$
$$P : S \times I \times \mathbb{N} \to S$$

That is, F returns an output value $y \in O$, given a state $s \in S$, an input value $x \in I$, and an *index* $j \in \mathbb{N}$; P returns a new state, given a state, an input value and an index. The index is used to model non-determinism, and can be seen as a "dummy" input.[4]

We require that F and P be total in S and I. F and P may be partial in $\mathbb{N}$ (i.e., in their index argument), however, we require that for any $s \in S$ and $x \in I$, there exists at least one $j \in \mathbb{N}$ such that both $F(s,x,j)$ and $P(s,x,j)$ are defined. If there is a unique such j for all $s \in S$ and $x \in I$ then the actor is *deterministic*. In that case we omit index j and write simply $F(s,x)$ and $P(s,x)$.

The semantics of an untimed actor can be then defined as a set of sequences of *transitions*, of the form:

$$s_0 \xrightarrow{x_0,y_0} s_1 \xrightarrow{x_1,y_1} s_2 \xrightarrow{x_2,y_2} \cdots$$

such that for all $i = 0, 1, ...$, we have $s_i \in S$, $x_i \in I$, $y_i \in O$, and there exists $j \in \mathbb{N}$ such that

$$y_i = F(s_i, x_i, j) \quad (1)$$
$$s_{i+1} = P(s_i, x_i, j) \quad (2)$$

Note that, exactly what the sets S, I and O are, and exactly how the functions F and P are defined, is a property of a given actor: it is in this sense that this semantics is *abstract*. Different actors will have different instantiations of this abstract semantics. Also note that the above elements essentially define a non-deterministic Mealy machine, where F is the *output function* of the machine, and P the state *update function*. This machine is not necessarily finite-state. The input and output domains may also be infinite.

Examples of untimed actors A simple untimed actor is the `Gain` actor which produces at its output a value $k \cdot x$ for every value x appearing at its input. k is a parameter of the actor. This actor is deterministic. It has a single state, and therefore a trivial (constant) update function P. Its F function is defined simply by: $F(x) = k \cdot x$.

[4] Since $j \in \mathbb{N}$, we can model unbounded, but enumerable, non-determinism. The reader may wonder why we do not model the pair F, P simply as a single function with type $S \times I \to 2^{O \times S}$, that is, taking a state and an input and returning a set of state, output pairs. The reason is that we want to decouple output and update functions, which allows to give semantics of modal models in a modular manner: see Section 4.2. Once the F and P functions have been decoupled, it is necessary for some bookkeeping in order to keep track of non-deterministic choices, that must be consistent among the two functions. This role is played by the index $j \in \mathbb{N}$.

3.1.2 Timed actors

The semantics of timed actors extend those of untimed actors with time. In particular, timed actors have special state variables, called *timers*, that measure time. Our timers are *dense-time* variables (they take values in the set of non-negative reals, $\mathbb{R}_{\geq 0}$) inspired by the model of [13]. A difference with [13] is that in our case timers can be created or destroyed dynamically, and the set of timers that are active at any given time is not necessarily bounded. Also, our timers can be *suspended* and *resumed*, which is not the case with the timers of [13]. Timers are *set* to some initial value when they are created, and then run downwards (i.e., decrease as time elapses) until they reach zero, at which point they *expire*. A timer can be *suspended* which means it is "frozen" and ceases to decrease with time. It can then be *resumed*. Suspended timers are also called *inactive*, otherwise they are *active*.

In the case of timed actors, the sets I and O often contain the special value ε denoting *absence* of a signal at the corresponding point in time. We will use this value in the examples that follow.

Consider a timed actor with fire and postfire functions F and P. The semantics of this actor can be defined as a set of sequences of *timed transitions* of the form:

$$s_0 \xrightarrow{x_0,y_0,d_0} s_1 \xrightarrow{x_1,y_1,d_1} s_2 \xrightarrow{x_2,y_2,d_2} \cdots$$

such that there exists a sequence of indices $j_0, j_1, ... \in \mathbb{N}$, and for all $i = 0, 1, ...$, we have $s_i \in S$, $x_i \in I$, $y_i \in O$, $d_i \in T$, and

$$y_i = F(s_i, x_i, j) \quad (3)$$
$$s_{i+1} = P(s_i \ominus d_i, x_i, j_i) \quad (4)$$
$$d_i \leq \min\{c \mid c \text{ an active timer in } s_i\} \quad (5)$$

$d_i \in \mathbb{R}_{\geq 0}$ denotes the time elapsed at state s_i. $s_i \ominus d_i$ denotes the operation which consists in decrementing all active timers in s_i by d_i. Condition (5) ensures that this operation will not result in some timers becoming negative, i.e., that no timer expiration is "missed". This condition therefore "forces" the environment of the actor to fire the actor at least at those instants when its timers are set to expire. Note that the actor could also be fired at other instants as well, for example, whenever an external event is received. The actor itself does not, and cannot, specify those other instants, because they are generally context-dependent.

Notice that, even though the above semantics does not explicitly mention suspensions and resumptions of timers, these actions can be easily modeled as part of the inputs x_i. In Ptolemy, these inputs are not accessible to the user, however, only to the director. This is particularly the case for hierarchical modal models, as described in Section 4.2.

Superdense time: It is worth noting that the delays d_i can be zero. This implies in particular that multiple output events can occur at the same real-time instant. It is convenient to model such cases using so-called *superdense* time, i.e., the set $\mathbb{R}_{\geq 0} \times \mathbb{N}$ [32, 28, 31]. Then, an output y can be seen as a signal with a superdense time axis, that is, as

a partial function from $\mathbb{R}_{\geq 0} \times \mathbb{N}$ to a set of values. For instance, in a run of the form

$$s_0 \xrightarrow{x_0,y_0,d_0} s_1 \xrightarrow{x_1,y_1,d_1} s_2 \xrightarrow{x_2,y_2,d_2} \cdots$$

where $d_0 = d_1 = 0$ and $d_2 > 0$, the output signal y can be seen as a function on superdense time, such that $y(0,0) = y_0$, $y(0,1) = y_1$, $y(d_2,0) = y_2$, and so on.

Examples of timed actors

First, let us consider a `DiscreteClock` actor that periodically emits some value v at its output, with period $\pi \in \mathbb{R}_{>0}$. Both v and π are parameters of the actor. The `DiscreteClock` actor has no inputs. It has a single state variable which is a timer c. Initially, $c = 0$ (as an option, the user can also set initially $c = \pi$, in which case the actor will not produce an event until after one period). The F and P functions of `DiscreteClock` are defined below (the actor is deterministic, so we omit reference to index j):

$$\begin{aligned} F(c) &= v \text{ if } c = 0, \text{ and } \varepsilon \text{ if } c > 0 \\ P(c) &= \pi \text{ if } c = 0, \text{ and } c \text{ if } c > 0 \end{aligned}$$

The definition of F states that an output with value v is produced when the timer c reaches zero, otherwise, the output is absent. The definition of P states that the timer is reset to π when it reaches zero and is left unchanged otherwise.

Next, let us consider the `ConstantDelay` actor, which, for every input with value x that it receives at time t, it produces an output with value x at time $t + \Delta$, where $\Delta \in \mathbb{R}_{>0}$, is a parameter of the actor. As state variables, `ConstantDelay` maintains a set of *active* timers C plus, for each $c \in C$, a variable v_c to memorize the value that must be produced when c expires. Initially C is empty. A new timer c is added to C whenever an input is received: at that point, c is set to Δ and v_c is set to x, the value of the input. When a timer c expires it is removed from C and output with value v_c is produced. Formally, a state s of `ConstantDelay` is a set of triples of the form (c, δ_c, v_c), where c is a timer, $\delta_c \in \mathbb{R}_{\geq 0}$ is the current value of c, and v_c is as explained above. The initial state is $s_0 = \emptyset$. The F and P functions of `ConstantDelay` can be defined as follows (again we omit j because of determinism):

$$F(s,x) = \begin{cases} v_c & \text{if } \exists (c,0,v_c) \in s \\ \varepsilon & \text{otherwise} \end{cases}$$

$$P(s,x) = \begin{cases} (s \setminus \{(c,0,v_c)\}) \cup Q & \text{if } \exists (c,0,v_c) \in s \\ s \cup Q & \text{otherwise} \end{cases}$$

$$Q = \begin{cases} \{(c', \Delta, x)\} & \text{if } x \neq \varepsilon \text{ and } \nexists (c', \delta_{c'}, v_{c'}) \in s \\ \emptyset & \text{otherwise} \end{cases}$$

Note that a "fresh" timer c' is created only when the input x is not absent, as defined by Q.

3.1.3 Untimed actors as a special case of timed actors

As expected, an untimed actor can be seen as a special case of a timed actor, with no timers. Because of this, untimed actors can also be given semantics in terms of sequences of timed transitions. In this case, Condition (4) reduces to Condition (2), and Condition (5) is trivially satisfied with the convention that the minimum of an empty set is infinity. This means that the time instants when untimed actors are fired are entirely determined by the context in which these actors are embedded.

3.2 Composite actors

As illustrated in Section 2, Ptolemy allows to build hierarchical models, by encapsulating a set of actors, plus a director, within a composite actor. The latter is itself an actor, thus can be further encapsulated to create new composite actors. Models of arbitrary hierarchy depths can be built this way.

A composite actor C has an abstract semantics just like any actor. How this abstract semantics is instantiated depends on: (a) the instantiation of the abstract semantics of the internal actors of C; and (b) the director that C uses.

Directors can be viewed formally as *composition operators*: they define functions F and P of a composite actor C, given defined such functions for all internal actors of C.

A large number of directors are included in Ptolemy, implementing various models of computation. It is beyond the scope of this document to formalize all these directors. We informally describe two of them, namely, *SR (synchronous reactive)* and *DE (discrete event)*. More information can be found in [23, 14, 29, 30, 8]. In the next section, we formalize the semantics of the *FSM Director*. The latter implements modal models, which is the main topic of this paper.

Synchronous Reactive (SR): Every time a composite actor C with an SR director is fired, the SR director repeatedly fires all actors within C until a *fixpoint* is reached. This fixpoint assigns values to all ports of actors of C. Note that, because of interconnections between actors, some output ports are typically connected to input ports of other actors of C, and therefore obtain equal values in the fixpoint. The fixpoint is defined with respect to a *flat CPO*, namely, the one that has a bottom element $\perp$ representing an "undefined" or "unknown" value, and all other, "true" values, greater than $\perp$ in the CPO order (see [14]). The fixpoint is computed by assigning initially $\perp$ to all outputs, and then iterating in a given order the F functions of all actors of C. Any execution order can be used and is guaranteed to reach the fixpoint, although some execution orders may be more efficient (i.e., may converge faster). When the fixpoint is reached, the fire() method of the SR director (and consequently, of C) returns.[5] The postfire() method P of C is implemented by invoking the P methods of all internal actors of C.

Discrete Event (DE): DE leverages the SR semantics, but extends it with time. (see [23, 8]). As is typical with DE simulators, the DE director maintains an *event queue* that stores events in timestamp order. Initially, the event queue is empty. When actors are initialized, some of them may post initial events to the event queue. Whenever the composite actor is fired, the earliest events are extracted from

[5] The fixpoint may contain $\perp$ values, which means the model contains feedback loops with *causality cycles*. In this case, the Ptolemy implementation returns a Java exception.

the event queue and presented to the actors that receive them. In contrast to standard DE simulators, Ptolemy incorporates the SR semantics for processing simultaneous events. In particular, a fixpoint is computed, starting with the extracted events at the specified ports, and $\perp$ values for all other, unknown, ports. Fire() returns when the fixpoint is found, as in the SR case. Postfire() consists in calling postfire() of internal actors, as in the SR case. During postfire(), actors may post new events to the queue.

4. Modal Model Semantics

A modal model M is a special kind of composite actor. In the visual syntax, M is defined by a finite-state machine M_c whose locations can be *refined* into sub-models, as illustrated in Figure 1. In Ptolemy terminology, M_c is called the *controller* of M. Each of these sub-models is itself a composite actor. Therefore, the internal actors of M are the composite actors that refine the locations of M_c, plus M_c itself. Note that a special case of modal model is an FSM actor: this is a modal model whose controller has no refinements. Another special case of modal model is a hierarchical state machine: this is a modal model whose refinements are FSM actors or are themselves hierarchical state machines.

In this section, we describe the semantics of modal models, starting with the simple case of FSM actors, and extending to the general case of modal models.

4.1 Semantics of FSM actors

FSM actors are untimed actors. For a given FSM actor M, its set of states is the set of all possible *valuations* of the *state variables* of M. The set of state variables includes all parameters of M (which in Ptolemy can be changed dynamically) as well as a state variable to record the current location of M. A valuation is a function assigning a value to every state variable. The initial state assigns to each parameter its default value (specified by the user) and to the location variable the initial location (also specified by the user). M may also have inputs and outputs, defined in Ptolemy's visual syntax by input and output ports that the user specifies, as in Figure 1.

A state s and an input x of M, together with the transitions of M, define a finite set $s_1, s_2, ..., s_k$ of *successor states*, as follows. Let l be the location of M at s and consider an outgoing transition from l. If the guard of this transition is satisfied by s and x we say that the transition is *enabled*. Suppose there are $k \geq 1$ enabled transitions. Enabled transition j defines successor state s_j. In particular, if the destination location of the transition is l_j, then the location at s_j is set to l_j. Moreover, any parameters that are set in the set action of the transition are assigned a corresponding new value at s_j, and the rest of the parameters remain unchanged (i.e., have the same value at s_j as at s). Therefore, this defines function P on s and x, as follows: $P(s,x,j) = s_j$, for $j = 1, ..., k$. If there are no enabled transitions at s and x, then there is a unique successor state, namely s itself, and we define $P(s,x,j) = s$, only for $j = 1$.

The output actions of the enabled transitions define a set $y_1, y_2, ..., y_k$ of output values, therefore, they define function F on s and x, as follows: $F(s,x,j) = y_j$, for $j = 1, ..., k$. The output values are the values that the output action assigns to output ports of the actor. If an output port is not mentioned in the output action, or if no transitions are enabled (and therefore no output actions are executed) then the value of this port is ε, i.e., "absent".

4.2 Semantics of general modal models

A general modal model M consists of its controller M_c, which is an FSM actor with n locations, $l_1, ...l_n$, plus a set of composite actors $M_1, ..., M_n$, where M_i is the refinement of location l_i. Some locations may have no refinement: this is handled as explained below. Without loss of generality, we assume that the *initial location* of M_c is l_1 (a controller has a single initial location). We also denote by S^c the set of locations: $S^c = \{l_1, ..., l_n\}$.

We denote by S^i, s_0^i, F^i, P^i, respectively, the set of states, initial state, fire and postfire functions of M_i. As explained in Section 3.1.3, untimed actors are special cases of timed actors, therefore, without loss of generality, we can assume that all composite actors M_i are timed. Then, denote by C^i the set of timers of M_i.

In addition, without loss of generality, we can assume that every location has a refinement. Indeed, if location l_i has no refinement, then the above elements can be defined trivially: S^i as a singleton set (i.e., containing a single state which is also the initial state), F^i as the identity function from inputs to outputs, and P^i as the constant function, since the state is unique.

In a modal model M, the sets I and O of input and output values are the same for all internal actors of M, namely, $M_c, M_1, ..., M_n$, and the same for M as well.

The set of states S of M is the cartesian product of the sets of states of all internal actors of M, and similarly for the sets of initial states, i.e.:

$$\begin{aligned} S &= S^c \times S^1 \times \cdots \times S^n \\ s_0 &= (l_1, s_0^1, ..., s_0^n) \end{aligned}$$

Although timers are just a special kind of state variables, it is convenient to be able to refer to them specifically. Therefore, we define C to be the set of timers of M, as

$$C = \bigcup_{i=1,...,n} C^i$$

In s_0, all timers except those in C^1 are set to their suspended state. Those in C^1 are set to their active state.

It remains to define functions F and P of M. Consider a state $s \in S$ and an input $x \in I$. Let $s = (s_c, s_1, ..., s_n)$ be the vector of component states of $M_c, M_1, ..., M_n$, respectively. Suppose the location of M_c at s_c is l_i. Let $J \subseteq \mathbb{N}$ be the set of indices j for which $F^i(s_i, x, j)$ and $P^i(s_i, x, j)$ are defined. We distinguish cases:

1. There are no outgoing transitions of M_c from location l_i that are enabled at s and x. Then, for $j \in J$, we define $F(s,x,j) = F^i(s_i,x,j)$, $P(s,x,j) = (s_c, s_1', ..., s_n')$, where:

(a) $s'_i = P^i(s_i, x, j)$;

(b) for all $m = 1, \ldots, n$ with $m \neq i$, we have $s'_m = s_m$.

2. There exist $k \geq 1$ preemptive outgoing transitions from l_i that are enabled at s and x. Suppose, without loss of generality, that the j-th such transition goes from location l_i to location l_j, for $j = 1, \ldots, k$, and denote its output action and set action by α_j and β_j, respectively. Then, for $j = 1, \ldots, k$, we define $F(s, x, j) = y_j$ and $P(s, x, j) = (s'_c, s'_1, \ldots, s'_n)$, where:

   (a) y_j is obtained from α_j, as in the FSM actor semantics;

   (b) s'_c is obtained from l_j and β_j as in the FSM actor semantics;

   (c) if the j-th transition is not a reset transition then s'_j is identical to s_j, except that all suspended timers in C^j are resumed; if the j-th transition is a reset transition then s'_j is the initial state of M_j: $s'_j = s^j_0$; (note that the timers of M_j, if any, are also re-initialized in the case of a reset transition);

   (d) s'_i is identical to s_i, except that all timers in C^i are suspended;

   (e) for all $m = 1, \ldots, n$ with $m \neq j$ and $m \neq i$, we have $s'_m = s_m$.

3. There are no preemptive outgoing transitions from l_i that are enabled at s and x, but there exist $k \geq 1$ non-preemptive outgoing transitions from l_i that are enabled at s and x. Let $j_1 = 1, \ldots, k$ and suppose that the j_1-th such transition goes from l_i to l_{j_1} and has output and set actions α_{j_1} and β_{j_1}. Let j_2 range in J. Then, for $j = j_1 \cdot j_2$, we define $F(s, x, j) = y_j$ and $P(s, x, j) = (s'_c, s'_1, \ldots, s'_n)$, where:

   (a) y_j is obtained by applying the output action α_{j_1} to $F^i(s_i, x, j_2)$, that is, to the output produced by M_i for non-determinism index j_2;

   (b) s'_c is obtained as in Case 2b.

   (c) s'_j is obtained as in Case 2c.

   (d) s'_i is obtained by applying the set action β_{j_1} to $P^i(s_i, x, j_2)$ and suspending all timers in C^i;

   (e) for all $m = 1, \ldots, n$ with $m \neq j$ and $m \neq i$, we have $s'_m = s_m$.

Item 1 treats the case where no transition of the controller is enabled: in this case, the modal model M behaves (i.e., fires and postfires) like its current refinement M_i. Item 2 treats the case where preemptive transitions of the controller are enabled, possibly in addition to non-preemptive transitions. In this case the preemptive transitions preempt the firing and postfiring of M_i, and only the outputs produced by the transition of the controller can be emitted. Item 3 treats the case where only non-preemptive transitions of the controller are enabled. In this case, before choosing and taking such a transition non-deterministically, we must fire (again, non-deterministically in general) the current refinement M_i.

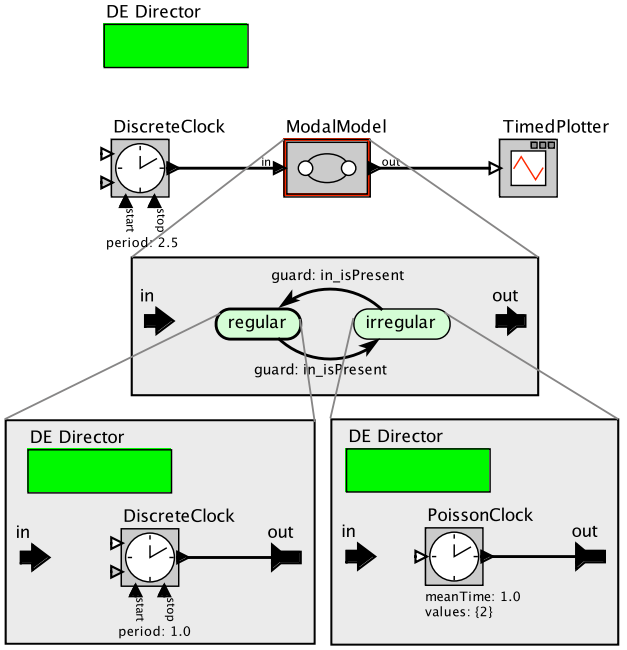

Figure 3. A Ptolemy model with a timed modal model.

Examples As a first example, consider the `ModalModel` actor of Figure 1. The controller of `ModalModel` is the automaton with locations labeled `normal` and `faulty`. The refinements of both these locations are FSM actors. As all refinements are untimed, `ModalModel` is also untimed. The refinement of `faulty` is a non-deterministic FSM actor, as the outgoing transitions of its `heating` location have both guard `true`. The state variables of `ModalModel` are the location variables of all FSM actors, plus the `count` parameter (the other parameters, such as `heatingRate`, etc., should in principle also be included in the state; however, they can be omitted since they remain invariant). A sample of the values that the F and P functions of `ModalModel` take is given below (because of determinism, the index parameter j is omitted):

$$\begin{aligned} F\big((normal, heating, cooling, 10), (22, \overline{fault})\big) &= -0.05 \\ P\big((normal, heating, cooling, 10), (22, \overline{fault})\big) &= \\ (normal, cooling, cooling, 10) & \\ F\big((normal, heating, cooling, 10), (22, fault)\big) &= -0.05 \\ P\big((normal, heating, cooling, 10), (22, fault)\big) &= \\ (faulty, cooling, heating, 0) & \end{aligned}$$

The first two equations correspond to Case 1 whereas the last two equations correspond to Case 3. No preemptive or reset transitions exist in this model.

Another example, that illustrates timed modal models, is shown in Figure 3. This model switches between two modes every 2.5 time units. In the `regular` mode it generates a regularly-spaced clock signal with period 1.0 (and with value 1, the default output value for `DiscreteClock`). In the `irregular` mode, it generates pseudo-randomly spaced events using a `PoissonClock` actor with a mean time between events set to 1.0 and value set to 2. The result of a typical run is plotted in Figure 4, with a shaded

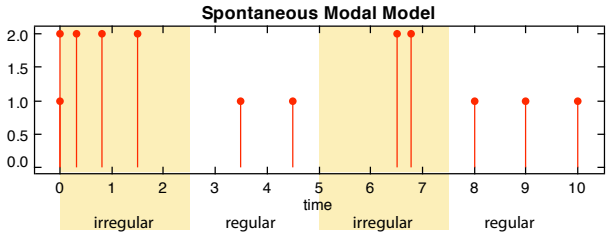

Figure 4. A plot of the output from one run of the model in Figure 3.

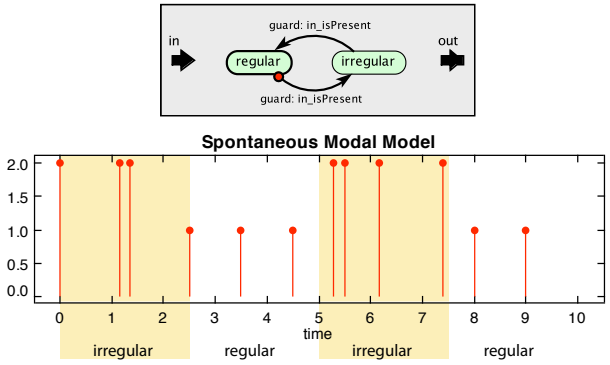

Figure 5. A variant of Figure 3 where a preemptive transition prevents the initial firing of the innermost `DiscreteClock` actor of that model.

background showing the times over which it is in the two modes. A number of observations worth making arise from this plot.

First, note that two events are generated at time 0, a first event with value 1, at superdense time $(0,0)$, and a second event with value 2, at superdense time $(0,1)$. The first event is produced by `DiscreteClock`, according to the semantic rules of Case 3a. If we had instead used a preemptive transition, as shown in Figure 5, then that first output event would not appear: this is according to the semantic rules of Case 2a and the fact that the action of the preemptive transition does not refer to the output port.

The second event is produced by `PoissonClock`, according to the semantic rules of Case 1. The reason for this second event is the following. When the model is initialized, a timer is set by `PoissonClock` to value zero: this means that this timer is to expire immediately, i.e., `PoissonClock` will produce an output immediately when it starts, and at random intervals thereafter.[6] When the `irregular` state is entered, this timer is resumed and since it has value 0, is ready to expire. This forces a new firing of `ModalModel` and ultimately of `PoissonClock`, which produces the event at superdense time $(0,1)$.

Another interesting observation concerns the output events with value 1 occurring at times 3.5, 4.5, 8, and so on. These events occur at times during which the model is at the `regular` mode. Notice that the model begins in the `regular` mode but spends zero time there, since it immediately transitions to the `irregular` mode. Hence, at time 0, the `regular` mode becomes inactive and the timer of `DiscreteClock` is suspended. Since no time has elapsed yet, the timer is equal to 1, the value of the period, at this time. When `regular` is re-entered at time 2.5, this timer is resumed, and expires one time unit later, i.e., at time 3.5. This explains the event at that time. Moreover, the timer is reset to 1 during postfire(), according to Case 1a. It expires again 1 time unit later, which explains the event at time 4.5. Finally, it is reset to 1 at time 4.5, suspended at time 5, and resumed at time 7.5, which explains the event at time 8.

The above examples may appear rather artificial, however, they are given mainly for purposes of illustration of the semantics. More interesting and realistic examples can be found in the open-source distribution of Ptolemy available from `http://ptolemy.eecs.berkeley.edu/`. Detailed descriptions of some of these examples can be found in other publications of the Ptolemy project. For modal models in particular, we refer the reader to the case studies described in [8].

5. Alternative Modal Model Patterns

It is instructive to briefly discuss alternative definitions of modal model semantics and justify our choices.

First, consider our design choice to have the refinement M_i of location l_i in a modal model M "freeze" while the controller automaton is in a location different from l_i. "Freezing" here means that M_i is inactive, in terms of its state which does not evolve at all. This includes in particular the timers of M_i, which are suspended until l_i is re-entered. An alternative would be to consider all refinements "live", but to feed the inputs of M only to the currently "active" refinement, say M_i, and to use the outputs of M_i as outputs of M. Let us term this alternative as the "non-freezing" semantics, for the purposes of this discussion.

One issue with the non-freezing semantics is that it is redundant from a modeling point of view. Indeed, as we show next, there exists a simple design pattern that allows the non-freezing semantics to be easily implemented in Ptolemy. Since this mechanism already exists, there would be no need to add modal models to get the same semantics. In fact, using different modeling patterns that result in the same semantics may be confusing.

This design pattern, which we call the *switch-select pattern*, is illustrated in Figure 6. There are five actors in this model, `M1`, `M2`, `Controller`, `BooleanSwitch` and `BooleanSelect`. `M1` and `M2` represent the refinements of the non-freezing modal model that the pattern captures, and `Controller` is its controller (which is assumed to have 2 locations in this example). The switch and select actors control the routing of the inputs/outputs to/from either `M1` or `M2`, depending on the state that the `Controller` is in. The latter may in turn generally depend on outputs of these actors, which is captured by the communication links between `Controller`, `M1` and `M2`.

Another issue with the non-freezing semantics is that it is less modular than the freezing semantics. In the freez-

[6] This is the default operation, which can be optionally modified by the user by setting the appropriate parameter of the `PoissonClock` actor.

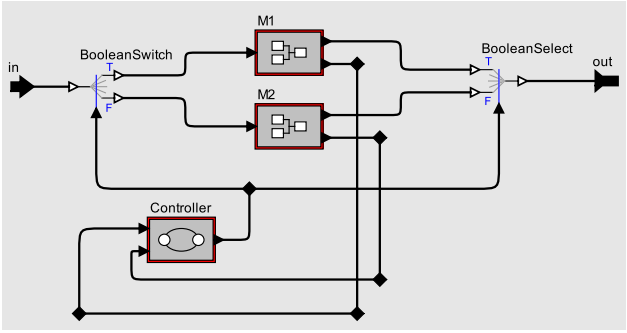

Figure 6. The switch-select pattern.

ing semantics, a subsystem (refinement of a certain location) is *completely unaffected* by being suspended. In the non-freezing semantics, behavior of a subsystem continues while the latter is inactive, only with absent inputs. Thus the evolution of the subsystem depends on how much time it remains inactive, for instance.

Another design choice could be to have time pass in inactive subsystems, i.e., to have their timers active, while having the rest of their state be frozen. The disadvantage of this approach is that for many components (e.g. `DiscreteClock`), the state is intrisically bound to time. It is therefore hard to separate the two notions.

Finally, it is worth mentioning the approach taken in the Simulink/Stateflow tool from the Mathworks. Simulink is a hierarchical block diagram language. Some Simulink blocks can be Stateflow models, that is, hierarchical state machines similar to Statecharts [22]. Simulink blocks, however, cannot be embedded into Stateflow as state refinements. The way to get modal behavior in Simulink/Stateflow is by connecting Stateflow outputs to *enable* inputs of Simulink blocks. When a block is disabled, it is frozen, as in the Ptolemy semantics. Contrary to Ptolemy, however, the output of a disabled block can still be used as it still exists: it is simply held constant while time passes.

6. Related work

A number of formalisms based on hierarchical state machines (HSMS) have been studied in the literature, including Statecharts [22], SyncCharts [5], and commercial variants such as Stateflow from the Mathworks or Safe State Machines from Esterel Technologies [6] (SSMs are based on SyncCharts). Hierarchical state machines are also one of the diagrams of UML. The main difference of Ptolemy modal models with respect to the above is that in Ptolemy modal model refinements are not restricted to state machines or concurrent state machines (built with AND states). In Ptolemy, refinements can include other domains as well, for instance, as in Figure 3. Note that AND states can still be modeled in Ptolemy, using concurrent `ModalModel` actors. For instance, the `TemperatureModel` and `ModalModel` actors shown in Figure 1 are concurrent: the `TemperatureModel` could very well be another modal model. Note that in this case, a MoC such as SR or DE must be specified, in order to define the semantics of the composition of these actors.

Even when we restrict our attention to pure HSMs with no concurrency, there are differences between the Ptolemy version and the models above. A variety of different semantics has been proposed for Statecharts for instance, see [10, 16]. Operational and denotational semantics for Stateflow are presented in [21, 20]. Implicit formal semantics of Stateflow by translation to Lustre are given in [34].

Also, contrary to Statecharts, SyncCharts and Stateflow, Ptolemy modal models do not use broadcast events for communication.[7] Guards may refer to input events, however, these events are transmitted using explicit ports and connections, and are evaluated when the fire() or postfire() methods are called (e.g., guard `in_isPresent` in Figure 3 is evaluated to true or false depending on whether the value of the input is present or absent when the fire() method is called).

Another difference with the above languages is that Ptolemy modal models include both untimed (*reactive*) and timed (*proactive*) models. Timed versions of Statecharts and UML (but not general modal models) have been proposed in [12, 17].

The semantics we present are somewhat operational in nature, given by functions that produce outputs and update the state. Our semantics is also abstract, as in Abstract State Machines [18]. Most importantly, our semantics is modular, in the sense that we show how the output and state update functions of composite actors are defined given output and state update functions of sub-actors.

Formal studies of HSMs can be found in [3, 4, 2].

7. Conclusions

We presented a modular and formal framework for Ptolemy, and described the semantics of modal models, as these are implemented in Ptolemy. Modal models allow hierarchical composition of state machines with other MoCs, therefore generalizing hierarchical state machines and enriching heterogeneous modeling with modal behavior.

Existing Ptolemy models emphasize actor semantics, by having an explicit notion of inputs and outputs. This is in contrast to languages such as Modelica, which are based on undirected equations. Note that feedback loops are allowed in Ptolemy, and can be used to capture some form of equational constraints. How these loops are handled depends on the domain used. In the SR and DE domains, for instance, the equations are solved by fixpoint computations, as mentioned above. In the future we intend to study equational constraints in more depth, borrowing ideas from languages such as Modelica. One direction would be to implement a Modelica domain in Ptolemy, which would work by translating Modelica models into, essentially, CT models, and

[7] It is worth pointing out that, although it is common to refer to communication in Statecharts as being "broadcast", this is slightly misleading, since it implies that all processes receive all signals, which is not the case. A more accurate description is "name matching" since, in fact, only those processes that refer to the signal by name receive it. Name matching is as static as ports in Ptolemy, but is less modular (changing the name in one part of the model requires changing it at other places as well). It also requires more effort to identify the communication links between processes (e.g., when determining causality loops in a diagram).

then handle the latter using numerical solvers. This translation could benefit from the code generation framework available in Ptolemy [38].

Although our discussion in this paper focused on discrete-time modal models, Ptolemy currently supports continuous-time models as well, via the *CT domain*, which allows a number of numerical solvers to be expressed, including those that use backtracking [28, 29]. A formalization of CT using the framework developed in this paper is a topic of future work.

The semantics developed in this paper are operational. It would be interesting to study also a denotational semantics of modal models. The work reported in [20] could be beneficial in that context.

Acknowledgments

Several people contributed to the FSM and modal-model infrastructure in Ptolemy. The modal domain was created primarily by Thomas Huining Feng, Xiaojun Liu, and Haiyang Zheng. The graphical editor in Vergil for state machines was created by Stephen Neuendorffer and Hideo John Reekie. Joern Janneck contributed to the semantics for timed state machines. Christopher Brooks created the online version of the models accessible by hyperlink from this document, and has also contributed enormously to the Ptolemy software infrastructure. Other major contributors include David Hermann, Jie Liu, and Ye Zhou.

References

[1] Ptolemy II Design Documents. Available at http://ptolemy.eecs.berkeley.edu/ptolemyII/designdoc.htm.

[2] R. Alur and R. Grosu. Modular refinement of hierarchic reactive machines. *ACM Trans. Program. Lang. Syst.*, 26(2):339–369, 2004.

[3] R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. In *26th International Colloquium on Automata, Languages, and Programming*, volume LNCS 1644, pages 169–178. Springer, 1999.

[4] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst.*, 23(3):273–303, 2001.

[5] C. André. SyncCharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, April 1996.

[6] C. André. Semantics of S.S.M (Safe State Machine). Technical report, Esterel Technologies, April 2003.

[7] A. Arnold and M. Nivat. Metric interpretations of infinite trees and semantics of non deterministic recursive programs. *Fundamenta Informaticae*, 11(2):181–205, 1980.

[8] K. Bae, P. Csaba Olveczky, T. H. Feng, E. A. Lee, and S. Tripakis. Verifying Hierarchical Ptolemy II Discrete-Event Models using Real-Time Maude. Technical Report UCB/EECS-2010-50, EECS Department, University of California, Berkeley, May 2010.

[9] C. Baier and M. E. Majster-Cederbaum. Denotational semantics in the CPO and metric approach. *Theoretical Computer Science*, 135(2):171–220, 1994.

[10] M. von der Beeck. A comparison of Statecharts variants. In H. Langmaack, W. P. de Roever, and J. Vytopil, editors, *3rd Intl. Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, pages 128–148, Lübeck, Germany, 1994. Springer-Verlag.

[11] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[12] W. Damm, B. Josko, H. Hungar, and A. Pnueli. A Compositional Real-time Semantics of STATEMATE Designs. In *Compositionality: The Significant Difference*, volume 1536 of *LNCS*, pages 186–238. Springer, 1998.

[13] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212. Springer, 1989.

[14] S. A. Edwards and E. A. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48(1), 2003.

[15] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(2):127–144, 2003.

[16] R. Eshuis. Reconciling statechart semantics. *Sci. Comput. Program.*, 74(3):65–99, 2009.

[17] S. Graf, I. Ober, and I. Ober. A real-time profile for UML. *Soft. Tools Tech. Transfer*, 8(2):113–127, 2006.

[18] Y. Gurevich. Evolving algebras: An attempt to discover semantics. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific, 1993.

[19] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1319, 1991.

[20] G. Hamon. A denotational semantics for stateflow. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 164–172, New York, NY, USA, 2005. ACM.

[21] G. Hamon and J. Rushby. An operational semantics for Stateflow. In *Fundamental Approaches to Software Engineering (FASE)*, volume 2984 of *LNCS*, pages 229–243, Barcelona, Spain, 2004. Springer.

[22] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[23] E. A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7:25–45, 1999.

[24] E. A. Lee. Finite State Machines and Modal Models in Ptolemy II. Technical Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley, Nov 2009.

[25] E. A. Lee. Disciplined heterogeneous modeling. In O. Haugen D.C. Petriu, N. Rouquette, editor, *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering, Languages, and Systems (MODELS)*, pages 273–287. IEEE, October 2010.

[26] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[27] E. A. Lee and S. Neuendorffer. Tutorial: Building Ptolemy II Models Graphically. Technical Report UCB/EECS-2007-129, EECS Department, University of California, Berkeley, Oct 2007.

[28] E. A. Lee and H. Zheng. Operational semantics of hybrid systems. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control (HSCC)*, volume LNCS 3414, pages pp. 25–53, Zurich, Switzerland, 2005. Springer-Verlag.

[29] E. A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, Salzburg, Austria, 2007. ACM.

[30] X. Liu and E. A. Lee. CPO semantics of timed interactive actor networks. *Theoretical Computer Science*, 409(1):110–125, 2008.

[31] X. Liu, E. Matsikoudis, and E. A. Lee. Modeling timed concurrent systems. In *CONCUR 2006 - Concurrency Theory*, volume LNCS 4137, Bonn, Germany, 2006. Springer.

[32] Z. Manna and A. Pnueli. Verifying hybrid systems. *Hybrid Systems*, pages 4–35, 1992.

[33] G. M. Reed and A. W. Roscoe. Metric spaces as models for real-time concurrency. In *3rd Workshop on Mathematical Foundations of Programming Language Semantics*, pages 331–343, London, UK, 1988.

[34] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and Translating a "Safe" Subset of Simulink/Stateflow into Lustre. In *Proceedings of the 4th ACM Intl. Conf. on Embedded Software (EMSOFT'04)*, pages 259–268. ACM, September 2004.

[35] G. Simon, T. Kovácsházy, and G. Péceli. Transient management in reconfigurable systems. In *IWSAS' 2000: Proc. 1st Intl. Workshop on Self-Adaptive Software*, pages 90–98, Secaucus, NJ, USA, 2000. Springer.

[36] J. Sztipanovits, D.M. Wilkes, G. Karsai, C. Biegl, and L.E. Lynd. The multigraph and structural adaptivity. *IEEE Trans. Signal Proc.*, pages 2695–2716, August 1993.

[37] R. K. Yates. Networks of real-time processes. In E. Best, editor, *Proc. of the 4th Int. Conf. on Concurrency Theory (CONCUR)*, volume LNCS 715. Springer-Verlag, 1993.

[38] G. Zhou, M.-K. Leung, and E. A. Lee. A code generation framework for actor-oriented models with partial evaluation. In Y.-H. Lee et al., editor, *Internation Conference on Embedded Software and Systems (ICESS)*, volume LNCS 4523, pages 786–799, Daegu, Korea, 2007. Springer-Verlag.

Profiling of Modelica Real-Time Models

Christian Schulze[1] Michaela Huhn[1] Martin Schüler[2]

[1]Technische Universität Clausthal, Institut für Informatik, Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, Deutschland
{Christian.Schulze | Michaela.Huhn}@tu-clausthal.de
[2]TLK-Thermo GmbH, Hans-Sommer-Str. 5, 38106 Braunschweig, Deutschland
M.Schueler@tlk-thermo.de

Abstract

Modeling and simulation of physical systems have become a substantial part in the development of mechatronic systems. A number of usage scenarios for such models like Rapid Control Prototyping and Hardware-in-the-Loop testing require simulation in real-time. To enable model execution on a hard real-time target, a number of adaptations are usually performed on the model and the solver. However, a profiling facility is needed to direct the developer to performance bottlenecks.

We present the concepts and a prototypical implementation of a profiler for the specific analysis of Modelica models running on Scale-RT, a Linux-based real-time kernel. The profiler measures the number of calls and execution times of simulation specific functions calls. Interpreting these results, the developer can directly deduce which components of a simulation model are most promising for optimization. Profiling results and their impact on model optimization are discussed on two case studies from the area of thermodynamic automotive systems.

Keywords Real-Time, Modelica, Profiling, Optimization, SimulationX, Scale-RT

1. Introduction

The modeling and simulation language Modelica is widely accepted in transport industries, in particular in the automotive area. Modelica is employed for modeling the physics of the controlled system in the software development process of electronic control components. Whereas so far simulation aimed for conceptual validation in the early concept phase, nowadays we find an increasing need for real-time simulation or even real-time execution of models on micro-controllers.

Prominent usages of real-time simulation are Rapid Control Prototyping (RCP) [7] and Hardware-in-the-Loop (HiL). These are techniques for the concept and development phases: The overall system is modeled as a combination of the controlled part and a model of the controller - often in a unified modeling and simulation environment. Combined simulation facilitates validation not only of the concepts, but - in a stepwise refinement process - also of the detailed functional and timing behavior of the controller under design, provided detailed physical models and sufficient computing resources are available. For this purpose, the major requirement is that simulation runs as fast as the real system. Several real-time platforms are available to support RCP or HiL, like the open-source Linux-based Scale-RT [9] running on standard PCs, or specific hardware solutions e.g. dSPACE systems.

Another usage of real-time simulation is to execute the model of the controlled system as part of the control: The idea of Model Predictive Control (MPC) is to predict the short term behavior of the physical system by feeding the sensored data from the system into the model and simulate its reaction on possible inputs from the controller, thereby optimizing the controller strategy. In on-board diagnostics, the results from a model running in parallel on the controller are compared to the measurements of the real system to deduce abnormal behavior that is a sign of failures. For these usages, the simulation model has to be executed on the same target as the control, i.e. a micro-controller with restricted resources in many cases. Consequently, being part of the control component imposes hard real-time constraints on model execution.

The usages we mentioned are in the context of *hard* real-time systems (HRT), i.e. systems for which the timely response has to be verified for *all* possible executions of a system component. In contrast to hard real-time, a *soft* real-time system is only required to perform its tasks according to a desired time schedule on the average [3]. As a consequence of the stringent needs for verification, the component behavior of HRT systems has to be analyzed in detail with respect to the timing constraints.

In order to guarantee predictable execution times, simulations on real-time targets typically use a fixed-step solver to solve the DAE-System (Differential Algebraic Equation). Moreover, such models require highly efficient modeling to not exceed the given step size. But even then source code that is automatically generated from Modelica models may violate the timing constraints.

3rd International Workshop on Equation-Based Object-Oriented Languages and Tools. October, 2010, Oslo, Norway.

During the solving process the solver generates events for every zero crossing of a zero function. The solver examines the DAE-system in an interval close about these events, so events cause additional work load and will increase the model runtime.

So, timing problems of the simulation model may arise from various causes like events or the internal complexity of the model. They become evident when runtime exceeds the solver step size and hence the HRT-system usually will abort the execution (although there are ways to construct solvers that are able to ignore such overruns (see [14]).

In case of a timing violation the developer of the simulation model has to improve the model's efficiency by reducing the model's complexity, by setting better start values, or other measures. But so far these steps are purely based on the developer's experience. Real-time profiling may help to give him or her a better understanding of the underlying DAE-System since Kernighan and Pike note "Measurement is a crucial component of performance improvement since reasoning and intuition are fallible guides and must be supplemented with tools like timing commands and profilers." [8]. Based on profiling results the developer is able to identify the components causing the main work load and decide whether a submodel has to be enhanced or an algebraic loop has to be broken.

There are several real-time profiling tools available in particular for Linux-based systems [12, 2], but these are general purpose profiles and not specifically well suited to analyze code for real-time targets that was automatically generated from tools like Dymola or SimulationX. First of all, the actual profiling shall be performed on the real-time target itself to get direct information about the runtime on a specific host. Profiling an execution on a standard PC under Windows and scaling the results for a specific target as it can be done in other domains will not give valid approximations here, because most approaches and tools employ another (fixed-step) solver for execution on real-time targets whereas variable step solvers are used under Windows which differ significantly with respect to their timing characteristics. In addition, the real-time target may impose further restrictions on the profiling, e.g. the real-time Linux Scale-RT 4.1.2 compiles and runs the model as a Kernel-Module.

The described usage scenario in the development of simulation models requires a precise measurement of function execution times and function call counting as well as measurement of certain code sections representing algebraic loops lead to the development of a new profiling tool that can be used on such a real-time operating system.

Within the source code of a model calls to external libraries may occur, e.g. fluid property libraries. For modeling of thermodynamic systems most of the work load is generated by those function calls. Therefore it is necessary to examine them closer.

In general, an algebraic loop results from connecting the output of a submodel to the input of that particular model. Due to this cyclic dependency relation, models containing algebraic loops have to be solved iteratively. Algebraic loops cause serious problems in simulation tools based on a simple input-output-block structure like Matlab Simulink. In equation based modeling the loops are traced back to the underlying equations and may be solved analytically but still many of them have to be solved numerically[13].

The profiling method introduced in this paper will measure the execution time of each function call, to count function calls separately in each relevant section and to measure the time needed to solve algebraic loops, so called "(non-)linear blocks". Especially for profiling of real-time models the overhead of the profiling method on the model runtime shall be kept small. This is achieved by implementing the producer-consumer-pattern as described in Section 4.

Basically this concept can be applied to each target operating system and simulation environment (e.g. Dymola). Even an online evaluation of the profiling results during execution of the model could be implemented. Until now we implemented this concept for models exported from SimulationX to Scale-RT. The instrumentation for the case studies has been done manually, but automatisation will be finalized soon.

2. Profiling on Real-Time Targets

Tracing and profiling are two related techniques that aid the developer to understand the behavior of a program. Tracing gives a detailed view on which function is called, who is the callee, how long does the execution take and also a call counting may take place. Profiling instead gives a statistical evaluation of average execution times and frequencies of the function calls or the profiled sections [12].

The tracing results on a particular program execution can be displayed as call-chains in a call-graph. A call-graph demonstrates the possibly complex call structures annotated with the execution times of the callee. Call-graphs are especially helpful to understand the communication of threads within multi-threaded applications. A Profiling tool generates simpler, statistical results without any structure or evaluation of the call context. However, as the execution of a simulation model follows a fixed elementary plan as described in Section 3, profiling is sufficient for our purposes. Profiling and tracing generally involve three phases:

- instrumentation or modification of the application to perform the measurement
- actual measurement during execution of the application
- analysis of results

Instrumentation adds instructions to the application for measuring execution times, updating counter variables and measuring the consumption of resources. Instrumentation cannot only be performed at source code level but also during compilation, linking or even at the target code level. Where the instrumentation takes place depends on the profiling tool. However, in any case the measured data need to be traced back to the source code level for interpretation.

The instrumentation will obviously increase the execution time of the application, because additional steps will be taken to measure and store the profiling or tracing data.

In general, the overhead caused by instrumentation shall be reduced to a minimum.

The two main approaches to profiling are based on sampled process timing and measured process timing. In profiling based on sampling, a hardware interval timer periodically interrupts the execution of the profiled application. During interruption the profiling tool examines which parts of the program have been executed since last interruption. Profiling tools like prof [5] and gprof [4] are based on sampling and commonly employed.

In profiling based on measured process timing the instrumentation procedures are called at function entry and exit. When entering or leaving a function a time stamp is recorded additionally to counters and timers. Profiling tools like TAU [11] are based on this approach and we will follow it, too.

3. Profiling of Modelica Models

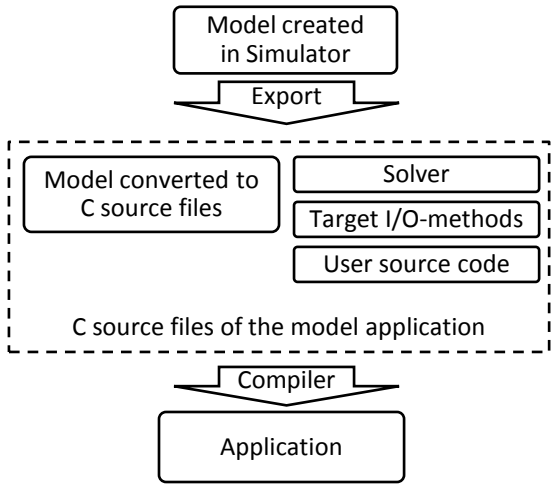

Figure 1. Export of models from a simulator to target

Simulation tools like SimulationX, Dymola or Matlab/Simulink Realtime Workshop share the basic process of exporting models which is depicted in Figure 1. The simulator transforms the model to a DAE-System and after that into C code written in a file. Several other sources are added to the transformed model including a mathematical solver and target specific I/O-methods, but those additional files are always the same. Furthermore, user code can be integrated manually at this point introducing user methods or user libraries. This set of files builds the source code for an application that calculates the desired results.

The C code originating from the models that were transformed by the simulator have a common simple structure. In the moment, this is specific to the framework used for simulation, but with the new Functional Mock-up Interface (FMI) [1] this is standardized. The FMI defines the external interface between the C-Code or even the target code generated from models and the solvers. By using the FMI interface, models can be connected to any solver that realizes the solver's side of the interface; and vice versa a solver may be attached to any model that communicates via FMI. Thereby models can be executed in "foreign" simulation frameworks. When connecting the model code to a solver which is part of the native simulator another internal and more efficient interface may be used. However, in our current approach the proprietary format of the model C code provided by Simulation X is taken as input for the profiling.

To execute the model the following steps are taken:

- export of the model
- compilation of the model application
- transfer to the real-time target
- execution

The work flow for Scale-RT 4.1.2 as the target system is as follows: SimulationX compiles the source of the model in the Cygwin environment, which has to be installed under Windows. This environment provides all libraries and includes needed by this version of Scale-RT. The resulting file is a tgz-file containing the compiled model and additional settings.

In order to be executed on the Scale-RT, the model application is sent to the target system, e.g. by using the Scale-RT Suite, which is part of the Scale-RT Environment as well as Cygwin. Subsequently the model application can be executed from the Scale-RT Suite. SimulationX is able to send the model application to the target system and execute it, too, but the results cannot be observed from there.

In general, the model is separated in an initialization and the simulation problem. Both of them consist of a number of integration steps as well as a set of explicit calculations of the outputs. A global fixed-step solver is used on real-time targets for solving. In case of an overrun the execution stops; so it is considered as a hard real-time simulation guaranteeing the delivery of results within a certain time.

Each step of the global solver consists of one method called several times representing the integration step and one method called only once outputting the simulation results through defined I/O-methods. Within those two methods every calculation including function calls to user libraries occur. Within the integration steps (non-)linear equations (algebraic loops) that could not be solved analytically are evaluated numerically using a local solver. So the structure of the source code for both, the initialization and the simulation problem, looks as follows:

- Global solver step
  - $n_I \cdot$ integration steps
    - $e_I \cdot$ external function calls
    - $c_I \cdot$ additional calculations
    - $a_I \cdot$ (non-)linear blocks
      - $e_{aI} \cdot$ external function calls
      - $c_{aI} \cdot$ additional calculations
  - $1 \cdot$ output of variables
    - $e_O \cdot$ external function calls
    - $a_O \cdot$ additional calculations

Because of the flat structure of the automatically generated source code the call-chain is not needed to understand and analyze performance bottlenecks in a model in many cases. We consider that a flat profiling for each relevant section as the best choice. A flat profiling should be performed on each (non-)linear block within the integration step as well as the integration step and the output of variables itself. This gives the clearest view on the work load caused by every single section.

Since hard real-time simulation shall guarantee the delivery of the results within the time limits, the maximum runtime of a model is more important than the average runtime. The profiling methods described in this paper measure and save the execution times of each simulation step separately. Then the average and variance as well as the maximum of the execution times are determined.

The aim of profiling is to direct the developer towards parts of the model that are worth optimizing. But the developer is in charge of optimizing the model manually. However, after optimization it has not only to be verified that the model application is running faster as before indeed, but also that the optimized model calculates its results with sufficient accuracy. So the work flow of real-time optimization is as follows:

1. check model runtime, if real-time constraints are satisfied finish optimization
2. perform profiling
3. analyze the profiling data and identify performance bottlenecks worth optimizing
4. optimize the model
5. check correctness of modification, if deviation errors are too big revert and go back to 3.
6. go back to 1.

4. Implementation

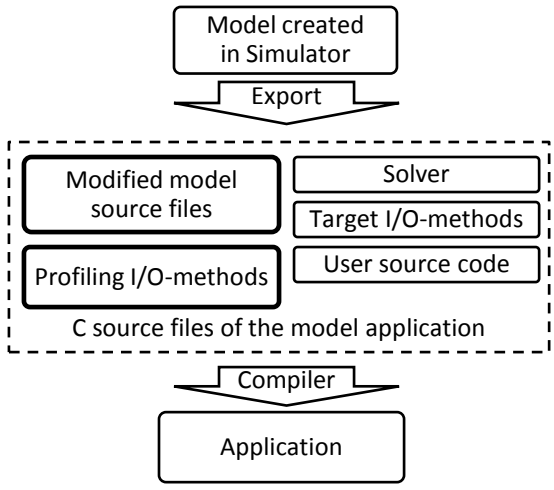

Figure 2. Modified export of models from a simulator to target

Comparing figure 1 and figure 2 reveals the modifications necessary for profiling. The converted model C source file and the file containing the main real-time routines have been modified to perform the profiling, access a FIFO buffer and to provide buffer memory to store the profiling data internally.

As explained, the contribution of the profiling on execution times shall be as small as possible. In our context where the model will be executed as a kernel model, an efficient solution for outputing profiling data is a major point for minimizing the overhead. Therefore we applied the consumer-producer pattern and divided the profiling into two tasks: The real-time Kernel task executing the model and a User Space task - outside the hard real-time context of the kernel - evaluating and storing the measured data. The two tasks are communicating through first-in-first-out (FIFO) buffer.

As displayed in figure 3 the source code for the model application is compiled as real-time task kernel module in Scale-RT. The goal of instrumentation of the kernel task is to log each external function call's execution time in detail. The instrumentation can easily be automated. For different analysis szenarios the instrumentation is configured to profile only the functions calls and sections of interest. For the case studies described in Section 5 insturmentation has to be done manually, but automatisation will be finished soon.

In order to store the profiling data on the hard disk without delaying the execution of the model, a consumer is created reading the FIFO buffer, interpreting the data and storing statistical data, the minima and maxima for each global solver step on the hark disk. As the execution times of different sections as well as the external function calls are measured the overhead of the global solver as well as the profiling overhead can be estimated by comparing it to the performance of the uninstrumented model.

For profiling, the model allocates memory of a fixed size for an intermediate buffer and a main buffer when it begins to execute. The main buffer is used to store the data until the non-real-time user task can process it. This buffer is implemented as a double buffer to avoid buffer overflows. As soon as the first buffer is filled the routine switches to the secondary buffer and sends the content of the first one to the FIFO buffer. The intermediate buffer is used to record all profiling data of one global solver step and is emptied in the main buffer at the end of the current step.

Since in version 4.1.2 of Scale-RT in combination with Cygwin the Kernel-Module cannot export symbols to the user address space, the main buffer cannot be accessed directly by the user task. Writing the data into the virtual file system procfs a.k.a FIFO buffer is a temporary workaround for this problem enabling the Kernel task to store data efficiently. The user task triggers the execution of a Kernel tasks method which copies the main buffer into the FIFO buffer. In future the main buffer will be accessed and read out by the user task directly, therefore these buffers will use shared memory allocation methods.

For each global solver step of the model, the profiling methods record the following information:

- execution time and frequency of an integration step

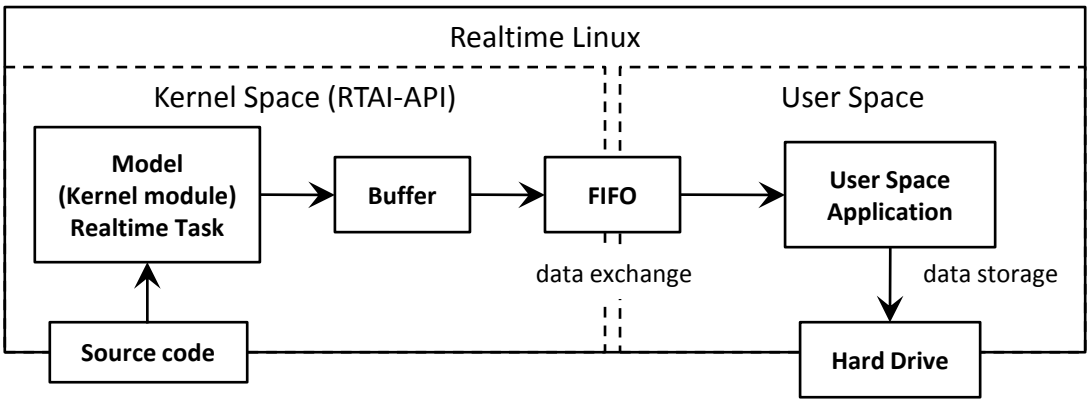

Figure 3. Communication between user task and model task

- execution time and frequency of each external function call within the integration step that does not reside inside a (non-)linear block
- execution time and number of loops of each (non-)linear block within the integration step
- execution time and frequency of each external function call within each (non-)linear block
- execution time of outputting variables at the end of the current step

5. Case Studies

5.1 Case 1: Moist Air inside the Cabin of a Car

5.1.1 Description

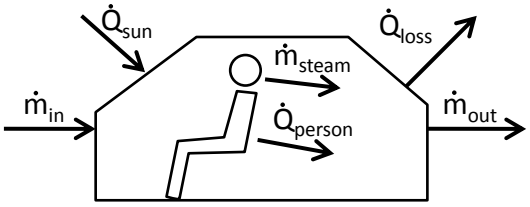

Figure 4. Model of the air inside a cars passenger cabin

This Modelica model describes a simple system of the air within a cabin of a car as displayed in figure 4 and it uses the TEMO-property-library.

There is a mass flow of moist air entering the system coming from the air conditioning and another mass flow of moist air leaving the system. There is an heat flow induced by the sun heating up the air inside the cabin and another heat flow out of the cabin due to heat losses. In addition to that there is a person inside the cabin who heats up the air and also increases the moisture by a given water mass flow.

The thermodynamic properties of the moist air are modelled on the basis of the ideal gas theory [15]. Condensing and simple frost formation can be described with the property equations given below. As the pressure in this case study is about $1bar$ with temperatures down to $0°C$ the error introduced by applying the ideal gas theory is very small.

$$\begin{aligned} \frac{dm}{dt}(h - pv) + & \\ (c_p - R_i) \cdot \frac{dT}{dt} \cdot m &= \dot{m}_{in} \cdot h_{in} \\ &\quad + \dot{m}_{steam} \cdot h_{steam} \\ &\quad - \dot{m_{out}} \cdot h_{out} \\ &\quad + \dot{Q}_{sun} + \dot{Q}_{person} \qquad (1) \\ \frac{dm}{dt} &= \dot{m}_{in} + \dot{m}_{steam} - \dot{m}_{out} \qquad (2) \\ \frac{dm_{steam}}{dt} &= \dot{m}_{in} \cdot \xi_{in} + \dot{m}_{steam} \\ &\quad - \dot{m}_{out} \cdot \xi_{out} \qquad (3) \end{aligned}$$

Equation (1) is the first law of thermodynamics applied to this model. The left side represents the dynamic change of energy inside the cabin, the right side embodies the heat and mass flows into and out of the cabin.

The pure mass balance is described in equation (2), the balance for the water inside the cabin is defined by equation (3). The concentration of water inside the air can be calculated using these definitions.

This model has been developed during development of the real-time TEMO-property-library, so it was used to optimize the structure and interface of the library and has been optimized several times. As a result, the number of calculations is reduced to a minimum.

5.1.2 Results

Figure 5 shows the execution time of a global solver step split into the main contributors described on page 3. The four integration steps cause almost 98% of the work load. The output of the results can be neglected as it causes less than 1% of the work load. The summing up the integration steps does not equal the model runtime, as the global solver still has to evaluate the results and to perform auxiliary operations. The profiling itself increases the gap between the sum of each single contribution and the total runtime of the model, but this manipulation cannot be avoided.

The impact of profiling on the execution times can be estimated by capturing the total runtime of the whole model before and after the instrumentation. If every external function call and every (non-)linear block is profiled then the

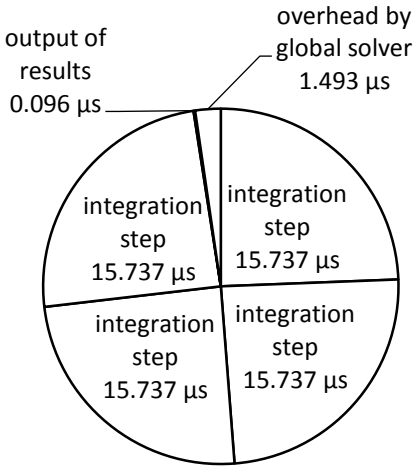

Figure 5. Integration steps cause main work load in the global solver steps

runtime of the model in both case studies increases by 4% at maximum.

As the external functions are called within the integration step, the execution time of the integration step inherits the profiling overhead caused by the external function calls. Therefore the gap between the sum of the executions times of integration steps and the output of variables is not as big as the increase of the whole model runtime. Because of this the overhead displayed in figure 5 can partially be assigned to the global solver.

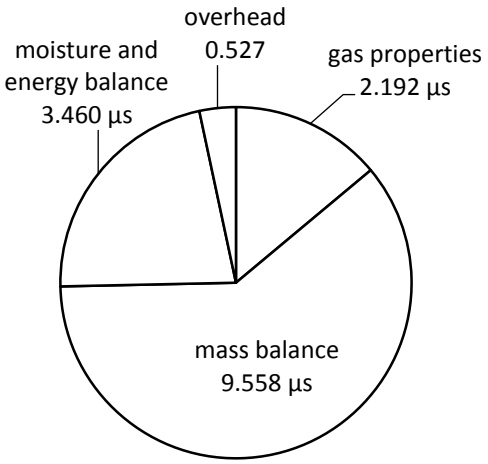

Figure 6. Work load of an integration step broken down to contributions

Figure 6 details the partitioning of the execution times within the global solvers integration step. There are just two algebraic loops, one representing the mass balance and another embodying the moisture and energy balance. The calls to the external gas property functions only occur outside the algebraic loops. The gap between these three summands and the execution time of one integration step is equal to the overhead. This overhead contains all calculations that are not external function calls and not algebraic loops. The rest is caused by the profiling methods.

The described model is already optimized so there is no algebraic loop or external function call causing extraordinary model runtime anymore. Figure 6 shows the balanced sharing of the given step time.

5.2 Case 2: Steady State Continuity

5.2.1 Description

This case was built up in Modelica using the real-time TEMO-property-library with the TIL-Library by TLK-Thermo and the Institute for Thermodynamics of the Technische Universität Braunschweig [10, 6].

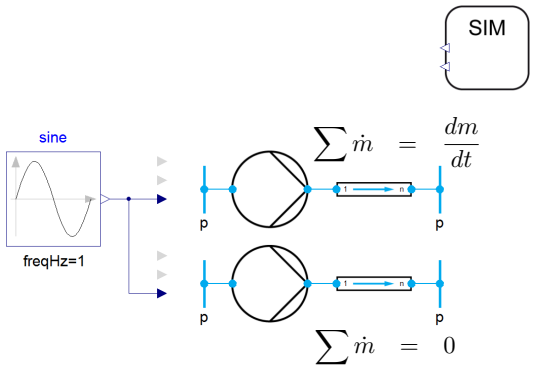

Figure 7. Non-linear pump and tube models with and without thermal expansion of incompressible liquid

As visualized in figure 7 the model is composed of two boundaries, a pump and a tube. The medium used in this case is incompressible water, so all fluid properties only depend on the temperature. Each property can be calculated using a Modelica function of temperature. The pressure increases at the pump is a second order function of the volume flow rate. The tube model is based on the finite volume concept and here composed of 2 cells. Within every cell there is a mass-, energy- and momentum-balance. The sine curve source sets the temperature at the inlet of the system. The temperature changes with an amplitude of $5K$ and an offset of $300K$.

In each component a parameter called "SteadyStateContinuity" is introduced by the TIL-Library. This parameter switches the mass balance of that component. In steady state the amount of mass flowing into a component equals the flow out at the same time (5). But in dynamic scenarios a mass flow is induced by a change of temperature due to the expansion of the fluid (7). The isobaric expansion coefficient β can be used to describe the expansion of a fluid due to temperature change (4). For incompressible liquids the density is not dependent on the pressure, so the change of density can directly be related to β.

$$\beta = -\frac{1}{\rho}\left(\frac{\partial \rho}{\partial T}\right)_p \tag{4}$$

$$0 = \dot{m}_{in} + \dot{m}_{out} \tag{5}$$

$$0 = \dot{m}_{in} + \dot{m}_{out} - V \cdot \rho \cdot \beta \cdot \frac{dT}{dt} \tag{6}$$

$$0 = \dot{m}_{in} + \dot{m}_{out} + V \cdot \frac{d\rho}{dt} \tag{7}$$

If a submodel for a component uses dynamic state continuity, the mass flow is directly related to the change of temperature. The DAE-system generated from this model must take this relation into account and hence the simulator has to increase the complexity of the DAE-system.

The change of density due to the change of temperature can be neglected in most cases of dynamic simulation since this effect is not relevant to the overall results of the whole model. By activating the steady state continuity the mass balance is not fulfilled anymore and mass may appear or disappear, but the main algebraic loop is broken into several smaller ones. There is no direct connection between mass balance and energy balance anymore, so the underlying smaller algebraic loops can be solved separately. This trick reduces the size of the DAE-System in paricular for the simulations of cycles.

For comparison, two subsystems with a tube and a pump were instantiated, where one is using the steady state continuity equation while the other one is not. The profiling should expose the work load caused by computing a negligible effect of density change by temperature.

5.2.2 Results

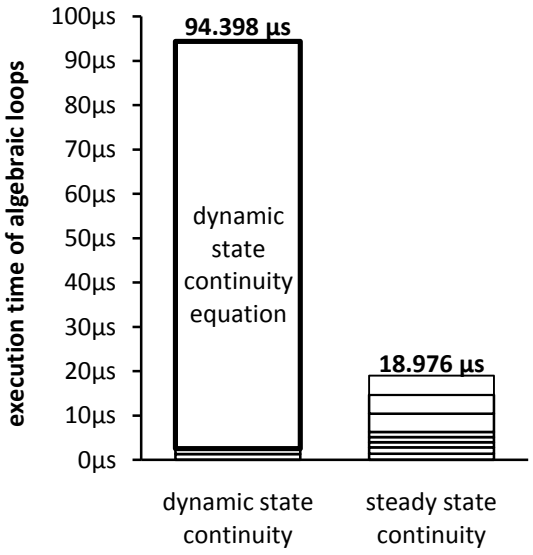

Figure 8. Profiling aids identifying the critical algebraic loop causing the main work load

Figure 8 visualizes the contributions of the each algebraic loop to the whole integration step separately for the steady state continuity submodel and the dynamic state continuity submodel. It allows the user to identify the critical calculations. The major work load in the submodel using dynamic state continuity equation is caused by that particular continuity equation. This algebraic loop generated from that equation has to be broken to reduce the execution time of this sub model. The simplification using steady state continuity is a method to break this loop into several smaller loops which can be solved more quickly.

Both models are equivalent in their results but differ with respect to their performance. As the global symbolic analysis performed during export selects different state

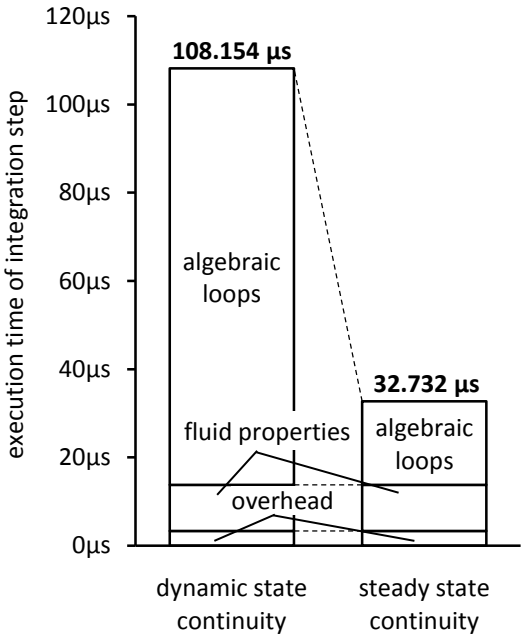

Figure 9. Solving times for algebraic loops in integration step of steady state continuity model are clearly faster

variables for each model, the contributions by the single algebraic loops cannot be related directly to the corresponding algebraic loops in the other model. The only way to link the algebraic loops back to the underlying equations is to trace back the involved variables.

These models were built using thermodynamic property functions which provide properties as a external function of temperature. This may cause additional algebraic loops if inverse calculation is needed, e.g. for finding the corresponding temperature to a given enthalpy. To avoid this the temperature inside the finite volumes of the tube is described as a differential state. As a result there is no algebraic loop including calls to the fluid property functions in both models.

Figure 9 relates the two models with respect to the work load caused by algebraic loops to the external function calls and the overhead of the global solver. The overhead and the amount of fluid property calculations is the same for both submodels. The contribution to the execution time of the integration step by the steady state submodel is significantly smaller.

There are other ways to break algebraic loops in a model, if the resulting relation between the variables embodies no or less important physical effects. For example a capacitor can be used to decouple the direct dependency between variables introducing a new differential state variable. Many physical models idealize a system that normally contains capacitors (e.g. the expansion of a tube due to a pressure increase) that have been neglected. Although the capacity may be very small, the effect is an uncoupling of the algebraic loops.

Figure 10 visualizes the mass flow at both sinks. The change of temperature at the inlet leads to a change of mass flow rate. In case of the dynamic state continuity equation

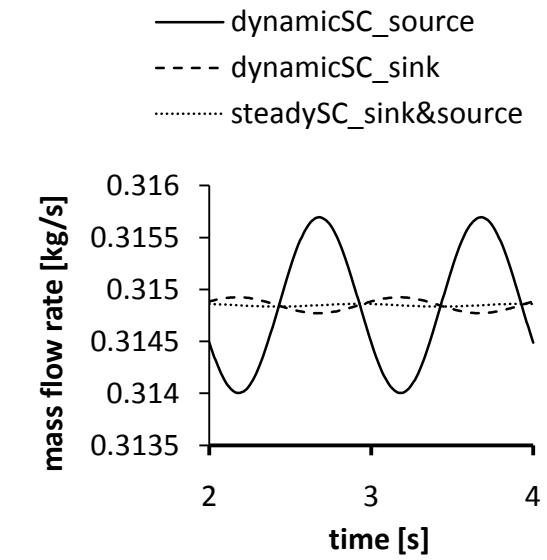

Figure 10. Error in mass flow due to usage of steady state continuity equation

the mass flow rate at the inlet is not equal to the outlet as a result of the expansion of the liquid. In case of the steady state continuity equation the mass flow entering all components is equal to the mass leaving the system and hence this also applies to the whole system.

The deviation between the mass flow rate entering and leaving those systems is smaller than 0.3%. So the simplification of using steady state continuity equation for dynamic state simulation is hardly affecting the results.

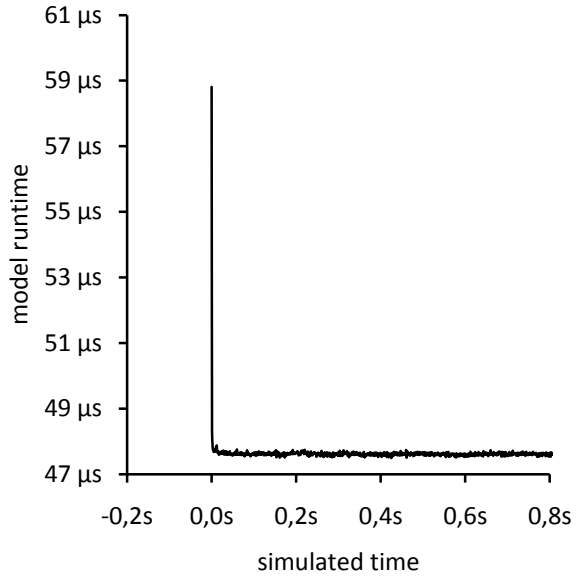

Figure 11. Model runtime during initialization is the bottle neck

The Figure 11 illustrates the temporal variation of the model runtime. After that first peak of $59\mu s$ during initialization of the model the runtime resides at a constantly lower level of $48\mu s$. There are no bigger changes or events inside the model after the initialization process. This case study was performed on a common Desktop PC with a Intel Pentium 4/ 540 CPU at 3.2 GHz without any realtime I/O-Interfaces.

6. Conclusion

This paper presents a brief description how profiling on source code that was automatically generated from Modelica tools like SimulationX can be performed under the target real-time operating system. Profiling can be a powerful tool aiding the user to understand the work load contributions by the internal algebraic loops. For optimization of Modelica models in general profiling should be introduced as a standard tool.

Acknowledgments

This work was funded by the Federal Ministry of Education and Research (BMBF), Germany, in the project TEMO (grant 01lS08013C).

We are thankful to Adina Aniculăesei for implementation support.

References

[1] MODELISAR (ITEA 2 07006). Functional mock-up interface for model exchange, January 26 2010.

[2] Tim Bird. Measuring function duration with ftrace. In *Proceedings of the Linux Symposium*, 2009.

[3] L. Dozio and P. Mantegazza. Linux real time application interface (rtai) in low cost high performance motion control. *Motion Control 2003*, 2003. Milano, Italy.

[4] S. Graham, P. Kessler, and M. McKusick. gprof: A call graph execution profiler. In *Proceedings of SIGPLAN Ś82 Symposium on Compiler Construction*, volume 17, number 6, pages 120–126, June 1982. SIGPLAN Notices.

[5] S. Graham, P. Kessler, and M. McKusick. An execution profiler for modular programs. In *Software - Practice and Experience*, volume 13, pages 671–685, 1991.

[6] M. Gräber, K. Kosowski, C. Richter, and W. Tegethoff. Modeling of heat pumps with an object-oriented model library for thermodynamic systems. In *6th Vienna International Conference on Mathematical Modelling*, Vienna, 2009. ISBN 978-3-901608-35-3.

[7] K. Hoffmann, F. Heßeler, and D. Abel. Rapid control prototyping with dymola and matlab for a model predictive control for the air path of a boosted diesel engine. In *E-COSM - Rencontres Scientifiques de l'IFP*, pages 25–33. Institut Francais du Petrole, 2006.

[8] Brian Kernighan and Rob Pike. Finding performance improvements: Excerpt from the practice of programming. *IEEE Software*, 16(2):61–65, 1999.

[9] Cosateq GmbH & Co. KG. Scale-RT, 2010.

[10] Christoph C. Richter. *Proposal of New Object-Oriented Equation-Based Model Libraries for Thermodynamic Systems*. PhD thesis, Technische Universität Carolo-Wilhelmina zu Braunschweig, 2008.

[11] S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable profiling and tracing for parallel scientific applications using c++. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 134–145. ACM, August 1998.

[12] Sameer Shende. Profiling and tracing in linux. In *Proceedings of Extreme Linux Workshop*, 1999.

[13] Karl Johan Åström, Hilding Elmqvist, and Sven Erik Mattsson. Evolution of continuous-time modeling and simulation. In *The 12th European Simulation Multiconference*, Manchester, UK, June 16 - 19 1998.

[14] Inc. The Mathworks. Execution and real-time implementation of a temporary overrun scheduler, 2006.

[15] VDI. Thermodynamische Stoffwerte von feuchter Luft und Verbrennungsgasen. *VDI-Handbuch Energietechnik*, 2000. VDI Richtlinie 4670.

Towards Improved Class Parameterization and Class Generation in Modelica

Dr. Dirk Zimmer

German Aerospace Center, Institute of Robotics and Mechatronics, Germany,
Dirk.Zimmer@dlr.de

Abstract

Class parameterization and class generation enhance the object-oriented means of Modelica, either by making them better accessible for the user or more powerful to apply for the library designer. Nevertheless, the current solution in Modelica does not properly distinguish between these two concepts, and hence it does not represent a fully satisfying solution. This paper presents a proposal or vision for a partial redesign of the language by separating class parameterization and class generation. In this way, the language becomes simpler and yet more powerful. The derived concepts may serve as guideline for future considerations of the Modelica language design.

Keywords ***language design, class-parameterization***

1. Introduction

This paper presents the concepts of class parameterization and class generation for equation-based modeling languages as Modelica. It is highlighted why these concepts are important for a modeling language and how they could be better regarded in the future.

The paper is organized as a proposal for a future design of Modelica. It is instructive in order to be concise. The suggestions are concrete in order to be illustrative. Nevertheless, what finally matters is the abstract idea behind our concept that could as well be finally realized in a different form.

To understand the current situation in Modelica [6,8], the problems of a language designer, and the motivation behind our proposal, let us review the most important fundamentals.

1.1 Processing Scheme

The translation of Modelica models into code for simulation purposes, involves several stages. These are depicted in Figure 1.

The semantics of the language concern nearly every part of this processing scheme. For instance, the causalization of an equation is done in stage 4, whereas the realization of a model extension concerns stage 2. Even with the same syntactic elements, a modeler can formulate expressions that belong to different stages.

An if-branch that depends on a parameter value corresponds to stage 2. If its condition is, however, dependent on a variable then it belongs to stage 4 and 5. In this paper, we are concerned with class parameterization and class generation. These two aspects belong to stage 2 of the processing scheme.

Modelica contains language constructs of all these processing stages in one single layer. This makes the language very powerful and highly convenient. To some degree this style results out of the declarative character of Modelica. It enables the modeler to focus on what he wants to model rather than thinking about how to create a computational realization. In this way, a modeler can achieve his or her goals without being fully aware of the underlying processing scheme.

Nevertheless, this puts an increasingly higher burden on the designers of such a language. Whereas the modeler does not need to know about the processing scheme, a language designer must have a very detailed knowledge. He or she is required to foresee all possible combinations with their potential problems that are introduced by a new language construct. As a language drifts towards higher complexity, this becomes a very hard task.

1.2 Structural Type System

The declarative style of Modelica is supported by a structural type system [1]. This means that the type results solely out of the structure of a class. Roughly speaking, type A is a sub-type of (or compatible to) type B, if all (public) elements of B are declared (by the same identifier) in A, and these elements are themselves sub-types of their counterparts in B.

In a structural type-system, the type is therefore independent from the methods used for its generation, and hence different lines of implementation may lead to compatible types. This is a big strength of structural type systems. Compatible types can have a common ancestor (mostly a partial model), but it is not required.

With respect to class generation and class parameterization, two additional definitions of compatibility must be concerned that impose additional restrictions on the simple sub-type relationship. Plug compatibility requires that, in addition to sub-type compatibility, no further connections are introduced that must be connected from outside. Plug compatibility is required when models get exchanged by class parameterization.

3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, October, 2010, Oslo, Norway.

EOOLT 2010 website:
http://www.eoolt.org/2010/

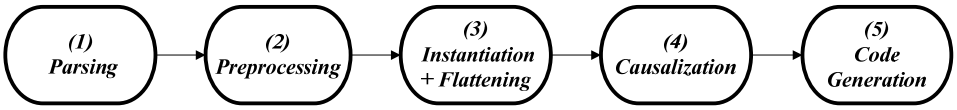

Figure 1: Processing Scheme of a Modelica Translator

Inheritance compatibility means that type A could replace type B as an ancestor for an arbitrary type C. To this end, the sub-type requirements are extended to protected elements. Inheritance compatibility is required for class generation purposes. The relation between these different sub-type relations is depicted in Figure 2.

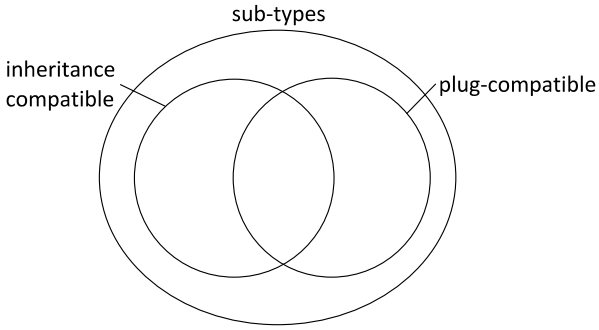

Figure 2. Set relation of different type requirements

1.3 Available Language Constructs

Let us briefly review those language constructs that are to be revised in the future. The use of all these keywords is then demonstrated by means of examples in the next section.

1.3.1 `replaceable` and `redeclare`

A modeler can declare a component B of model M as **`replaceable`**. By doing so, this component can be replaced, either in a possible extension of the model M or by a modifier that is applied to an instance M.

In order to replace the component B, the keyword **`redeclare`** (**`redeclare replaceable`** resp.) has to be applied. A new component A is then put into the place of B.

The type of the component A can be further constrained with the keyword **`constrainedby`**. It is applied at the original declaration that was marked as replaceable.

1.3.2 Parameters for classes

The keyword **`replaceable`** cannot only be applied to the declaration of components but also to the definition of **`models`**, **`packages`**, **`records`**, etc. To this end, the term replaceable model (or package, record, …) has been introduced.

Such definitions can then be extended by the use of the term **`redeclare model`** or (**`redeclare replaceable model`** resp.). Also the replacement of definitions can be constrained by the keyword **`constrainedby`**.

It is in general not possible to extend from replaceable model definitions. An exception is enabled by the term **`redeclare [replaceable] model extends`**.

1.3.3 Conditional Declarations

In addition to these tools, there are also conditional declarations available in Modelica. To this end, a short if-statement is appended to the normal declaration of a component. It is, however, not possible to combine conditional declarations with replaceable components or with components that are being redeclared.

2. The Important Difference between Class Generation and Class Parameterization

The presented language elements in Modelica may now serve two entirely distinct purposes: class parameterization and class generation. It is very important to make a proper distinction between these two concepts, since the lack of this distinction is the root of the current problems in Modelica. In order to clarify the situation, we present a representative set of examples for both concepts.

2.1 Examples for Class Parameterization

Class parameterization means that a class itself or a component is a parameter.

Class parameterization with respect to Modelica does mostly mean, model parameterization. To this end, a sub-component is made exchangeable by means of the parameter menu. Let us review three typical examples of this process.

2.1.1 Container Model (Wheels and Tires)

The container model is one of the most primitive methods to achieve class-parameterization. Essentially, it represents a set of conditionally declared components. Given a parameter value (mostly an enumeration value), one of the conditions evaluates to true, whereas all other components are disabled.

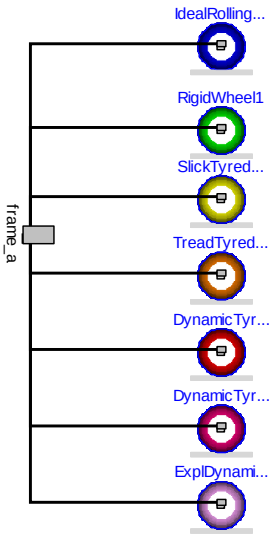

Figure 3. Container model for different wheel models

Figure 3 presents a container model that enables switching between different wheel models. The model parameterization is done indirectly by transforming a regular parameter into the conditional declaration of sub-models.

```
model MultiLevelWheel
public
  parameter ModLevels level //enumeration
  Interfaces.Frame_a frame_a;
  …
  IdealWheel wheel1(...)
    if level == ModLevels.IdealWheel;
  RigidWheel wheel2(...)
    if level == ModLevels.RigidWheel;
  SlickTyredWheel wheel3(...)
    if level == ModLevels.SlickTyredWheel;
    …
equation
  connect(wheel1.frame_a, frame_a);
  connect(wheel2.frame_a, frame_a);
  connect(wheel3.frame_a, frame_a);
  …
end MultiLevelWheel;
```

Given the construct of **replaceable**/**redeclare**, this design pattern has actually become redundant. It is, however still applied. It is better suited if the sub-models shall not be public but protected. Another application results, if the standard dropdown list (of the Dymola GUI) for replaceable components is not the preferred parameterization since another user interface is demanded.

2.2 Exchangeable Resistor Model

The standard method of model parameterization is performed by means of a replaceable model. An electric circuit may contain a resistor component. If it is declared as replaceable:

```
model Circuit1
  replaceable Resistor R1(R=100);
  …
end Circuit1;
```

A potential user of this circuit model may now exchange the resistor

```
model Test
  Circuit1 C(
    redeclare ThermoRes R1(R=100)
   );
   …
end Test;
```

If the circuit contains two resistors, each can be redeclared separately. Alternatively, the circuit can have a parameter for the model definition.

```
model Circuit2
  replaceable model R = Resistor(R=100);
  R R1;
  R R2;
  …
end Circuit2
```

A user can now redefine the model definition:

```
model Test
  Circuit2 C(
    redeclare model R = ThermoRes(R=100)
  );
   …
end Test
```

2.2.1 Media-Exchange

Having parameters for class definitions enables more advanced modeling techniques. The models of the Modelica Fluid [4,5] library serve as a good example. Here each fluid model contains a parameter for a package definition. Given this package, the model declares now those package members that it requires.

```
model TemperatureSensor
  replaceable package Medium =
    Interfaces.PartialMedium;
  Interfaces.FluidPort_in port(
    redeclare package Medium =
      Medium
    )
  Medium.BaseProperties medium;
  Modelica.Blocks.Interfaces.RealOutput
T(unit="K");

equation
  …
  port.p = medium.p;
  port.h = medium.h;
  port.Xi = medium.Xi;
  T = medium.T;
end Temperature;
```

2.3 Examples for Class Generation

Class Generation is a collective term for all those methods that are used to generate a new class. Most commonly, the new class is created out of one or more existing ones.

The most common technique of class-generation in Modelica is class extension that is represented by the keyword extends.

Mostly, replaceable and redeclare are used for class parameterization, but there are also applications for class generation. The following two examples shall demonstrate this.

2.3.1 MultiBondLib

The MultiBondLib [11] features various mechanical libraries based on the bond-graphic modeling methodology. There is the planar mechanical library and the 3D-mechanical library. In addition, there is the 3D-mechanical library that includes the modeling of force-impulses. This library was derived from its continuous-system version. To this end, the connector of the classic mechanical package was made replaceable.

```
connector Frame
  Potentials P;
  flow SI.Force f[3];
  flow SI.Torque t[3];
```

```
end Frame;

model FixedTranslation
  replaceable Interfaces.Frame_a frame_a;
  replaceable Interfaces.Frame_b frame_b;
  …
end FixedTranslation;
```

The connector of the impulse library was then extended from its continuous version.

```
connector IFrame
  extends Mech3D.Interfaces.Frame;
  Boolean contact;
  SI.Velocity Vm[3];
  SI.AngularVelocity Wm[3];
  flow SI.Impulse F[3];
  flow SI.AngularImpulse T[3];
end IFrame;
```

Finally, each component of the impulse-library was inherited from its continuous counterparts, had its connectors replaced and the required impulse equations added:

```
model FixedTranslation
  extends Mech3D.Parts.FixedTranslation(
    redeclare Interfaces.IFrame_a frame_a,
    redeclare Interfaces.IFrame_b frame_b
   );
  …
equation
  …
  frame_a.contact = frame_b.contact;
  frame_a.F + frame_b.F = zeros(3);
  frame_a.T + frame_b.T +
    cross(r,R*frame_b.F) = zeros(3);
  frame_a.Vm + ( transpose(R) *
    cross(frame_a.Wm,r) ) = frame_b.Vm;
  frame_a.Wm = frame_b.Wm;
end FixedTranslation;
```

Making the connector directly replaceable is not the preferred solution given the current means of the language. It would be better to use a model parameter C (via replaceable model) for the connectors and declare the connectors by the use of C. At its time of creation, however, this solution was not available for the MultiBondLib.

2.3.2 Medium equations in the MediaLib

Another example for class generation can be found in the Modelica MediaLib. Here, an individual package is created for each medium. Among other members the package contains a model `BaseProperties` that describes those balance equations that are specific to the medium (e.g. the ideal gas law).

A new medium may now inherit from an existing medium package and redefine its `BaseProperties` model. In this way a class is generated for each medium:

```
partial package SingleGasNasa
  extends PartialPureSubstance(…)
  redeclare model extends BaseProperties(…)
equation
  …
  MM = data.MM;
  R = data.R;
  h =h_T(data, T, h_offset);
  u = h - R*T;
  d = p/(R*T);
  state.T = T;
  state.p = p;
end BaseProperties;
```

2.4 Foresight versus Hindsight

Since both, class parameterization and class generation are performed during the preprocessing stage in the translation process, it may be tempting to use one set of tools for both purposes as is done in Modelica. However, this turns out to be problematic because of the entirely distinct motivation behind these two concepts.

Class parameterization is requested by the model designer to be performed by a user of its library. Thus, it is performed in *foresight* since the corresponding parameterization needs to be declared. Rules for class parameterization must be rather strict to prohibit abuses by the user to a meaningful extent.

In contrast, class generation is done in *hindsight*. It is performed by the model-designer and requested from a previous library. Since it is done in hindsight and mostly performed by experts, rules for class generation should not be prohibitive. It is not possible to foresee which models might be extended; so a potential keyword **`extendable`** does not make much sense. It is, however, also not foreseeable which elements might be **`redeclared`**; so the keyword **`replaceable`** is inappropriate. Prohibitive measures will tend rather to corrupt existing classes than to prevent the faulty creation of new classes.

2.5 Different Aspects of the Type System

Another vital difference between class parameterization and class generation is highlighted by the criteria of the type system that are relevant for each concept.

A proper class parameterization requires that the new type A is compatible to the original type B. Obviously A must be a sub-type of B. An even more strict requirement is that it needs to be plug-compatible since it is not possible (and certainly not convenient) to introduce new connections into a parameterized model.

Plug-compatibility is of no relevance for class generation. When a new class is generated, new connections can also be introduced in an effortless way. Instead, it is important that the new type is inheritance-compatible since potential extensions of a redefined model ought to remain valid.

Evidently, separate aspects of the type system need to be concerned for both tasks.

2.6 Current Deficiencies

The confusion of class parameterization and class generation involves a number of disadvantages:

- *Non-uniform parameterization*: The syntax that has been chosen for class parameterization purposes is different to those of normal parameters. One unfortunate consequence of this decision is that class

parameterization becomes inaccessible for normal parameter computations. For instance, it is not possible to combine if-statements with redeclarations. This means that redeclarations cannot be coupled to conditions.

- *Inappropriate sub-elements*: Since model parameters are not properly declared as parameters but more as a replaceable sub-element this leads to inappropriate structures. For instance, models that contain sub-packages. It makes sense that a model cannot contain a package, but it makes no sense that a model can contain a package just because it is replaceable.
- *Prohibitive class generation*: Since potential redeclarations and redefinitions must be marked as replaceable in advance, the options for class generation are unnecessarily limited. Often this leads to an ex post modification of the original library in order to enable the desired class generation
- *Unwanted parameterization*: Since potential redeclarations or definitions for the purpose of class generation need to be marked as replaceable an unwanted parameterization is introduced into the models. In order to avoid this, the replaceable objects are often moved to the protected section.
- *Unnecessary restriction*: To extend from replaceable model definitions is currently prohibited in Modelica (with one exception). This restriction will turn out to be unnecessary.
- *Overelaborated syntax*: The current syntax is simply more complicated than actually necessary and can be simplified.

3. Design Decisions

For the partial redesign of Modelica, we establish the following guidelines:

- Separate class parameterization and class generation

  We want to clearly separate class parameterization from class generation. In this way, the specific needs and motivation of each concept can be optimally taken into account.

- Give classes first class status on the parameter level

  We want to treat class parameters just as any normal parameter. There is no reason why parameters should be restricted to base-types or quasi base-types. This will simplify and unify the syntax. Furthermore, class parameterization can be integrated in the normal computation process for parameters.

- Enable non-prohibitive class generation

  Class generation shall be performed by a special subset of keywords. It shall be designed in a way that it is not hampered or prohibited by means that require foresight. Maximum freedom should be given to the modeler in order to create new classes. On the other hand side, existing classes shall be protected from corruption.

- Unify and simplify the language

  The complete language should be simpler and more powerful after the revision. It should also be more intuitive to understand and to learn.

4. Improved Class Parameterization

In this section, we will propose new language constructs for class parameterization. In order to show their potential applications and highlight their advantages, we will review the examples of chapter 2.1.

4.1 Unification of Expression

In a first, preparatory step, we integrate the expression of classes into normal statements. To this end, we have to slightly change the modifier syntax of an expression: the modification is now applied in curly brackets instead of round parentheses.

This change has been implemented in order to make a class-definition with its modifier distinguishable from a function-call with its argument-list. In this way you know that `foo(x=a)` represents a function call but `bar{x=a}` represents a class-definition with its modifier. The term `baz{p=b}(a)` represents then consequently a parameterized function call.

Since classes can be used in expressions, the language power is increased, e.g., by using classes in if-clauses or as arrays:

```
// The result of this if-clause is a class
if expr then foo2{x2=b} else foo2{x2=c};

// An array of 4 classes
 foo2[4]
 foo2{x2=a}[4]
```

One might hesitate, to integrate class-expressions as basic part of normal expressions, since this gives classes a first-class status [2,3] and opens up the grammar quite substantially. It might seem smarter (and easier to achieve) to form two separate kinds of expressions that are distinguished on the top level: normal expressions and class expressions. However, this is misleading for the following reasons.

Firstly, normal expressions and class expressions can both start with a name. This means that an undefined number of look-up tokens are required to distinguish these two kinds of expressions. Practically this means that an extra keyword is needed, but this leads to an ugly and unpractical syntax. Also, many syntax elements would need to be doubled and still two kinds of grammars would be required. Hence such a solution would not be fully generic.

`A1.B2.C3.Model{…}` → class expression
`A1.B2.C3.Function(…)` → normal expression

Secondly, the integration of class-expressions into normal expressions provides an important generalization for future language extensions. Whereas many syntactic formulations such as `foo{x2=3} + foo{x2=2}` are semantically still invalid for this proposal, this may change in future revisions. Let us envision a future

version of Modelica (5 or 6) that enables anonymous declarations of models or records. Then, the former statement `foo{x2=3} + foo{x2=2}` may become valid if, for instance, `foo` is a record and overloads the + operator. Hence the integration of class-expressions opens up a number of fruitful opportunities for future language revisions. It is notable that the first-class status of higher-level language constructs is absolutely common in contemporary programming languages. Even a few equation-based modeling languages (Sol [9,10], Hydra [7], Modeling Kernel Language [2]) have explored this important topic.

In this proposal, only the following uses of class-expressions shall be semantically supported. All other uses yield error messages.

- Pure class expressions: `foo{x=y}`
- Class expressions in if-statements: **`if`** a **`then`** `foo{x=y}` **`else`** `bar{x=y}`
- Array-lists of class-expressions: `{foo{x=y}, bar{x=y}, …}`

4.2 Say It As You Want It: Treat Component Parameters as Normal Parameters

If a component (let us suppose: a resistor) shall be a parameter of a model, it is the most natural thing, just to write it down as a normal parameter. Instead of the awkward formulation:

```
model Circuit1
  replaceable ThermoRes R1(k=0.5)
  constrainedby Resistor(R=100);
  …
end Circuit1
```

simply write it as a component parameter:

```
model Circuit1
  parameter component Resistor R1{R=100} =
ThermoRes{k=0.5}
  …
end Circuit1
```

A user of this circuit model may now give a new parameter value and thereby replace the prior model.

```
model Test
  Circuit1 C{R1 = ThermoRes{k=1.2}};
  …
end Test
```

The type of the parameter hereby represents the constraint type for the parameterized model. Naturally one can apply modifiers also on the constraint type, and of course the new resistor type must be plug-compatible to this constraint type.

The keyword **`component`** is necessary in order to avoid potential ambiguities. These originate from the fact that the formulation of a component parameter is a language construct that performs two tasks at the same time. One, it enables direct class-parameterization of a component. Two, it declares a component that invokes an instance.

Hence it must be clarified if the = operator assigns a value

```
parameter Real r = 1;
```

or a component (sub-model)

```
parameter component Resistor R = ThermoRes;
```

In these cases, the meaning is clear, but when records are concerned both interpretations of the assignment are meaningful:

```
//value assignment
parameter Complex c1 = Complex.j();

//component assignment
parameter component Complex c2 =
                        Quaternion;
```

In fact, the keyword does not just change the interpretation of the assignment, but also if the values of the instance shall be constant or not. In the example above, `c1` is constant-valued but `c2` may express variable values.

4.3 Say It As Want You Want It: Treat Class Parameters as Normal Parameters

The very same can be done for parameters that identify class definitions, such as model parameters or package parameters. Again, the best solution is to simply write it down as one wants it to have. So, instead of writing:

```
replaceable model R1 = Resistor{R=100}
  constrainedby Resistor;
```

you can simply turn the model into a parameter:

```
parameter model Resistor R1 =
                  Resistor{R=100};
```

Since such parameters will ultimately always be used for class parameterization, plug-compatibility shall also be required here. In this way, the temperature sensor of section 2.1 could be formulated as follows:

```
model TemperatureSensor
 parameter package
  Interfaces.PartialMedium Medium;

  Interfaces.FluidPort_in port{
    Medium = Medium}
  Medium.BaseProperties medium;
  Blocks.Interfaces.RealOutput T{unit="K"};
equation
  …
end Temperature;
```

4.4 Improved Computational Power

One obvious advantage is that the language has been unified. Now, the same notation is used for all kinds of parameters. It has also become simpler. The keywords **`replaceable`**, **`redeclare`** and **`constrainedby`** are not needed any more.

Another major advantage is that class parameters can be computed with as any other parameter. In this way,

conditional declarations become redundant in many cases. Let us review the example of the container model. Here we had to transform an enumeration into a class. This was done by number of conditional declarations. Now, we have the option to use an array of model parameters for this purpose.

```
model MultiLevelWheel
public
  parameter TModLevels level //enumeration
  Interfaces.Frame_a frame_a;
  …
protected
  final parameter model BaseWheel
    wheelModels[7]= { IdealWheel {…},
                      RigidWheel{…},
                      SlickTyredWheel{…},
                      … };
  final parameter component BaseWheel wheel
    = wheelModels[level];
equation
  connect(wheel.frame_a, frame_a);
end MultiLevelWheel;
```

Here we can organize different model parameters in an array. In the same way, this could be done in a record. It is important to notice that class parameters become accessible to all kinds of computations. Especially useful is the conditional evaluation:

```
parameter Boolean constantTemp = true;
final parameter BaseTempModel
ambientTemperature =
  if constantTemp then ConstTempModel{…}
  else TempFileHistory{…};
```

One inconvenience of the proposed notation is that it sometimes leads to redundant formulation. In some applications, the default parameter value will equal the type constraint.

```
parameter component Resistor R1{R=100} =
                    Resistor{R=100};
```

Here, `Resistor{R=100}` had to occur twice. If this turns out to be a frequent case, one may consider adding a new keyword **`itself`** in order to provide some syntactic sugar.

```
parameter component Resistor R1{R=100} =
                    itself;
```

5. Improved Class Generation

Having available powerful and well-integrated means for class parameterization, we can now provide separate means for class generation. To this end, we need to keep our eye on two different targets:

1. Enable the convenient creation of new classes out of existing classes.
2. Prevent the corruption of existing classes.

The second goal is easily forgotten, but it is equally important to the first goal. Again, we explain the new language constructs by means of examples and review for this purpose the models from section 2.2.

5.1 The New Role of Redeclared

We replace the former keyword **`redeclare`** by a new keyword **`redeclared`**. The new keyword is now solely implemented for the purpose of class generation. It can actually be applied similar to the former keyword. Let us therefore review the example of the mechanical impulse library where we wanted to exchange the continuous-system connector with an extended counterpart.

```
model FixedTranslation
  extends
Mechanics3D.Parts.FixedTranslation;
  redeclared Interfaces.IFrame_a frame_a;
  redeclared Interfaces.IFrame_b frame_b;
  …
equation
  …
  frame_a.contact = frame_b.contact;
  frame_a.F + frame_b.F = zeros(3);
  frame_a.T       +       frame_b.T       +
cross(r,R*frame_b.F) = zeros(3);
  frame_a.Vm
  + transpose(R)*cross(frame_a.Wm,r)
  = frame_b.Vm;
  frame_a.Wm = frame_b.Wm;
end FixedTranslation;
```

This solution is very similar to the existing methods in Modelica, but there are crucial and important differences. Most importantly, the elements are not **`redeclared`** in the modifier of the extension belonging to the existing class, but in the public section of the new class. In this way, we prevent the existing class from being corrupted and we prohibit an abuse of the keyword **`redeclared`** for the purpose of class parameterization. For this reason, the use of **`redeclared`** in modifiers is strictly forbidden.

Furthermore the **`redeclaration`** can be applied to all inherited components without restriction. It is not necessary (nor desirable) to mark these components as replaceable beforehand in the inherited models. Doing so would be not even superfluous but even harmful since...

1. it would require an inappropriate amount of foresight.
2. it is very tempting to add the **`replaceable`** keyword ex post and thereby to corrupt the original models that should not be touched
3. the **`replaceable`** keyword actually introduces an unwanted parameterization of the original model.

Hence the original translation model can be formulated just normally without replaceable connectors.

```
model FixedTranslation
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;
    …
end FixedTranslation;
```

Also the class requirements that are imposed on redeclarations are different. For class parameterization the replaced models must have been plug-compatible to the constraint or the original model, respectively. This was necessary since additional connections could not be introduced anymore. In the case of class generation, it would be easy to add a new connection in the new model, and hence the only requirement is that the **redeclared** model is a sub-type of the original type. Plug-compatibility is not requested anymore.

5.2 The New Role of Redefined

Another application of class generation is the redefinition of whole models, packages, etc. To this end, the keyword redefined is provided. When class definitions are inherited (for instance by inheriting a package), any definition can be redefined. To clarify this, let us review the example of the MediaLib.

```
partial package SingleGasNasa
  extends PartialPureSubstance{…};

  redefined model BaseProperties{…}
    extends itself;
  equation
    …
  end BaseProperties;

end SingleGasNasa;
```

In principle, not much has changed on the syntax level. Nevertheless, there are again important differences to the prior solution. First of all, the original `BaseProperties` model did not need to be marked as replaceable. The reasons for this are exactly the same as for redeclared components. Correspondingly, the use of redefined is also banned from modifiers.

Second, the use of **redefined** on class definitions imposes different type restrictions than the use of **redeclared** on components. Since class definitions might get extended within the inherited package, the redefined type must be inheritance-compatible to the original type. It is still possible that there are conflicts for inheritance since the redefined class may generate a sub-type that leads to name clashes. However, these type of errors can be fairly well reported and it is a rather uncommon situation.

Since, now the proper type restrictions are applied (and not the inappropriate plug-compatibility), the redefinition of base-classes is enabled. In fact, this can be a powerful design tool. Let us consider once more the mechanical impulse library of the MultiBondLib: instead of redeclaring the connectors in each component, it would be far more elegant, to extend the whole package and redefine the connector. Then all components adapt automatically and the missing equations can be added to each component by providing an extended redefinition of itself.

```
package Mechanics3D;

  connector Frame
    Potentials P;
    flow SI.Force f[3];
    flow SI.Torque t[3];
  end Frame;

  connector Frame_a extends Frame;
      …
  end Frame_a:

  model FixedTranslation
    Interfaces.Frame_a frame_a;
    Interfaces.Frame_b frame_b;
    …
  end FixedTranslation;

  …
end Mechanics3D;

package Mechanics3DwithImpulses;
  extends Mechanics3D;

  redefined connector Frame extends itself
    Boolean contact;
    SI.Velocity Vm[3];
    SI.AngularVelocity Wm[3];
    flow SI.Impulse F[3];
    flow SI.AngularImpulse T[3];
  end Frame;

//The extended model Frame_a will automatically adapt and does not need to be redefined.

  redefined model FixedTranslation
  //Here the connectors do not need to be  redeclared
  equation
    …
    frame_a.contact = frame_b.contact;
    frame_a.F + frame_b.F = zeros(3);
    frame_a.T + frame_b.T +
      cross(r,R*frame_b.F) = zeros(3);
    frame_a.Vm + (transpose(R)*
      cross(frame_a.Wm,r))= frame_b.Vm;
    frame_a.Wm = frame_b.Wm;
  end FixedTranslation;
  …

end Mechanics3DwithImpulses;
```

6. Final Review

Let us quickly review the proposed modifications of the language.

6.1 Simplification of the Language

The grammar has become simpler and more unified (see appendix). The keywords **replaceable** and **constrainedby** have become obsolete. Non-uniform and complicated construction as: **redeclare replaceable model extends** can also be removed from the language. The keyword **redeclare** is replaced by **redeclared** or **redefined** respectively that have a different meaning. These new keywords are also removed from the modifiers, which simplifies the grammar.

6.2 Higher Degree of Expressiveness

The unification of class parameters and normal parameters not only simplifies the grammar and makes the language more intuitive to understand. It also improves the expressiveness of the language. Now we can compute with class parameters just as with normal parameters and create all kinds of models.

The separation of class generation from class parameterization helps to protect existing classes from the introduction of unwanted parameterization. Since class generation is now applicable to all components (but only in a new class), less foresight is required and more can be done in hindsight without having to modify the original models. This separation also helps to impose the correct class requirements for each operation.

6.3 Deficiencies of this proposal

Syntactically, the introduction of the keyword **`itself`** is regrettable. Here the former notation was more convenient. However, the new grammar enables to formulate a component or class parameter without a default value and, in this way, to enforce a parameterization in an evident manner. This is not possible in the current grammar. Also, the keyword **`itself`** can be reused in the extends-clause and here it leads to a more natural and better understandable expression.

Semantically, the new notation almost completely covers the expressiveness of the current Modelica. Only for the redefinition of classes, there exists no short notation. For instance, if the redefinition of a model occurs in a modifier of a replaceable class, then we have a problem.

```
replaceable package Medium =
  PureSubstance(redeclare model
    BaseProperties = myProperties
  )
```

The new language version (maybe rightfully) prohibits such ad-hoc class-generations. To this end, we have to create a new class and assign it to the model parameter.

```
partial package newMedium
  extends PureSubstance;
  redefined  model BaseProperties
   = myProperties;
  …
end newMedium;

package parameter Medium = newMedium;
```

Please keep in mind: this is only the case for this specific kind of redefinitions. Most of the current redeclarations get replaced by parameter assignments and these are totally uncritical. Hence, this is a rather uncommon case and since such a transformation is better implemented manually. To our knowledge, this application does hardly occur.

6.4 Backward Compatibility

Backward compatibility is major issue since this proposal would definitely represent a drastic change of the Modelica language. Unfortunately, it is not easy to achieve. Our proposal distinguishes class generation from class parameterization. This was not done before. Hence, one needs to separate what is currently intermixed. It is possible to do so for 90% of all occurring cases but there remain, inevitably, some cases that cannot be resolved automatically.

6.5 Final Conclusions

The main two points of this paper are:

- It is highly meaningful to distinguish class generation from class parameterization since entirely different motivations are underlying these two concepts.
- Introducing class expressions (and thereby giving classes a first-class status) can drastically simplify the grammar while making the language more powerful. Class parameterization is only one possible application of class expressions.

Appendix

These are the resulting grammar changes to the Modelica language. Please note, this represents not our exact proposal. We provide this just in order to show the simplifications and to concretize the conceptual explanations of this paper.

The following keywords are removed from the language:

`replaceable`
`constrainedby`
`redeclare`

The following keywords are introduced into the language**:**

`component`
`redeclared`
`redefined`
`itself`

The following grammar changes are listed according to the order of the language specification. New elements are underlined, removed elements are ~~scratched.~~

B 2.2 Class Definition

```
element:
       import_clause |
       extends_clause |
       [ redeclare ]
       [ redeclared ]
       [ final ]
       [ inner ] [ outer ]
       (( class_definition |
       component_clause ) |
       replaceable ( class_definition |
       component_clause)
         [constraining_clause comment])
```

B 2.3 Extends

```
extends_clause :
```

```
      extends (name | itself)
     [ class_modifications ] [annotation]

contraining_clause:
      constrainedby name
[class_modification ]
```

B 2.4 Component Clause

```
type_prefix:
      [flow | stream]
      [discrete | (parameter  [par-
specifier]) | constant ]
      [input | output]
par_specifier:
      (component | class | model | record
| block | connector | type |
       package | function | operator |
operator function | operator record)
```

B 2.5 Modification

```
modification:
      class_modification ["="
(expression|itself) ]
      | "=" (expression|itself)
      | ":=" expression

class modification :
      "{" element_modification {"," 
element_modification }  "}"

argument_list:
      argument {“,” argument}

argument :
      element_modification_or_replaceable
      element_redeclaration

element_modification_or_replaceable
      [ each ] [ final ] (element
modification | element_replaceable)

element_modification:
      [ each ] [ final ] name [
modification ] string_comment

element_redeclaration :
      redeclare [ each ] [ final ]
      ( ( class_definition |
component_clause) | element_replaceable )

element_replaceable:
      replaceable (class definition |
component_clause1)
      [constraining clause ]

component_clause1:
      type_prefix type_specifier
component_declaration1

component_declaration1 :
      declaration comment
```

B 2.7 Expressions

```
primary:
      UNSIGNED_NUMBER
      | STRING
      | false | true
      | class_expression
      | (der | initial)
(function_call_args)
      | component reference
      | "(" output_expression_list } ")"
      | "[" expression_list { ";"
expression_list } "]"
      | "{" function_arguments "}"
      | end

class_expression:
 name [class_modification]
 [(function_call_args)]
```


Acknowledgements

I’d like to thank Martin Otter for the enjoyable and fruitful discussions on this topic and for further suggestions. I also like to thank Hans Olsson for further critical remarks.

References

[1] David Broman, P. Fritzson, S. Furic. Types in the Modelica Language. *Proceedings of the Fifth International Modelica Conference*, Vienna, Austria Vol. 1, 303-315, 2006.

[2] David Broman and Peter Fritzson Higher-Order Acausal Models *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT)*, Paphos, Cyprus, 2008.

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

[4] Hilding Elmquist, H. Tummescheit, Martin Otter. Object-Oriented Modeling of Thermo-Fluid Systems. *Proceedings of the 3rd Modelica Conference*, pp 269-286, 2003.

[5] Rüdiger Franke, F. Casella, M. Otter, M. Sielemann, S.E. Mattson, H. Olsson, H. Elmquist. An Extension of Modelica for Device-Oriented Modeling of Convective Transport Phenomena. *Proc. 7th International Modelica Conference*, Como, Italy, 2009.

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

[7] George Giorgidze and Henrik Nilsson. Higher-Order Non-Causal Modelling and Simulation of Structurally Dynamic Systems. In: *Proceedings of the 7th International Modelica Conference*, pp. 208 - 218, Como, Italy, 2009.

[8] Modelica® - *A Unified Object-Oriented Language for Physical Systems Modeling*. Language Specification, Version 3.2, www.modelica.org.

[9] Dirk Zimmer. *Equation-Based Modeling of Variable-Structure Systems.* PhD-Dissertation, ETH Zurich, 2010.

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

[11] Dirk Zimmer and F.E. Cellier, The Modelica Multi-bond Graph Library. In: *Simulation News Europe*, Volume 17, No. 3/4, pp. 5-13, 2007.

Notes on the Separate Compilation of Modelica

Ch. Höger F. Lorenzen P. Pepper

Fakultät für Elektrotechnik und Informatik, Technische Universität Berlin
{choeger, florenz, pepper}@cs.tu-berlin.de

Abstract

Separate compilation is a must-have in software engineering. The fact that Modelica models are compiled from the global sources at once results from the language design as well as from the way compiled physical models are finally simulated. We show that the language in fact can be compiled separately when certain runtime conditions are met. We demonstrate this by transforming some specific Modelica language features (structural subtyping and dynamic binding) into a much simpler form that is closer to current OO languages like C++ or Java.

Keywords Modelica, Separate Compilation

1. Introduction

Separate compilation is the state of the art in today's software engineering and compilation tools. Therefore it is also a natural request for Modelica tools. Large models (having hundreds or even thousands of equations) evolve over time, undergo small incremental changes and are often developed by whole teams. So it is unacceptable in practice, when a small modification in one class causes the recompilation of hundreds of unchanged classes. This could be avoided by the creation of smaller compilation units which are finally combined to create the desired model.

Unfortunately, it is not a priori clear that this approach is feasible for Modelica, since some features of the language are quite complex to handle with separate compilation. Potential problems could in particular arise in connection with the process of flattening, structural subtyping, inner/outer declarations, redeclarations and expandable connectors. We will sketch in the present paper solutions for all of these issues with the exception of expandable connectors, which would require a more in-depth discussion of the operational semantics of runtime instantiation.

The flattening of the whole model can actually be avoided by using object-oriented features of the target language: Instead of creating the set of equations, a compiler can directly translate the object tree into the target language. The equations can then be simply collected at runtime. This principle also leads to a elegant solution for the compilation of expandable connectors, as we will show in Section 2.

While Modelica's type system gives great flexibility in code re-usage, its translation into an object-oriented language with nominal subtyping like C++ is not straightforward. We will show how a compiler can handle this with the usage of coercions in Section 3.

Rumor has it that *separate compilation* is made very complex through the presence of Modelica's **inner** and **outer** pairing. We will show (in Section 4) that this rumor is unsubstantiated: By combining some kind of parameterization with standard typing principles we can not only solve the separate-compilation issue in a straightforward manner but also obtain a very clean and well comprehensible semantic definition of the inner/outer principles.

Finally we will summarize the costs and drawbacks of the separate compilation of Modelica models in Section 5.

1.1 Related work

Although separate compilation of Modelica models seems to be an important topic, only little research has yet been done in this field. In [8] and [9] some extensions to the language are proposed that would allow users to write models which can be compiled separately with nearly no impact on the quality of the generated code (as mentioned in Section 5). The work presented in [12] aims to decide wether or not separate compilation is possible (with respect to causalization) and worth the price for a given model. While both approaches do not allow the separate compilation of arbitrary compilation units, they could be combined easily with our method to raise the quality of generated code.

2. Principles of Separate Code Generation

As stated earlier the main reason for separate code generation is to reduce the overall compilation time by re-using output units that have not been affected by source level changes after the last compiler run. This at least requires a compiler to create output files for each input file given. Furthermore the content of those generated files should not depend on any context of their usage but only on their interfaces: If content from a source file is used two or more times the output file(s) should be usable in all those cases.

While the above requirements would generally suffice for separate compilation, a good design additionally enforces the separation of the compiler from the build sys-

3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. October, 2010, Oslo, Norway.

tem (so that the build system of choice can be used). This requires the compiler to produce a predictable set of output files for a given input file and the language to allow the detection of dependencies with simplistic tools. A good example for the usage of this principle is the compilation of C++ programs with GNU Make: Since the C++ compiler will create one object file per given input file and dependencies are clearly marked as file-wise includes, the build system can decide which source files need to be rebuilt given only modification times and regular pattern rules. Although imports in Modelica are only allowed from packages (and packages have a clear mapping into a file system layout) a Modelica type name cannot be mapped to the file it is defined in by such pattern rules. Therefore a complete dependency analysis must fully implement Modelica's (rather complex) lookup mechanisms. For smaller projects this dependency analyis might still be handled manually, but for complex uses one would have to generate e.g. makefiles automatically, to get all the benefits from incremental builds.

While the possibility to have a more fine grained build system is already extremely useful, there is another advantage of separate compilation that should be considered for Modelica: Currently Modelica library developers have to hide their expert knowledge from their customers by obscure methods like the Digital Rights Management that is part of the current Modelica specification[1]. If a library would be compiled separately, only the source code of the interface (comparable to a C/C++ header file) would have to be shipped. Also a clear separation between interface and code would be possible without any additional cost.

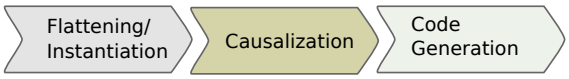

Figure 1. Common compilation of Modelica.

Unfortunately the compilation target of a Modelica model is a single matrix consisting of both differential and algebraic equations (short DAE). The usual processing scheme can be seen in Figure 1: After parsing (and correctness-checking) a Modelica model is flattened, meaning that both the inheritance and instance trees are resolved into sets of variables and (acasual) equations. Those equations can then be causalized and finally translated into the target language.

As can be seen in Figure 1, causalization of the model's equations depends on the model being completely instantiated. Obviously, there is no method to work around this dependency. Therefore, the only way to achieve real separate compilation is to move the code generation stage **in front** of the instantiation. This means that we are not going to generate code from the DAE but actually code that itself can generate the DAE with some help from the runtime system. Although this design has been in use with the Mosilab [10] compiler, it has neither been used for separate compilation nor formally specified. The closest thing to a specification of this method (although not implemented for Modelica) would be the Modeling Kernel Language semantics by David Broman [2].

The runtime instantiation implies that there is no need to translate flattened models (since the generation of the DAE can be done in the correct order). Also there is no need to generate all equations directly at compile-time. Instead the compiler might generate special functions that can create equations at runtime, which allows a quite natural translation of advanced language features like expandable connectors.

With this method, the output of our compilation process can easily satisfy the above requirements by directly mapping every Modelica source file to a compilation unit of the target language. The compiler of course has to use a naming scheme that allows the generated files to reference each other, but this is a trivial task.

3. Subtypes

In Modelica's type system [3] subtype relations are defined implicitly by the set of fields of a class. For separate compilation this means that (contrary to e.g. Java or C++) there is actually no need for a class B to know anything about a class A to be a subtype of that class. Therefore the compiler output of B must be able to be used with A and vice versa even if both were compiled completely separately.

```
class A                                  1
  output Real x;                         2
 equation                                3
  x = sin(time);                         4
end A;                                   5
class B                                  6
  Real y;                                7
  output Real x;                         8
 equation                                9
  y = 23.0;                              10
  x = 42.0;                              11
end B;                                   12
class Foo                                13
  replaceable A a;                       14
  output Real x;                         15
 equation                                16
  connect(a.x, x);                       17
  ...                                    18
end Foo;                                 19
class Bar                                20
  Foo foo1;                              21
  Foo foo2(redeclare B a);               22
end Bar;                                 23
```

Listing 1: Example for redeclaration.

In the listing of Listing 1 the instantiation of `foo2` is modified with a so called *redeclaration*, a construct that allows the enclosing class to replace an element of its child with an element of any compatible type (in other words: a subtype). Since both `Foo` and `B` may already have been compiled earlier, there is no way to modify one of the compiled classes to be type compatible to the redeclaration.

The same problems can arise from the use of the **extends** declaration:

```
class Baz                                   1
  extends Foo(redeclare B a);               2
end Baz;                                    3
```

In the above example we assume that code generation does not compile code from `Foo` into `Baz` (which would violate the first requirement of separate compilation, since compiling `Foo` would become useless). Therefore, if `Foo` is already compiled, all code generated from its equations has to be made type compatible with the new type of the field `a` *prior* to knowing how that new type actually looks like. If the target language has a nominal type system [1] (a rather common case), this is actually impossible without compiling the complete source code at once [5] — which is clearly no option in our case.

While we cannot change the code itself due to separate compilation requirements, we can build a bridge between the use and declaration site by means of coercion semantics.

3.1 Coercion semantics

Coercion semantics [11] is an implementation technique for languages with subtyping that translates a program with subtyping into a simpler one without subtyping. The general idea is as follows:

Whenever a given program context Γ requires an object a of class A any object b of a subclass B of A may also be used (context criterion). Since the runtime representation of b and a are different (usually apparent by a different type in the target language), e.g. b might have additional fields, the compiled form of Γ must be able to cope with many different object representations. If we demand that the compiled form of Γ is independent of the object given, i.e. a or b, it must inspect the object at runtime to access its fields correctly. This inspection, unfortunately, has a non-negligible runtime-overhead. We can circumvent this overhead by always passing an object of class a. Therefore, we have to coerce, hence the name coercion semantics, b to a before handing it to Γ. Since B is a subclass of A this coercion is always possible. Experience in other systems [6, 7] shows that the coercion is much cheaper, in terms of performance, than runtime inspection of objects. Applying coercions semantics, the compiled form of Γ has to operate on objects of class A, exclusively.

We illustrate this technique by an example. Class `Bar` in Listing 1 creates an instance of Class `Foo` while redeclaring `Foo`'s field `a` from `A` to `B`. To enable reuse of the code of `Foo` — `Foo` now acting as the context Γ — without expanding `B` into `Foo` and recompiling we coerce an object of class `B` to class `A`. We can describe the effect of this coercion by introducing a temporary class `B'` that is structurally identical to `A` but has `B`'s fields:

```
class B                                  class B'
  Real y;                                  output Real x;
  output Real x;        coerce           equation
equation                  ⟹               x = 42.0;
  x = 42.0;                              end B';
  y = 23.0;
end B;
```

We now redeclare `a` to `B'` instead of `B` and replace line 21 of Listing 1 by

```
Foo foo2(redeclare B' a);
```

This is possible since all information of `B` that is relevant to `Foo` is contained in `B'` and objects of `B'` can be represented in the same way as objects of `A`.

3.2 xModelica

To make use of this technique in Modelica with as little overhead as possible we have to make use of the fact that the object tree of a Modelica model is static. Thus coercions should not contain values (and be updated every time the original value changes), but rather express equivalence between two names. Usually in Modelica this can be expressed by equations, but since those are part of the actual model (which we do not want to change) and would have to be evaluated at runtime, we prefer to introduce the notion of references.

Technically we achieve this by adding some expressive power to Modelica. Inside a compiler this is simply done by providing additional constructs and fields in the abstract syntax tree, which do not have counterparts in the external syntax. But for discussing these issues it is preferrable to present the concepts in concrete syntax. Therefore we augment Modelica by some notations that allow us to express the additional concepts, thus obtaining an extended language *xModelica*. (Keep in mind, xModelica is not available to users, it is just a "prettyprinting" of the internal abstract syntax trees.) The following changes are made to transform a Modelica program into its xModelica representation:

1. Introduce type modifiers **{indirect, direct}**. Modifiers of this kind are well known in languages; for example, Java has a modifier set `{public, protected, standard, private}` (of which `standard` is invisible, that is, represented by writing nothing) or the modifier set `{final, nonfinal}` (of which the second one is again invisible). Modelica itself does the same with **inner** or **outer**.
2. A function **indirect** is added that creates an instance of an indirect type from a direct variable. This function is defined for every type but does not really need an implementation. In fact, this function could be seen as an explicit coercion itself (explicit because direct types are not subtypes of indirect types).
3. This modifier is an actual part of the type, that is, the type **direct** `Real` is different from the type **indirect** `Real`. But to simplify the usage of indirect types, we consider them being subtypes of their respective direct counterpart.

[1] Even if the generated code has no type system at all, at least the memory layout of objects would have to be made compatible.

4. The default type declaration is **direct**. Indirect types will only be introduced by some transformations. The distinction between indirect and direct types allows the clear usage of object references (which are not part of Modelica): Every instance of an indirect type can be seen as the reference to a direct type. This interpretation gives a natural meaning of the function **indirect** which simply returns a new reference to an already existing direct typed object. Note, that instead of explicitly defining a function **direct** that would do the opposite, we kept the design simple by using the subtype relation mentioned above (which has the same natural interpretation). Also we intentionally did not introduce something like references of references (we do not need that level of complexity until now).
5. The instantiation of a class can now have indirect parameters (declared after the name of the class). Since they are a special kind of parameter, we denote them by the special brackets $\langle \ldots \rangle$. The usual scoping rules apply to those parameters as well as the ability to use default parameters. The actual indirect parameters must be given at object instantiation.
   Note: In a clean language design, one could argue that the indirect arguments should be added to the class, not to objects. But we only want an external representation of the internal data structures. Hence, we properly reflect the fact that the additional field is indeed added to the object, not to the class.

```
record A* ⟨indirect output Real x⟩                                  1
end A*;                                                             2
class A                                                             3
  direct output Real x;                                             4
  ...                                                               5
end A;                                                              6
class Foo⟨indirect A a = indirect(A())⟩                             7
  direct output Real x;                                             8
  ...                                                               9
end Foo;                                                            10
function C_{B →A}                                                   11
  input indirect B b;                                               12
  output direct A* a⟨x = indirect(b.x)⟩();                          13
end C_{B →A};                                                       14
class Bar                                                           15
  Foo foo1;                                                         16
  B a;                                                              17
  A* a' = C_{B →A}(indirect(a));                                    18
  Foo foo2⟨indirect(a')⟩();                                         19
end Bar;                                                            20
```

Listing 2: Example of Listing 1 converted to xModelica

In our example of Listing 1 a coercion can then be expressed as shown in Listing 2. As can be seen the coercion transformation consists of several parts:

1. A class `A*` is introduced that contains an **indirect** field for each field of `A`. This class is by definition always a subtype of the original class (and can thus safely be used instead of the original). Note that it can be created while compiling the definition of `A`.
2. Every replaceable field of `Foo` is moved up to the indirect parameters list as the default definition of a indirect parameter of the same name. Again the resulting class is a subtype of the original class.
3. A coercion function $C_{B \to A}$ `: B` $\to$ `A*` is created. This function lifts every field of `B` into an indirect field of the same name in a instance of `A*`.
4. Finally the coercion function is applied to a local instance of `B` and its output is fed into the instatiation of `Foo`.

With this method the redeclaration statements can be transformed into a much more common concept. Therefore the translation into object oriented (or even functional) code is now much simpler. The new interface of instantiation ensures that no dependencies from redeclaration statements are left in the generated code and thereby make separate compilation possible.

3.3 Subtyping in functions

Subtyping in Modelica is not restricted to models and records but also covers functions. Unfortunately, the Modelica Specification (version 3.2 [1]) is unclear about when one function is a subtype of another: Since Functions are special classes in Modelica (and their parameters special fields), the subtyping rule for functions just references the rule for classes which makes no difference between **output** and **input** fields. Thus the principle of contravariance in function parameters [4] is violated. Since we want to focus on separate compilation, we will assume that this problem has been solved, by redefining the rules for subtypes of functions in the Modelica Specification.

```
function foo                                                        1
  input Bar bar;                                                    2
  output Baz bar;                                                   3
  ...                                                               4
end foo;                                                            5
...                                                                 6
MyBaz baz; //subtype of Baz                                         7
MyBar bar; //subtype of Bar                                         8
algorithm                                                           9
...                                                                 10
baz := foo(bar);                                                    11
```

Listing 3: Example for function application.

Function application with subtypes is pretty straightforward due to our coercion routine: The Modelica fragment in Listing 3 can easily be transformed into its corresponding xModelica fragment:

$C_{Baz \to MyBaz}$`(foo(`$C_{MyBar \to Bar}$`(bar)))`

The application of (subtype) functions is a little bit more complicated but still no big problem, as long as the above mentioned error is corrected. With the principle of contravariance a function $f : A \to B$ is a subtype of $g : C \to D$ if, for the types A, B, C, D, $A \subseteq C$ and $D \subseteq B$ holds true.

```
class C;                                          1
class A extends C;                                2
class B;                                          3
class D extends B;                                4
function F                                        5
  input A a;                                      6
  output B b;                                     7
end F;                                            8
function G                                        9
  input C a;                                     10
  input D b;                                     11
end G;                                           12
function H                                       13
  input function F f;                            14
  input  A a;                                    15
  output B b = f(bar);                           16
  ...                                            17
end H;                                           18
...                                              19
function G g //subtype of F;...                  20
B b := H(g, bar);                                21
```

Listing 4: Example for function contravariance.

The problem here is that while parameter coercion can take place just at the location of the function call, this is not possible for function parameters (simply because the possible type of the function is again unknown). Therefore the function parameter itself must be subject to coercion but not the function application. Since we did not introduce a xModelica notation for function parameters, we will use a mathematical notation here. The parameter g of `foo` in line 21 will be replaced by $C_{D\to B} \circ g \circ C_{A\to C}$. We'll leave it open on how to implement function composition but since function parameters are allowed in Modelica some kind of functional object will be needed in the runtime system anyway. The step to a higher order function is not too big from there.

4. Dynamic Binding (inner/outer)

One of the Modelica language features that causes trouble among users is the inner/outer pairing. Some people consider it as being mandatory for reasons of practical usability in certain application scenarios, others are deterred by the incomprehensibility and error-proneness, which the feature exhibits in particular in slightly more intricate constellations.

What are the reasons for these contradictory viewpoints? As usual a clarification of such seeming ad-hoc phenomena can be obtained by mapping them to classical concepts of programming language theory. Then it is immediately seen that we simply encounter the standard dichotomy between *static and dynamic binding* of variables. And – as usual – problems arise, whenever two concepts (even though each of them may be clean and clear in isolation) are mixed in some odd fashion.

As usual, the *scope* of a name `x` is the textual region of the program text, where it is known; let us denote that

```
class A                                           1
  Real x;                                         2
  class A1                                        3
    ... use x ...                                 4
  end A1;                                         5
  class A2                                        6
    Real x;                                       7
    ... use x ...                                 8
  end A1;                                         9
  ...                                            10
end A;                                           11
```

Listing 5: Simple scoping

region as $\texttt{scope}_{\texttt{x}}$. And it is also standard knowledge that such scopes can contain *holes* due to declarations of the same name inside the scope. As a consequence, any point in the program has for each name `x` a unique scope $\texttt{scope}_{\texttt{x}}$. In Listing 5, the scope of the variable `x` in line 2 is the whole region of the class `A` with the exception of class `A2`; hence, $\texttt{scope}_{\texttt{x}_2} = \texttt{[1..5]} \cup \texttt{[10..11]}$. Analogously, the scope of the variable `x` on line 7 is the region of class `A2`, that is, $\texttt{scope}_{\texttt{x}_7} = \texttt{[6..9]}$.

```
{ int a=0;                                        1
 fun f(int x) = a * x;                            2
 { int a = 2;                                     3
  f(3)                                            4
  ...                                             5
 }                                                6
}                                                 7
```

Listing 6: Different binding example.

Static binding states that for an applied occurrence of a name `x` at some program point p the corresponding declaration is directly given by the scope $\texttt{scope}_{\texttt{x}}$, in which p lies. In the above example, the application of `x` in line 4 refers to the declaration of `x` in line 2, since $4 \in \texttt{scope}_{\texttt{x}_2}$ and analogously `x` in line 8 refers to the declaration in line 7, since $8 \in \texttt{scope}_{\texttt{x}_7}$.

Dynamic binding uses a more complex principle for the association between an applied occurrence of a name `x` and its corresponding declaration – at least for local applications in functions, classes etc. Now we don't use the scope of the point, where `x` is applied, but the scope of the point, where the function, the class etc. is applied.

Listing 6 illustrates dynamic binding in some fictitious λ-style language, thus demonstrating that the concept is long known in many languages.

Under static binding the non-local name `a` in line 2 would refer to the declaration in line 1 such that the call `f(3)` would yield the value 0. But under *dynamic binding* the application of the non-local name `a` would refer to the declaration of `a` that is valid in line 4, that is, to the declaration in line 3. Hence the call `f(3)` yields the value 6.

```
class A                                   1
  outer Real x;                           2
end A;                                    3
class E                                   4
  inner Real x;                           5
  class F                                 6
    inner Real x;                         7
    class G                               8
      Real x;                             9
      class H                            10
        A a;                             11
      end H;                             12
      H h;                               13
    end G;                               14
    G g;                                 15
  end F;                                 16
  F f;                                   17
end E;                                   18
class I                                  19
  inner Real x;                          20
  E e;                                   21
  A a;                                   22
end I;                                   23
```

```
class A⟨indirect Real x⟩                  1
                                          2
end A;                                    3
class E                                   4
  indirect Real x;                        5
  class F                                 6
    indirect Real x;                      7
    class G⟨indirect Real dx⟩             8
      direct Real x;                      9
      class H⟨indirect Real dx⟩          10
        direct A a⟨dx⟩;                  11
      end H;                             12
      H h⟨dx⟩;                           13
    end G;                               14
    G g⟨x⟩;                              15
  end F;                                 16
  F f;                                   17
end E;                                   18
class I                                  19
  indirect Real x;                       20
  E e;                                   21
  A a⟨x⟩;                                22
end I;                                   23
```

Listing 7: Complex example from section 5.4 of the Modelica Specification.

```
class A                                   1
  outer Real x;                           2
  ... use x ...                           3
end A;                                    4
class B                                   5
  inner Real x;                           6
  A a1, a2;                               7
  ... use a1.x ... a2.x ...               8
end B;                                    9
```

Listing 8: Simple example for inner/outer.

```
class A⟨indirect Real x⟩                  1
  ... use x ...                           2
end A;                                    3
class B                                   4
  indirect Real x;                        5
  direct A a1⟨x⟩, a2⟨x⟩;                  6
  ... use a1.x ... a2.x ...               7
end B;                                    8
```

Listing 9: inner/outer removed.

This kind of binding is well known from a number of functional languages, from object-oriented languages such as Java (in the form of method binding with superclasses), but also from typesetting languages such as TEX.

In Modelica it takes the syntactic form of **inner**/**outer** pairs. The simplest instance is illustrated by the example given in Listing 8 (adapted from Section 5.4 of the Modelica 3.1 reference).

Here `B.x` ≡ `B.a.x` ≡ `A.x` holds, that is, all three names are the same. Except for the possibly aberrant syntax, this looks fairly simple. However – as we will see in a moment – there are much more intricate scenarios, where the correct associations are not so easily seen.

In the class `I` in the complex example from Listing 7 (section 5.4, due to space concerns only parts of the example are shown) we have, among others the following equivalences: `e.f.x` ≡ `e.f.g.h.a.x` (lines 22+8 and lines 22+8+12) or `a.x` ≡ `x` (lines 23+1 and line 21). By contrast, other applications are different, for example `e.x` ≢ `e.f.x` (lines 22+6 and 22+8) or `e.f.g.x` ≢ `e.f.g.h.x` (lines 9+10+14 and line 11).

The root of the problem is that we encounter an isolated occurrence of dynamic binding in an otherwise statically binding language. It is this clash of paradigms that makes things both hard to digest and hard to implement. Therefore the obvious solution is to transform the dynamic bindings into static bindings. This will be sketched in the following.

We will use the code from Listing 9 to illustrate the augmented concepts. As is illustrated by this example, our transformation consists of four parts:

1. Both **outer** and **inner** declarations are converted to indirect types.
2. An outer declaration is converted into a parameter of the corresponding class. This new parameter has (in contrast to replaceable fields) no default value. Therefore this class becomes what in other languages is called abstract, since it cannot be instantiated without that parameter.
3. Finally we add to each instantiation of the abstract class a corresponding argument. If the current class has no indirect field, that class is made abstract too and the parameter is pulled up.

```
model CI                                          1
  outer Boolean b;                                2
  Real x(start=1);                                3
equation                                          4
  der(x) = if b then ?x else 0;                   5
end CI;                                           6
model Sub                                         7
  Boolean c = time<=1;                            8
  inner outer Boolean b = b and c;                9
  CI i1;                                          10
  CI i2;                                          11
end SubSystem;                                    12
model System                                      13
  Sub s;                                          14
  inner Boolean b = time>=0.5;                    15
  // s.i1.b will be b and s.c                     16
end System;                                       17
```

```
model CI⟨indirect Boolean b⟩                      1
                                                  2
  Real x(start=1);                                3
equation                                          4
  der(x) = if b then x else 0;                    5
end CI;                                           6
model Sub⟨indirect Boolean db⟩                    7
  Boolean c = time =< 1;                          8
  indirect Boolean b = db and c;                  9
  CI i1⟨b⟩;                                       10
  CI i2⟨b⟩;                                       11
end SubSystem;                                    12
model System                                      13
  Sub s⟨b⟩;                                       14
  indirect Boolean b = time>=0.5;                 15
  // s.i1.b will be b and s.c                     16
end System;                                       17
```

Listing 10: Simultaneous inner/outer.

4. If necessary (e.g. because an element with the same name already exists) the newly introduced parameter is renamed.

Let us briefly look at the typing issue. We again take an example from Section 5.4 of the Modelica 3.1 specification (Listing 11). By the standard rules of typing under static binding it is clear that the argument `x` that would be given to the instantiation in line 7 has type **indirect** `Real`, whereas the requested type in line 3 is **indirect** `Integer`. This shows that the transformation naturally preserves the requirement that a inner declaration shall be a subtype of all corresponding outer declarations.

```
class A                                           1
 inner Real x; //error                            2
 class B                                          3
  outer Integer x;                                4
  ...                                             5
 end B;                                           6
 B b;                                             7
end A;                                            8
```

Listing 11: Type error in inner/outer usage.

```
partial function A                                1
  input Real u;                                   2
  output Real y;                                  3
end A;                                            4
function B                                        5
  extends A;                                      6
algorithm                                         7
  ...                                             8
end B;                                            9
class D                                           10
  outer function fc = A;                          11
  ...                                             12
equation                                          13
  y = fc(u);                                      14
end D;                                            15
class C                                           16
  inner function fc = B;                          17
  D d;                    // equation now y = B(u)  18
end C;                                            19
```

Listing 12: inner/outer with functions.

Finally, there is the odd case of allowing both inner and outer modifiers simultaneously. According to the Modelica 3.1 specification this "conceptually introduces two declarations with the same name: one that follow the above rules for inner and another that follow the rules for outer." With this advise our transformation works straightforward for that case too: When a direct field of the same name already exists, the indirect parameter is renamed. The example from Listing 10 shows how the informal interpretation of simultaneous inner/outer declarations fits naturally in our transformation.

Also functions are an interesting issue (taking the example from the specification shown in Listing 12). Listing 13 shows that this case can be handled easily too, if we also allow functions as indirect parameters. Since (direct) function parameters are allowed in the latest Modelica specification and there is no conceptual difference between direct and indirect function parameters, doing so does not introduce any complexity. The only inconvenience results from the fact that functions are not first class citizens of Modelica. Therefore a (useless) instantiation has to be made. But since our target language is object oriented, this would probably have to happen anyway (in a functional language the instantiation would just be a renaming).

What does this achieve? The answer is simple: We eliminated dynamic binding! In other words, we now have a uniform system of static bindings (and thus a uniform compiler design without complex exceptions) but still retain all effects of Modelica's dynamic inner/outer binding in a semantically correct fashion.

```
partial function A                                          1
  input Real u;                                             2
  output Real y;                                            3
end A;                                                      4
function B                                                  5
  extends A;                                                6
algorithm                                                   7
  ...                                                       8
end B;                                                      9
class D⟨indirect function A fc⟩                             10
                                                            11
  ...                                                       12
equation                                                    13
  y = fc(u);                                                14
end D;                                                      15
class C                                                     16
  indirect function B fc;                                   17
  D d⟨fc⟩;                    // equation now y = B(u)      18
end C;                                                      19
```

Listing 13: Listing 12 translated to xModelica.

This also applies to the important issue of *separate compilation*. We now only implement the classical static-binding variant of separate compilation and obtain a correct treatment of Modelica's inner/outer binding for free.

5. Drawbacks and Costs of Separate Compilation

Although separate compilation is a worthy goal, there are of course some drawbacks that occur from our design:

1. By creating code that does not contain a DAE but only tries to create one, we lose the ability to check for certain model properties. The most important of those is probably the requirement of the model being balanced. Since we do not know how compiled models are actually used, it is at least hard, if not impossible, to know if a model is balanced (if, e. g. only an interface is shipped with pre-compiled code). This is a general problem with separate compilation and usually solved by having the linker checking the global assertions. Since there is no Modelica linker, currently the only solution left is to move these checks into the runtime as well.
2. A more subtle effect of separate compilation is the compatibility of generated code. If module `A` is compiled with a more recent compiler than module `B`, there is no guaranty that both work together as expected, since small changes in the compiler may render both modules incompatible. The only way to workaround this issue is either to not change the output format at all (rather unrealistic) or to have some kind of versioning that hinders the user from plugging together incompatible compilation units.
3. The probably most far-reaching restriction that results from our approach is the impossibility to run optimizations at compile-time. For Modelica this means that there can be no causalization or index reduction done by the compiler. Again this problem depends on the absence of a dedicated Modelica linker. Thus both these operations have to be run before the actual simulation starts. Luckily this is generally possible and even necessary if one adds model structural dynamics to Modelica [13] (which is e. g. already present in the Mosilab compiler [10]).

6. Conclusion

With the usage of separate compilation a Modelica compiler comes closer to what state-of-the art tools can offer a software engineer. This applies to the overall compilation effort as well as to the opportunity to ship pre-compiled libraries with a clean interface.

We have shown that the most complex features of the Modelica language can in fact be transformed into an object oriented language by only using features that are much more common. This enables the creation of Modelica compilers for many target languages. The used translation scheme gives a new way of handling constructs with special semantics like stream connectors by simply moving those semantics into the runtime system.

While the language is thereby ready for separate compilation, some problems still remain open for future work. The most important is the conciliation of separate compilation with causalization. The same applies for symbolic manipulations. We will investigate both topics in the future.

References

[1] The Modelica Association. Modelica - a unified object-oriented language for physical systems modeling, 2010.

[2] David Broman. Flow lambda calculus for declarative physical connection semantics. Technical Report 1, Linköping University, PELAB - Programming Environment Laboratory, The Institute of Technology, 2007.

[3] David Broman, Peter Fritzson, and Sébastien Furic. Types in the modelica language. In *Proceedings of the Fifth International Modelica Conference*, 2006.

[4] Luca Cardelli. A semantics of multiple inheritance. In *Proc. of the international symposium on Semantics of data types*, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[5] Gilles Dubochet and Martin Odersky. Compiling structural types on the jvm: a comparison of reflective and generative techniques from scala's perspective. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 34–41, New York, NY, USA, 2009. ACM.

[6] Christopher League, Zhong Shao, and Valery Trifonov. Representing java classes in a typed intermediate language. *SIGPLAN Not.*, 34(9):183–196, 1999.

[7] Christopher League, Zhong Shao, and Valery Trifonov. Type-preserving compilation of featherweight java. *ACM Trans. Program. Lang. Syst.*, 24(2):112–152, 2002.

[8] Ramine Nikoukhah. Extensions to modelica for efficient code generation and separate compilation. In *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools*, Linköping Electronic Conference Proceedings, pages 49–59. Linköping University Electronic Press, Linköpings universitet, 2007.

[9] Ramine Nikoukhah and Sébastien Furic. Towards a full integration of modelica models in the scicos environment. *Proceedings of the 7th International Modelica Conference*, 43(74):631–645, 2009.

[10] Christoph Nytsch-Geusen and Thilo et al Ernst. Mosilab: Development of a modelica based generic simulation tool supporting model structural dynamics. In Gerhard Schmitz, editor, *Proceedings of the 4th International Modelica Conference, Hamburg, March 7-8, 2005*, pages 527–535. TU Hamburg-Harburg, 2005.

[11] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[12] 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.

[13] Dirk Zimmer. *Equation-based Modeling of Variable-structure Systems*. PhD thesis, ETH Zürich, 2010.

Import of distributed parameter models into lumped parameter model libraries for the example of linearly deformable solid bodies

Tobias Zaiczek Olaf Enge-Rosenblatt

Fraunhofer Institute for Integrated Circuits, Design Automation Division, Dresden, Germany,
{Tobias.Zaiczek,Olaf.Enge}@eas.iis.fraunhofer.de

Abstract

Modelling of heterogeneous systems is always a trade-off between model complexity and accuracy. Most libraries of object-oriented, equation-based, multi-physical simulation tools are based on lumped parameter description models. However, there are different ways of including spatial dependency of certain variables in the model. One way that might be quite difficult is to manually discretize the model into an interconnection of lumped parameter models. This approach can get very time-consuming and is always sensitive to modelling or identification errors.

To avoid these issues, we try to take advantage of the well-established methods for automatically discretizing a distributed parameter model for example by means of Finite Element methods. However, to achieve a suffiently good approximation, these methods very often result in large-scale dynamic systems that can not be handled within equation-based simulators. To overcome this drawback there exist different approaches within the literature.

On the basis of deformable mechanical structures, one way of including distributed parameter models into libraries of lumped parameter models for the purpose of common simulation is pointed out in the present paper. For the implementation of the resulting models the authors take advantage of equation-based modelling libraries as new models can here easily be integrated.

Keywords distribuded parameter systems, FEM import, mechanical systems, deformable bodies

1. Introduction

Simulation of physical heterogeneous systems is getting more and more important during the design process of technical systems. Anyway, the simulation of such systems is not an easy task. Due to the different domains of physical laws interacting with each other, accurate models may tend to get very complex. One fundamental principle of modelling is therefore the hierarchical decomposition and interconnection of physical systems. This can be done according to the physical domain (e. g. mechanical and electrical part), according to common physical behaviour (e. g. solid bodies), or according to the interaction and dependancies of physical laws.

The common objectives of these different classifications are the reduction of the model complexity for each part, the achievement of modularity and exchangeability, the increase of reusability of the models, and the enhancement of their understanding. Appropriate assumptions on the complexity can thus be made for every part and the model can be seperately described by its physical laws. Of course, in general, the decomposed parts, the so-called subsystems of the model, interact with each other. These interactions can only be expressed via certain finite sets of variables, the so called interconnectors.

One very common assumption on submodels is the description of the physical system as a lumped parameter system. Here, all variables are assumed to be a function of time without any spatial dependancy. For such a system one ends up with a system of algebraic and ordinary differential equations, a so called DAE. This assumption is made in many libraries of object-oriented, equation-based, multi-physical simulation tools.

Nevertheless, a lumped parameter description is not always suitable to describe certain effects of physical systems accurately.

Distributed parameter models are characterized by the fact that all variables are regarded not only as functions of time but also as functions of some spatial coordinates. Hence, the set of independent variables increases to more than one variable. This type of models can be characterized in integral or differential form with appropriate initial and boundary conditions. The differential formulation results in a system of partial differential equations (PDEs).

In our paper, we only regard linear inhomogeneous systems of PDEs that can be written down as

$$\mathcal{D}_t \boldsymbol{u} + \mathcal{R}\boldsymbol{u} = \boldsymbol{w} \tag{1}$$

with dependent variables $\boldsymbol{u}$, independent variables time t and position (e. g. x, y, and z), and a dependent source term $\boldsymbol{w}$. $\mathcal{D}_t$ and $\mathcal{R}$ denote appropriate linear operators with respect to (w. r. t.) time and w. r. t. all spatial coordinates, respectively.

3rd International Workshop on Equation-Based Object-Oriented Languages and Tools. October, 2010, Oslo, Norway.

This type of models appears in several, different domains of application as a quite natural description of the physical behaviour. The following three examples should illustrate this fact.

Example 1: *Heat flow*
The equations for the heat flow within a homogeneous structure can be described by the following PDE:

$$\frac{\partial u}{\partial t} - \alpha \Delta u = \frac{\partial u}{\partial t} - \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) = q$$

where u is the temperature and q is the heat source density, each a function depending on the time t as well as the spatial coordinates x, y, and z. ∎

Example 2: *Electrical transmission line*
An electrical transmission line can be modelled in terms of two variables i and u both depending on time t and the length z as:

$$\frac{\partial u}{\partial z} + L' \frac{\partial i}{\partial t} + R' i = 0 \qquad \frac{\partial i}{\partial z} + C' \frac{\partial u}{\partial t} + G' u = 0.$$

The quantities R', L', G', and C' are parameters of the model and describe the resistance and the inductance load of the transmission line, the conductance and the capacitance load between the lines, each per unit length, respectively. ∎

Example 3: *Structural mechanics*
When analysing the behaviour of solid bodies under static or dynamic forces, we can use the theory of structural mechanics to get a linearized model in terms of the displacement $\boldsymbol{u}$ from the undeformed position and the volume force density $\boldsymbol{k}_0$ as

$$\frac{\partial^2 \rho_0 \boldsymbol{u}}{\partial t^2} = \boldsymbol{k}_0 + \mathrm{Div}(\boldsymbol{H}\, \mathrm{Grad}\, \boldsymbol{u}).$$

In this equation the operators $\mathrm{Div}(\cdot)$ and $\mathrm{Grad}(\cdot)$ are defined by

$$\mathrm{Grad}(\boldsymbol{u}) \equiv \frac{1}{2} \left(\partial_i u_j + \partial_j u_i \right)_{i,j=1,2,3}$$
$$\mathrm{Div}(\boldsymbol{S}) \equiv \left(\sum_{j=1}^{3} \partial_j S_{ij} \right)_{i=1,2,3}$$

and the quantities ρ_0 and $\boldsymbol{H}$ denote the mass density and the symmetric stiffness tensor resulting from the material properties, respectively. ∎

In these examples no care has been taken of the initial and boundary values, that of course influence the solvability as well as the solution of the problem. One can very often assume to have initial conditions for the independent variable t and boundary conditions of appropriate type for all spatial independent variables x, y, and z.

For the simulation of such distributed parameter models, there exist different methods, that automatically discretize the models (see e. g. [2, 11]). Very often these algorithms result in large scale dynamic systems that cause a high order of complexity. In any case, it is desirable to include such models into equation-based simulators (as e. g. Dymola). To this end, two different but sometimes complementing approaches have been established [17]. The first approach tries to combine the models in one simulator. The second approach aims to use different spezialized, well adapted simulators for each domain and tries to link these simulators for all necessary interactions [5, 13, 19]. While both ways have their own advantages and drawbacks ([14, 17]), this paper will focus on the first approach.

There have already been different authors attending the import of PDEs into Modelica (as e. g. in [12]). Anyway, in difference to other papers, our paper does not aim to directly include PDEs into the modeling language Modelica. Our focus is given to the necessary preprocessing of PDEs in general and for the import of flexible bodies into the multi-body library.

The first part of the paper covers the detailed discussion of the general approach of including distributed parameter models. Afterwards, in section three this approach is applied to the import of mechanical Finite Element discretized models into a classical multi-body library. Here, all issues of section two are picked up and explained for the concrete example. Section four presents some examples for the foregoing work flow in order to validate the generated models. In section five, an outlook is given while section six summarizes the content of this paper.

2. General considerations for the import of distributed parameter systems

The import of distributed parameter models requires the definition of an appropriate interface to lumped parameter simulation libraries. For the sake of exchangeability, this interface is supposed to be compatible to the connectors of the other library elements. The issue of creating such an interface for the distributed parameter model is discussed in subsections 2.1 and 2.3.

Another question that arises for the import of distributed parameter models concerns the embedding regarding a numerical method to solve this kind of problems, as equation-based simulation tools in general do not have solvers for PDEs. Anyway, for the numerical solution of distributed parameter models there already exist serveral different algorithms. They all have one property in common: they try to solve the simulation task in a finite-dimensional solution space. The process of deriving a finite-dimensional model is also called discretization. In fact, one could distinguish between the discretization in terms of the time t and the discretization in terms of the spatial independent coordinates x, y, and z. Since for the spatial discretization there exist already many elaborate numerical tools, our starting point will be the spatially discretized system rather than the original distributed parameter model. This topic is treated in subsection 2.2.

However, there is a difficulty arising from the spatial discretization. The discretized models become generally very large in scale and are therefore often intractable for equation-based solvers. Subsection 2.4 is dedicated to this issue.

2.1 Connectors of the distributed parameter model

This subsection covers the definition of an appropriate interface to the lumped parameter simulation library as it is essential for the import of distributed parameter models. For the unobstructed integration and exchangeability of the imported models it is necessary to design the interface in

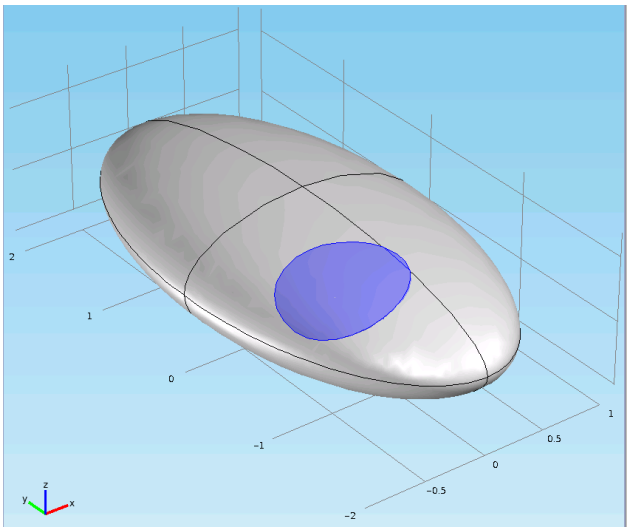

Figure 1. Connector definition for the distributed parameter model

compliance with the existing connector classes of the library. A connector class in an object-oriented simulation tool is a class that defines the set of variables that are necessary for the interconnection of the elements in this library. In a mechanical multi-body library for example, these connector classes are often called flanges and include all necessary kinematic and kinetic variables. The connectors of an electrical circuit library often consist of the two quantities current and voltage, while for a heat flow library the connectors typically contain the two variables heat flow and temperature.

In this paper, we present an approach of creating such an interface within the distributed parameter model. Figure 1 gives an illustration for the explanations below. As a first step, we assume to have a connector of the lumped parameter model library with a defined set $\mathcal{C}$ of variables. Furthermore, let $\mathcal{V}$ be the domain of all dependent quantities of the distributed parameter model. In structural mechanics for example, we could identify $\mathcal{V}$ with the set of all points of the considered structure. In Figure 1 the gray and blue colourized set $\mathcal{V}$ has been chosen to be an ellipsoid.

Then, it seems very meaningful to consider a connector of the distributed parameter system as a (perhaps lower dimensional) subset $\mathcal{V}_C$ of the domain $\mathcal{V}$ that satisfies the two following constraints (see also Figure 1: the blue coloured shape).

1. There exist two (disjoint) subsets $\mathcal{U}^m \subset \mathcal{C}$ and $\mathcal{W}^m \subset \mathcal{C}$ that are minimal sets of variables in order to uniquely determine the behaviour of all variables in $\boldsymbol{u}$ and $\boldsymbol{w}$ belonging to elements of the subset $\mathcal{V}_C$ of the structure and
2. the values of all variables in $\mathcal{U}^m$ and $\mathcal{W}^m$ are uniquely defined by a "valid" spatial characteristic of all dependent variables in $\boldsymbol{u}$ and $\boldsymbol{w}$ belonging to elements of the subset $\mathcal{V}_C$.

Now, let $\boldsymbol{\xi}^m$ be the column vector of all variables in $\mathcal{U}^m$ and $\boldsymbol{\eta}^m$ the column vector of all variables in $\mathcal{W}^m$. As we will see later on, these conditions can be expressed as some constraint equations between the variables $\boldsymbol{\xi}^m$ and $\boldsymbol{u}$ within $\mathcal{V}_C$ as well as between the variables $\boldsymbol{\eta}^m$ and $\boldsymbol{w}$ within $\mathcal{V}_C$[1]. Thus, we will denote all possible values of $\boldsymbol{u}$ and $\boldsymbol{w}$ within $\mathcal{V}_C$ as "valid" if they satisfy the constraint equations.

2.2 Discretization of the model

As already noticed, it seems very convenient to start from the spatially discretized model rather than from the original PDEs. The reason is, that for the spatial discretization there already exist sophisticated tools that might use different methods as for example

- FDM (Finite Difference method),
- FEM (Finite Element method),
- FVM (Finite Volume method), or
- BEM (Boundary Element method).

For all models of this paper, only the Finite Element method has been used to discretize the model. Since we only regard linear inhomogeneous systems with maximal second order of derivatives w. r. t. time, we can already write the spatially discretized model as

$$\boldsymbol{A}_2 \frac{d^2\boldsymbol{\xi}}{dt^2} + \boldsymbol{A}_1 \frac{d\boldsymbol{\xi}}{dt} + \boldsymbol{A}_0 \boldsymbol{\xi} = \boldsymbol{\eta} \tag{2}$$

with generally time dependent matrices $\boldsymbol{A}_i$ ($i \in \{0, 1, 2\}$) and time dependent vector $\boldsymbol{\eta}$. The vector $\boldsymbol{\xi}$ collects the variables $\boldsymbol{u}$ for every node of the Finite Element mesh, while $\boldsymbol{\eta}$ consists of appropriate spatial integral terms of the variable $\boldsymbol{w}$ for every node.

For the case of mechanical systems we denote $\boldsymbol{A}_2 = \boldsymbol{M}$ as the mass matrix, $\boldsymbol{A}_1 = \boldsymbol{D}$ as the damping matrix, and $\boldsymbol{A}_0 = \boldsymbol{K}$ as the stiffness matrix of the model. The column vector $\boldsymbol{\eta}$ combines all force components acting on the discretized structure while the column vector $\boldsymbol{\xi}$ covers all dispacement variables of the nodes.

In the spatially discretized version of the heat flow equation the matrix $\boldsymbol{A}_2$ vanishes due to the non-existence of second order time derivatives in the PDEs. Matrices $\boldsymbol{A}_1 = \boldsymbol{C}$ and $-\boldsymbol{A}_0 = \boldsymbol{G}$ can here be interpreted as the heat capacitance matrix and the heat conductance matrix, respectively. The column vector $\boldsymbol{\eta}$ collects the heat source densities for each element of the spatially discretized structure. The dependent vector $\boldsymbol{\xi}$ contains the temperatures of all nodes within the structure.

2.3 Connectors of the discretized model

In subsection 2.1 we already defined a connector for the original distributed parameter model. For the spatially discretized model, we can now adapt this definition. To do this, we define a set Ω_C as the set of all nodes of the body that lie in $\mathcal{V}_C$. In Figure 2, these are all nodes within the blue area. The connector of the discretized model is formed by the dependent variables of all nodes of Ω_C. Hence, the conditions derived in subsection 2.1 can easily be stated.

[1] More precisely, since $\boldsymbol{u}$ and $\boldsymbol{w}$ are functions of all independent variables, we must use the restriction of $\boldsymbol{u}$ and $\boldsymbol{w}$ to the set $\mathcal{T} \times \mathcal{V}_C$, with time $t \in \mathcal{T}$.

Assume the vectors $\boldsymbol{\xi}^s$ and $\boldsymbol{\eta}^s$ to be the column vectors of all coordinates of $\boldsymbol{\xi}$ and $\boldsymbol{\eta}$ that belong to nodes within the subset $\mathcal{V}_C$, respectively. Then, there must exist two injective mappings $\boldsymbol{\varphi}$ and $\boldsymbol{\psi}$ from the set of all values of $\boldsymbol{\xi}^m$ and $\boldsymbol{\eta}^m$ to the set of values of $\boldsymbol{\xi}^s$ and $\boldsymbol{\eta}^s$, respectively. The second condition is then inherently satisfied, if we denote an element of the image of $\boldsymbol{\varphi}$ and $\boldsymbol{\psi}$ as a "valid" spatial characteristic. The mapping $\boldsymbol{\varphi}$ can then also be interpreted as a constraint equation on the differential system of equations (2).

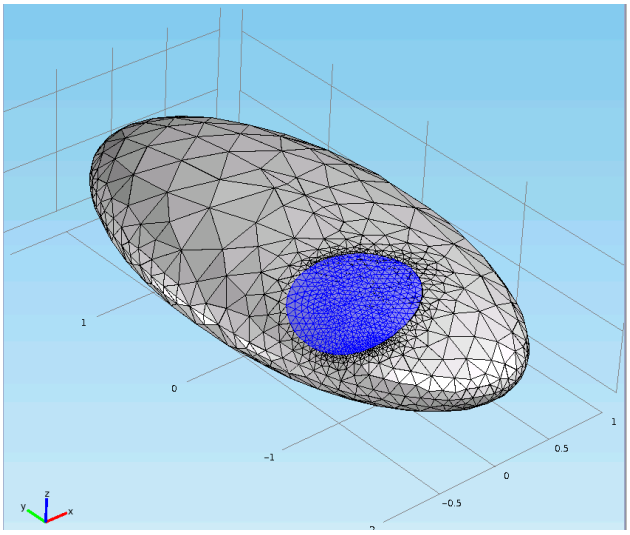

Figure 2. Connector definition for the discretized model

For simplicity, we assume these mappings $\boldsymbol{\varphi}$ and $\boldsymbol{\psi}$ to be linear in $\boldsymbol{\xi}^m$ and $\boldsymbol{\eta}^m$. The constraint equations can thus be stated as

$$\boldsymbol{\xi}^s = \hat{\boldsymbol{\Phi}}\boldsymbol{\xi}^m, \qquad \boldsymbol{\eta}^s = \hat{\boldsymbol{\Psi}}\boldsymbol{\eta}^m \tag{3}$$

with constant full column rank matrices $\hat{\boldsymbol{\Phi}}$ and $\hat{\boldsymbol{\Psi}}$.

In order to take these constraints into account, we first have to define another column vector $\boldsymbol{\xi}^r$ consisting of all the remaining variables of $\boldsymbol{\xi}$ that are neither in $\boldsymbol{\xi}^m$ nor in $\boldsymbol{\xi}^s$.

Then, we can add the variables in $\boldsymbol{\xi}^m$ to the set of dependent variables by defining the vector $\hat{\boldsymbol{\xi}}$ according to

$$\boldsymbol{\xi} \equiv \begin{pmatrix} \boldsymbol{\xi}^r \\ \boldsymbol{\xi}^s \end{pmatrix} = \begin{pmatrix} \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} \end{pmatrix} \begin{pmatrix} \boldsymbol{\xi}^m \\ \boldsymbol{\xi}^r \\ \boldsymbol{\xi}^s \end{pmatrix} \equiv \boldsymbol{\Gamma}\hat{\boldsymbol{\xi}}.$$

Now, we expand all matrices and vectors to

$$\hat{\boldsymbol{A}}_2 = \boldsymbol{\Gamma}^{\mathrm{T}}\boldsymbol{A}_2\boldsymbol{\Gamma}, \qquad \hat{\boldsymbol{A}}_1 = \boldsymbol{\Gamma}^{\mathrm{T}}\boldsymbol{A}_1\boldsymbol{\Gamma},$$
$$\hat{\boldsymbol{A}}_0 = \boldsymbol{\Gamma}^{\mathrm{T}}\boldsymbol{A}_0\boldsymbol{\Gamma}, \qquad \hat{\boldsymbol{\eta}} = \boldsymbol{\Gamma}^{\mathrm{T}}\boldsymbol{\eta}.$$

Note, that the inserted variables $\boldsymbol{\xi}^m$ still do not influence the model except through the constraint equation (3), that can also be written implicitly as

$$\boldsymbol{0} = \begin{pmatrix} \hat{\boldsymbol{\Phi}} & \boldsymbol{0} & -\boldsymbol{I} \end{pmatrix} \hat{\boldsymbol{\xi}} \equiv \boldsymbol{\Xi}\hat{\boldsymbol{\xi}}.$$

Since all variables in $\boldsymbol{\xi}^s$ can be expressed through the variables in $\boldsymbol{\xi}^m$, we can write down the vector $\hat{\boldsymbol{\xi}}$ in terms of the newly defined vector $\bar{\boldsymbol{\xi}}$ as

$$\hat{\boldsymbol{\xi}} = \begin{pmatrix} \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I} \\ \hat{\boldsymbol{\Phi}} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{\xi}^m \\ \boldsymbol{\xi}^r \end{pmatrix} \equiv \breve{\boldsymbol{\Phi}}\bar{\boldsymbol{\xi}}. \tag{4}$$

Applying the Lagrange Multiplier Theorem (see e. g. [7]), equation (2) yields

$$\begin{aligned} \hat{\boldsymbol{A}}_2\ddot{\hat{\boldsymbol{\xi}}} + \hat{\boldsymbol{A}}_1\dot{\hat{\boldsymbol{\xi}}} + \hat{\boldsymbol{A}}_0\hat{\boldsymbol{\xi}} &= \hat{\boldsymbol{\eta}}_e + \boldsymbol{\Xi}^{\mathrm{T}}\boldsymbol{\lambda} + \hat{\boldsymbol{B}}\boldsymbol{\eta}^m \\ \boldsymbol{\Xi}\hat{\boldsymbol{\xi}} &= \boldsymbol{0} \\ \boldsymbol{\xi}^m &= \hat{\boldsymbol{C}}\hat{\boldsymbol{\xi}} = \begin{pmatrix} \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} \end{pmatrix} \hat{\boldsymbol{\xi}}, \end{aligned}$$

where we split up $\hat{\boldsymbol{\eta}}$ into a sum of the external influences $\hat{\boldsymbol{\eta}}_e$ and the influence of the connector variables $\boldsymbol{\eta}^m$ by the matrix $\hat{\boldsymbol{B}}$, that is given by $\hat{\boldsymbol{B}}^{\mathrm{T}} = \begin{pmatrix} \boldsymbol{I} & \boldsymbol{0} & \hat{\boldsymbol{\Psi}}^{\mathrm{T}} \end{pmatrix}$. Furthermore, we added an equation that expresses the connector variables $\boldsymbol{\xi}^m$ in terms of the variables in $\boldsymbol{\xi}$.

Note, that the columns of $\breve{\boldsymbol{\Phi}}$ are orthogonal to the rows of $\boldsymbol{\Xi}$ and thus we can multiply the first equation by $\bar{\boldsymbol{\Phi}}^{\mathrm{T}}$ from the left and replace the vector $\hat{\boldsymbol{\xi}}$ by means of equation (4). Then, the Lagrange Multipliers λ_i in $\boldsymbol{\lambda}$ disappear and hence we get the new equations as

$$\bar{\boldsymbol{A}}_2\ddot{\bar{\boldsymbol{\xi}}} + \bar{\boldsymbol{A}}_1\dot{\bar{\boldsymbol{\xi}}} + \bar{\boldsymbol{A}}_0\bar{\boldsymbol{\xi}} = \bar{\boldsymbol{\eta}}_e + \bar{\boldsymbol{B}}\boldsymbol{\eta}^m \tag{5a}$$
$$\boldsymbol{\xi}^m = \bar{\boldsymbol{C}}\bar{\boldsymbol{\xi}}. \tag{5b}$$

2.4 Model order reduction

The linear system of differential equations (5) is typically large in scale to achieve a good approximation of the constraint PDE for all nodes over a wide range of the frequency domain.

Anyway, equation-based simulation tools apply computer algebra to derive a solvable system of differential algebraic equations. The computational efforts and the memory consumption for these operations increase dramatically for a growing number of equations and variables. Thus, these simulators are generally not able to handle large-scale dynamic systems directly.

However, for many applications it is already sufficient to approximate the behaviour between the variables $\boldsymbol{\eta}^m$ and $\boldsymbol{\xi}^m$ in a relevant range of the frequency domain. Hence, it is preferable and, as stated, often necessary to reduce the size of the system drastically by an appropriate reduction method.

This reduction method should of course take the interesting range of the frequency domain into account. There are many different methods [1, 6] for the linear model order reduction and they all produce a matrix V defining a linear mapping from the set of all reduced variable vectors $\boldsymbol{q}$ to the set of all original vectors $\bar{\boldsymbol{\xi}}$. Consequently, we can write down the relation between those vectors as

$$\bar{\boldsymbol{\xi}} = \boldsymbol{V}\boldsymbol{q}. \tag{6}$$

From a mathematical point of view, this mapping can also be interpreted as a linear constraint on the governing differential equations (5).

Thus, one could apply the same algorithm as for the connector constraint equations and end up with the following equation

$$\boldsymbol{A}_{q,2}\ddot{\boldsymbol{q}} + \boldsymbol{A}_{q,1}\dot{\boldsymbol{q}} + \boldsymbol{A}_{q,0}\boldsymbol{q} = \boldsymbol{\eta}_{e,q} + \boldsymbol{B}_q\boldsymbol{\eta}^m \quad (7a)$$

$$\boldsymbol{\xi}^m = \boldsymbol{C}_q\boldsymbol{q} \quad (7b)$$

where

$$\boldsymbol{A}_{2,q} = \boldsymbol{V}^{\mathrm{T}}\bar{\boldsymbol{A}}_2\boldsymbol{V}, \quad \boldsymbol{A}_{1,q} = \boldsymbol{V}^{\mathrm{T}}\bar{\boldsymbol{A}}_1\boldsymbol{V}, \quad \boldsymbol{A}_{0,q} = \boldsymbol{V}^{\mathrm{T}}\bar{\boldsymbol{A}}_0\boldsymbol{V},$$
$$\boldsymbol{B}_q = \boldsymbol{V}^{\mathrm{T}}\bar{\boldsymbol{B}}, \qquad \boldsymbol{C}_q = \bar{\boldsymbol{C}}\boldsymbol{V}, \qquad \boldsymbol{\eta}_{e,q} = \boldsymbol{V}^{\mathrm{T}}\bar{\boldsymbol{\eta}}_e.$$

3. Import of mechanical structures

As a non-trivial example, an approach of importing models from structural mechanics into a multi-body library is presented in the subsequent subsections. All previously discussed issues will be picked up and explained for this example. Some simulation results for an implementation in Modelica will follow in section 4.

3.1 Introduction

The task of including the dynamic behaviour of deformable bodies into classical multi-body libraries has already been investigated by several authors [3, 15, 18]. Here, a different approach will be presented, as we try to derive the differential equations of motion directly from the parameters of the Finite Element model.

Compared to the work flow presented in the foregoing section, an additional challenge arises for this task. Even though we start from the linearized model for small deformations of the body we have to take into account the nonlinear character of the large motions of the considered body. Hence, we also have to add nonlinear terms to the equations of motion.

Before doing so, some assumptions and simplifications which are used for our approach are listed below.

- Only the linear elastic behaviour of solid bodies is considered. In this paper no beam or plate elements are treated.
- All geometric as well as physical nonlinearities are neglected within the solid body.
- All properties of the solid body are assumed to be constant over time.
- The interconnection points are modelled as rigid bodies, where the joints and bodies of the multi-body library can be rigidly attached (see below).
- The rigid body modes and the mass matrix are considered not to dependent on the deformation of the body.

3.2 The Finite Element model

The starting point of the presented task is the output of a Finite Element simulation tool. Within the tool, the solid body can be described concerning its geometry and its material properties. Using the Finite Element solver a spatially discretized model can be generated that can be written down as

$$\boldsymbol{M}\ddot{\boldsymbol{\xi}} + \boldsymbol{D}\dot{\boldsymbol{\xi}} + \boldsymbol{K}\boldsymbol{\xi} = \boldsymbol{\eta} \quad (8)$$

with the quantities already explained in section 2. This linear system of differential equations describes the elastic behaviour of the body under small deformations and small displacements[2].

For our further calculations we thus have to export the matrices $\boldsymbol{M}$, $\boldsymbol{D}$, and $\boldsymbol{K}$. In addition we need to export the position of all nodes of the undeformed body. For a possible visualization of the body, one could also export some mesh or element information.

3.3 Connector definition

In order to include the model into a multi-body library, one has to define connectors (flanges). These must be compatible to the connectors of the library and thus must include all variables that are defined in the connector class of the library. For multi-body libraries each of the sets of dependent variables $\mathcal{U}^m$ and $\mathcal{W}^m$ consist of at least six elements[3] (directly related to the six degrees of freedom (DoF) of a rigid body), three translational and three rotational elements.

A connector of the spatially discretized model can be composed by considering a subset of nodes, the so-called slave nodes of the flange. The dependent variables $\boldsymbol{\xi}^s$ belonging to the slave nodes can then be experessed in terms of the variables of the connector class $\boldsymbol{\xi}^m$ by a constraint equation, which must satisfy condition 1 and 2 in section 2.1.

As the flange is supposed to be an interconnection element to a rigid body library, it seems very meaningful to use a constraint equation on the slave nodes that rigidly attaches all slave nodes to each other. Hence, all nodes composing a flange can be interpreted as a single rigid body interconnected with the structure. The variables $\boldsymbol{\xi}^m$ provide the position and orientation of that rigid body, that can also be seen as a node with translational and rotational DoFs, the so-called master node. Using geometric linearization one can state the constraint equations for the n_m connectors in a linear way according to

$$\boldsymbol{\xi}_i^s = \hat{\boldsymbol{\Phi}}_i\boldsymbol{\xi}_i^m, \quad i = 1, .., n_m,$$

where $\boldsymbol{\xi}_i^s$ denotes the vector of coordinates of the slave nodes and $\boldsymbol{\xi}_i^m$ the vector of the connector variables belonging to the i-th connector. The constraint matrix constists of three lines for each slave node and is given by

$$\hat{\boldsymbol{\Phi}}_i = \begin{pmatrix} \boldsymbol{I} & \tilde{\boldsymbol{r}}_{c,i} - \tilde{\boldsymbol{r}}_j \end{pmatrix}_j \quad j \in \Omega_i$$

with Ω_i the set of indices of all slave nodes belonging to the connector i, $\boldsymbol{r}_j$ the position vector to the node j, $\tilde{\boldsymbol{r}}_j$ its cross product matrix, and $\tilde{\boldsymbol{r}}_{c,i}$ the cross product matrix of the connector reference point. This connector reference position can be chosen arbitrarily w. r. t. the undeformed shape of the body and is thus part of the designing process.

In order to satisfy condition 2 of subsection 2.1, there are also constraints on the minimal number of slave nodes.

[2] Please note, that it is very important to model the body as a free body within the Finite Element simulation tool, i. e. without any constraints on the undeformed motion of the body.

[3] In many libraries more variables are used for the connector definition in order to avoid singularities and numerical problems.

Every flange must consist of at least three nodes that do not lie in one line in order to uniquely define the orientation of the master.

For further calculations we introduce the following quantities

$$\hat{\boldsymbol{\Phi}} \equiv \begin{pmatrix} \hat{\boldsymbol{\Phi}}_1 & & \boldsymbol{0} \\ & \ddots & \\ \boldsymbol{0} & & \hat{\boldsymbol{\Phi}}_{n_m} \end{pmatrix},$$

$$\boldsymbol{\xi}^s \equiv \begin{pmatrix} \boldsymbol{\xi}^s_1 \\ \vdots \\ \boldsymbol{\xi}^s_{n_m} \end{pmatrix}, \quad \text{and} \quad \boldsymbol{\xi}^m \equiv \begin{pmatrix} \boldsymbol{\xi}^m_1 \\ \vdots \\ \boldsymbol{\xi}^m_{n_m} \end{pmatrix}$$

and we summarize the coordinates of all nodes in $\boldsymbol{\xi}^r$ that are not in $\boldsymbol{\xi}^s$. In addition we assume without loss of generality[4], that the vector $\boldsymbol{\xi}^{\mathrm{T}} = \begin{pmatrix} (\boldsymbol{\xi}^r)^{\mathrm{T}} & (\boldsymbol{\xi}^s)^{\mathrm{T}} \end{pmatrix}$.

Hence, we can express $\boldsymbol{\xi}$ in terms of $\boldsymbol{\xi}^r$ and the connector variables $\boldsymbol{\xi}^m$ by

$$\boldsymbol{\xi} = \begin{pmatrix} \boldsymbol{\xi}^r \\ \boldsymbol{\xi}^s \end{pmatrix} = \begin{pmatrix} \boldsymbol{0} & \boldsymbol{I} \\ \hat{\boldsymbol{\Phi}} & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{\xi}^m \\ \boldsymbol{\xi}^r \end{pmatrix} \equiv \bar{\boldsymbol{\Phi}} \bar{\boldsymbol{\xi}}.$$

Applying the Lagrange Multiplier Theorem and projecting the equations of motion into the achievable subspace, i. e. the image of $\bar{\boldsymbol{\Phi}}$, we get the constraint equations of motion

$$\bar{\boldsymbol{M}} \ddot{\bar{\boldsymbol{\xi}}} + \bar{\boldsymbol{D}} \dot{\bar{\boldsymbol{\xi}}} + \bar{\boldsymbol{K}} \bar{\boldsymbol{\xi}} = \bar{\boldsymbol{\eta}}_e + \bar{\boldsymbol{B}} \boldsymbol{\eta}^m \tag{9a}$$

$$\boldsymbol{\xi}^m = \bar{\boldsymbol{B}}^{\mathrm{T}} \bar{\boldsymbol{\xi}} \tag{9b}$$

with the new quantities

$$\bar{\boldsymbol{M}} = \bar{\boldsymbol{\Phi}}^{\mathrm{T}} \boldsymbol{M} \bar{\boldsymbol{\Phi}}, \quad \bar{\boldsymbol{D}} = \bar{\boldsymbol{\Phi}}^{\mathrm{T}} \boldsymbol{D} \bar{\boldsymbol{\Phi}}, \quad \bar{\boldsymbol{K}} = \bar{\boldsymbol{\Phi}}^{\mathrm{T}} \boldsymbol{K} \bar{\boldsymbol{\Phi}}, \quad \bar{\boldsymbol{\eta}}_e = \bar{\boldsymbol{\Phi}}^{\mathrm{T}} \boldsymbol{\eta}_e, \quad \bar{\boldsymbol{B}} = \begin{pmatrix} \boldsymbol{I}_{6n_m} \\ \boldsymbol{0} \end{pmatrix}.$$

The quantity $\boldsymbol{\eta}^m$ summarizes all forces and torques acting on the body through the connectors while $\boldsymbol{\eta}_e$ covers all remaining external forces influencing the body.

For the sake of simplicity, at this point it is very convenient to change the node numbering according to the position within the vector $\bar{\boldsymbol{\xi}}$. For all further calculations this change will be presumed.

Because the equations above are typically large in scale, the next subsection is dedicated to the reduction of the system size.

3.4 Model order reduction

As already discussed in subsection 2.4, in the majority of cases, it is necessary to reduce tremendously the number of variables and equations of the dynamic model (9).

For our model we used a sophisticated model order reduction algorithm that has been implemented at the Fraunhofer Institute for Integrated Circuits, Design Automation Division in Dresden ([8, 9]). Anyway, there are a lot of other algorithms that can be used to reduce the model size.

[4] If the coordinates in $\boldsymbol{\xi}$ have a different order, the assumed order can be achieved by simply permuting the appropriate lines and columns of all matrices and vectors in equation (8).

However, for a proper inclusion of the large motion behaviour it is necessary that the matrix $\boldsymbol{V}$ includes the six rigid body modes of the compound, i. e. that the matrix includes six columns with the displacements of all nodes when moving the undeformed body a little according to its six DoFs.

3.5 Inclusion of nonlinear terms

In addition to many other physical domains in a mechanical multi-body library we have to take some nonlinear terms into account, namely the nonlinear dynamic forces resulting from the large motions in the three-dimensional space. To do so, we consider the original equations (9) in a moving frame. So, we replace all time derivatives $\dot{()}$ w. r. t. the inertial frame $\mathcal{I}$ by time derivatives $\mathring{()}$ w. r. t. the moving reference frame $\mathcal{B}$ and express the acceleration as a linear superposition of the acceleration w. r. t. the moving reference frame and the acceleration of the moving reference frame itself. Then we can write equations (9) as

$$\bar{\boldsymbol{M}} \left(\mathring{\mathring{\bar{\boldsymbol{\xi}}}} + \boldsymbol{a}_0 \right) + \bar{\boldsymbol{D}} \mathring{\bar{\boldsymbol{\xi}}} + \bar{\boldsymbol{K}} \bar{\boldsymbol{\xi}} = \bar{\boldsymbol{\Phi}}^{\mathrm{T}} \boldsymbol{\eta}_e + \boldsymbol{\eta}_c + \bar{\boldsymbol{B}} \boldsymbol{\eta}^m$$

$$\boldsymbol{\xi}^m = \bar{\boldsymbol{B}}^{\mathrm{T}} \bar{\boldsymbol{\xi}}$$

with $\boldsymbol{a}_0$ consisting of three lines, namely

$$(\boldsymbol{a}_0)_i = \ddot{\boldsymbol{r}}_0 + (\dot{\tilde{\boldsymbol{\omega}}} + \tilde{\boldsymbol{\omega}}^2)(\tilde{\boldsymbol{r}}_i + \boldsymbol{\xi}_i) + 2\tilde{\boldsymbol{\omega}} \dot{\boldsymbol{\xi}}_i, \quad i \in \Omega_r$$

for every node in Ω_r which is the set of all node indices that are neither slave nore master node of any connector.

For every master node, one has to add the following six lines to the vector $\boldsymbol{a}_0$

$$(\boldsymbol{a}_0)_i = \begin{pmatrix} \ddot{\boldsymbol{r}}_0 + (\dot{\tilde{\boldsymbol{\omega}}} + \tilde{\boldsymbol{\omega}}^2)(\tilde{\boldsymbol{r}}_{c,i} + \boldsymbol{\xi}_i) + 2\tilde{\boldsymbol{\omega}} \dot{\boldsymbol{\xi}}_i \\ \dot{\boldsymbol{\omega}} + \tilde{\boldsymbol{\omega}} \dot{\boldsymbol{\xi}}_i \end{pmatrix}, \quad i \in \Omega_m$$

with Ω_m being the index set of connectors.

Furthermore, we have to add for every master node an additional nonlinear expression due to the rotational DoFs that can be combined in the vector $\boldsymbol{\eta}_c$ as

$$(\boldsymbol{\eta}_c)_i = \begin{cases} \begin{pmatrix} \tilde{\boldsymbol{\omega}} & \boldsymbol{0} \\ \boldsymbol{0} & \tilde{\boldsymbol{\omega}} \end{pmatrix} \bar{\boldsymbol{M}}_{i,i} \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{\omega} \end{pmatrix} & \text{for } i \in \Omega_m \\ \boldsymbol{0} & \text{for } i \in \Omega_r. \end{cases}$$

Then, applying the model order reduction and an appropriate projection to the linear part of the equations, one can end up with the following equations of motion:

$$\begin{pmatrix} m\boldsymbol{I} & m\tilde{\boldsymbol{r}}_s^{\mathrm{T}} & \boldsymbol{M}_{b1}^{\mathrm{T}} \\ m\tilde{\boldsymbol{r}}_s & \boldsymbol{\Theta}_0 & \boldsymbol{M}_{b2}^{\mathrm{T}} \\ \boldsymbol{M}_{b1} & \boldsymbol{M}_{b2} & \boldsymbol{M}_q \end{pmatrix} \begin{pmatrix} \ddot{\boldsymbol{r}}_0 \\ \dot{\boldsymbol{\omega}} \\ \ddot{\boldsymbol{q}} \end{pmatrix} + \check{\boldsymbol{D}} \dot{\boldsymbol{q}} + \check{\boldsymbol{K}} \boldsymbol{q} \tag{10a}$$

$$+ \boldsymbol{g}(\boldsymbol{\omega}, \boldsymbol{q}, \dot{\boldsymbol{q}}) = \boldsymbol{V}^{\mathrm{T}} \bar{\boldsymbol{\Phi}}^{\mathrm{T}} \boldsymbol{\eta}_e + \boldsymbol{B}_q \boldsymbol{\eta}^m$$

$$\boldsymbol{\xi}^m = \bar{\boldsymbol{B}}^{\mathrm{T}} \bar{\boldsymbol{\xi}} \tag{10b}$$

Here $\boldsymbol{M}_q = \boldsymbol{V}^{\mathrm{T}} \bar{\boldsymbol{M}} \boldsymbol{V}$ and $\boldsymbol{B}_q = \boldsymbol{V}^{\mathrm{T}} \bar{\boldsymbol{B}}$.

4. Simulation examples

Within this section some examples are presented, that illustrate the previously explained work flow and show the accuracy of this approach.

All models are formulated using the modelling language Modelica. For this purpose a tool, the FEM-Import-Tool, has been developed that reads the data exported from the Finite Element tool, does all necessary modifications including an optional model order reduction, and finally generates a Modelica based model. The model can than easily be integrated into a threedimensional multi-body library. Here, the programme is able to generate two different types of models, one tailored to the particular needs of the SimulationX multi-body library and one fitting to the Modelica Standard Library. For the latter, it was very beneficial to use an equation based language like Modelica, as the body class from the multi-body library had only to be extended for a model of flexible bodies.

4.1 Example 1 – static deformation of a long beam

As a first example the static deformation of a long beam due to a static force and torque will be investigated. On one side the beam will be rigidly fixed, while on the other side a force and a torque will be applied. Table 1 lists all relevant properties of the analysed beam.

Length:	$l = 1\,\mathrm{m}$
Width:	$b = 0,01\,\mathrm{m}$
Height:	$h = 0,02\,\mathrm{m}$
Density:	$\rho = 7850\,{}^{\mathrm{kg}}/_{\mathrm{m}^3}$
Young's modulus:	$E = 2 \cdot 10^{11}\,\mathrm{Pa}$
Poisson's ratio:	$\nu = 0.3$
Damping (Rayleigh): $(\boldsymbol{D} = \alpha\boldsymbol{M} + \beta\boldsymbol{K})$	$\alpha = 1\,\mathrm{s}^{-1}$ $\beta = 0.001\,\mathrm{s}$

Table 1. Model parameters of the long beam

According to the foregoing sections, first, the body has to be modelled within a Finite Element programm as a solid body in order to extract all necessary parameter matrices. Figure 3 shows the discretized ANSYS model (with all the boundary conditions applied to it).

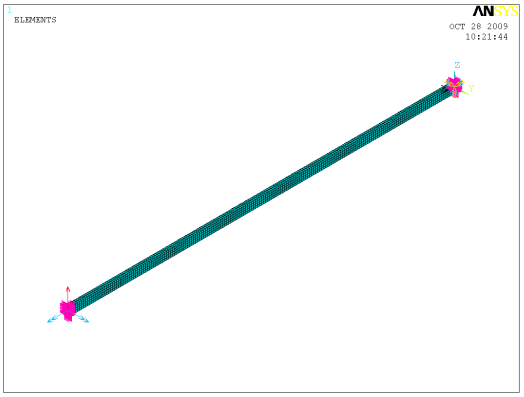

Figure 3. Discretized model of the long beam

4.1.1 Reference

According to [16][5], the theoretical solution for the deviation u_z and the rotations φ_x as well as φ_y of the free flange are given by

$$u_z = \frac{F_z l^3}{3EI_y} - \frac{M_y l^2}{2EI_y}, \qquad \varphi_x = \frac{M_x l}{S_t}$$
$$\varphi_y = \frac{M_y l}{EI_y} - \frac{F_z l^2}{2EI_y}$$

with

$$I_y = \frac{bh^3}{12}, \qquad S_t = \frac{Ghb^3}{3}.$$

As a second reference, the same static analysis has been carried out with ANSYS.

Afterwards the model for the multi-body library has been generated using the FEM-Import-Tool. Here a model order reduction was necessary to reduce the model complexity. The produced Modelica model could than be used to implement a simulation model in Dymola for this specific example as seen in Figure 4.

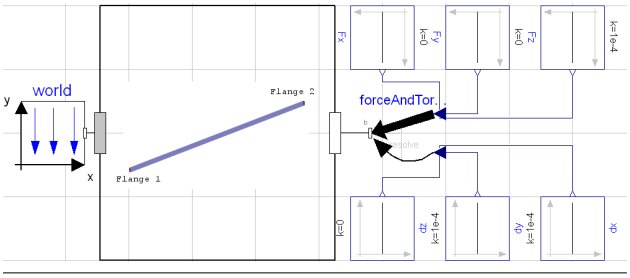

Figure 4. Dymola model of the long beam example

4.1.2 Results in Dymola

The results of the simulation in Dymola are listed in the following Table.

u_z [m]	φ_x [rad]	φ_y [rad]
$-1.1897 \cdot 10^{-008}$	$2.6823 \cdot 10^{-007}$	$3.5687 \cdot 10^{-008}$

4.1.3 Interpretation

Table 2 shows the relative errors of the results of the simulation in Dymola compared to the reference calculation and the reference simulation in ANSYS, respectively.

	Calculation	ANSYS
u_z	0.000614	$4.08 \cdot 10^{-005}$
φ_x	0.444	$1.75 \cdot 10^{-005}$
φ_y	0.000751	$1.36 \cdot 10^{-005}$

Table 2. Relative error of the results of Dymola compared to the reference calculation and ANSYS results

The results in Dymola coincide with the reference results in ANSYS up to the fourth decimal place. Also, the relative error of the variables u_z and φ_y are suffiently small with less than 0.08% compared to the theoretical solutions. The large deviation of the variable φ_x compared to the theoretical solution can be explained through the bad approximation of the polar geometrical moment of inertia S_t in [16]. A better approximation would lead to much better results.

[5] The equation for φ_y in [16] contained a wrong sign, which was corrected here.

4.2 Example 2 – eigenfrequency analysis of an L-shaped beam

For the second example an eigenfrequency analysis for an L-shaped beam (see Figure 5) was performed and compared to the theoretical calculations in [20] as well as to reference simulation results in ANSYS. The beam has a rectangular cross section. It is modelled as a solid body.

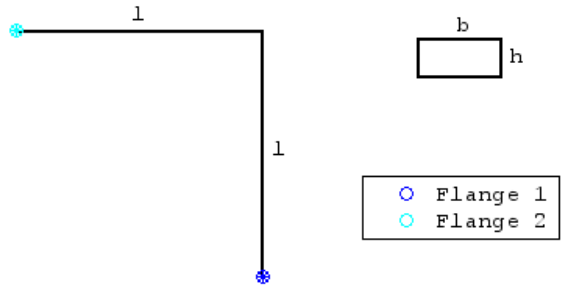

Figure 5. L-shaped beam with its flanges

The dimension and its material properties can be taken from Table 3.

Length:	$l = 1\,\mathrm{m}$
Width:	$b = 12\,\mathrm{mm}$
Height:	$h = 5\,\mathrm{mm}$
Density:	$\rho = 7900\,{}^{\mathrm{kg}}/_{\mathrm{m}^3}$
Young's modulus:	$E = 2.1 \cdot 10^{11}\,\mathrm{Pa}$
Shear modulus:	$G = 82 \cdot 10^{9}\,\mathrm{Pa}$
Damping (Rayleigh): ($\boldsymbol{D} = \alpha\boldsymbol{M} + \beta\boldsymbol{K}$)	$\alpha = 0\,\mathrm{s}^{-1}$ $\beta = 0\,\mathrm{s}$

Table 3. Model parameters of the L-shaped beam

The beam is rigidly fixed on one flange (flange 1) to the inertial frame, while the second flange remains free. The objective of this example is the proof of accuracy concerning the first in-plane eigenfrequencies of the generated model.

Again, for the generation of a Modelica model, the first step was to export all necessary data from an appropriate Finite Element model. For this example each line of the L-shaped beam has been discretized into ten elements. Afterwards the Modelica model has been generated using FEM-Import-Tool with the expansion points $s = \pm 40\pi i$ and $s = \pm 280\pi i$ for the model order reduction.

4.2.1 Reference

The theoretic results from the paper [20] and the eigenmode analysis of ANSYS are taken as the reference for the Dymola simulation.

Figure 6 shows the first five eigenmodes of the L-shaped beam in the considered plane. These eigenmodes are compared to the simulation results in Dymola.

4.2.2 Results in Dymola

The results of the eigenvalue analysis in Dymola can be compared to the theoretic results from [20], as well as the ANSYS results which all are listed in the following table.

Figure 6. First five eigenmodes of the L-shaped beam

	Theory [20]	ANSYS	Dymola
f_1	3.331 Hz	3.337 Hz	3.337 Hz
f_2	9.070 Hz	9.121 Hz	9.121 Hz
f_3	44.772 Hz	44.802 Hz	44.802 Hz
f_4	65.687 Hz	66.03 Hz	66.03 Hz
f_5	143.179 Hz	143.173 Hz	143.173 Hz

4.2.3 Interpretation

Table 4 lists the relative errors of the Dymola frequencies compared to the eigenfrequencies of the reference simulation in ANSYS and the theoretic results.

	Calculation	ANSYS
f_1	0.00185	$5.16 \cdot 10^{-005}$
f_2	0.00564	$1.73 \cdot 10^{-005}$
f_3	0.000667	$3.44 \cdot 10^{-006}$
f_4	0.00523	$6.07 \cdot 10^{-006}$
f_5	$4.39 \cdot 10^{-005}$	$1.98 \cdot 10^{-006}$

Table 4. Relative error of the eigenfrequency analysis in Dymola compared to ANSYS and [20]

The maximal relative error is approximately 0.56% compared to the theoretic results and smaller than 0.01% compared to the ANSYS results. Hence, the deviation is suffiently small.

4.3 Example 3 – T-square under uniform rotation

The third example is a dynamic test that shows the effect of a uniform rotation of a body. The T-square under investigation (see Figure 8) is fixed at one flange (flange1) to a revolute joint that rotates with $\sqrt{3} \cdot 60\,\mathrm{rpm}$ around its axis ($\boldsymbol{n} = \frac{1}{\sqrt{3}}\begin{pmatrix}1 & 1 & 1\end{pmatrix}$). The static deformation due to the centrifugal forces acting on the body are determined.

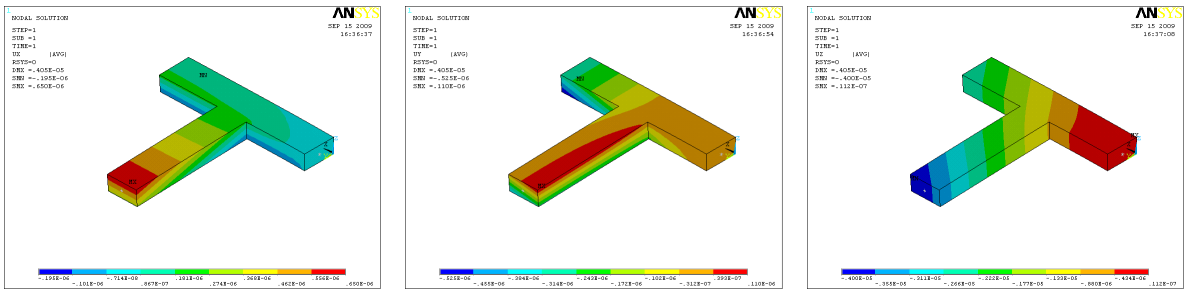

Figure 7. Reference results for the static deformation of the rotating T-square

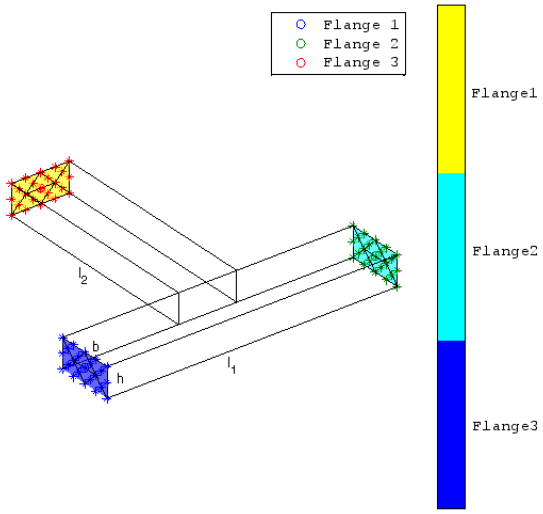

Figure 8. T-square with its flanges and dimensions

All relevant properties of the T-square are listed in Table 5.

Length 1:	$l_1 = 20\,\mathrm{cm}$
Length 2:	$l_2 = 15\,\mathrm{cm}$
Width:	$b = 4\,\mathrm{cm}$
Height:	$h = 2\,\mathrm{cm}$
Density:	$\rho = 7850\,{}^{\mathrm{kg}}/_{\mathrm{m}^3}$
Young's modulus:	$E = 2 \cdot 10^{11}\,\mathrm{Pa}$
Poisson's ratio:	$\nu = 0.3$
Damping (Rayleigh): $(\boldsymbol{D} = \alpha\boldsymbol{M} + \beta\boldsymbol{K})$	$\alpha = 1\,\mathrm{s}^{-1}$ $\beta = 0.001\,\mathrm{s}$

Table 5. Model parameters of the T-square

4.3.1 Reference

After discretizing the model all necessary data have been exported and a reference simulation has been carried out within the Finite Element tool ANSYS (see Figure 7).

Furthermore a Modelica model has been generated and simulated with the tool SimulationX.

4.3.2 Results in SimulationX

The simulation has been carried out in two different ways. In the first case, a model was used in which the flanges had already been strutted within the Finite Element tool by means of constraint equations. For the other simulation this was not the case. Here, all flanges had been strutted in the model generator.

The results of the simulation for both versions do not differ up to the sixth decimal place and are shown in the table below.

Displacement		
	Flange 2	Flange 3
u_x	$9.6957 \cdot 10^{-009}$	$5.0039 \cdot 10^{-007}$
u_y	$-4.0418 \cdot 10^{-007}$	$-1.3583 \cdot 10^{-007}$
u_z	$-2.3823 \cdot 10^{-006}$	$-3.6993 \cdot 10^{-006}$

4.3.3 Interpretation

Table 6 lists the relative error of all results of the simulation in SimulationX compared to the reference simulation in ANSYS.

	Flange 2		Flange 3	
	Without CEs	With CEs	Without CEs	With CEs
u_x	0.000217	0.000316	0.000152	0.000152
u_y	0.00015	0.000151	0.000188	0.000219
u_z	0.000283	0.000286	0.000174	0.000168

Table 6. Relative error between results of reference simulation and results of SimulationX

The maximal relative error between ANSYS and SimulationX is lower than 0.03%. Hence the deviations remain sufficiently small and the model achieves a good approximation.

5. Outlook

As stated at the beginning, there are also other examples from different physical domains that use nearly the same procedure to derive models for the combined simulation with lumped parameter models. One example has been investigated in [5]. Here, a thermal model of an electric motor has been studied as depicted in Figure 9. The work flow that has been used can exactly be mapped on the method described in section 2. This fact gives motivation to further investigate examples of different domains in order to enhance and consolidate the work flow explained in this paper.

For a more general and more sophisticated method of including distributed parameter models into libraries of lumped parameter models, the approach of port-based modelling seems very attractive and promising. Especially

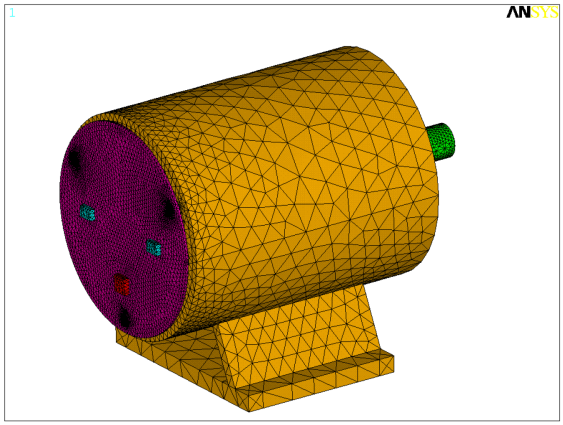

Figure 9. Discretized thermal model of a electric motor

in the area of port-hamiltonian systems already some remarkable results have been achieved that might be used to effectively integrate distributed parameter models [4, 10].

In this context, for every port, there exist pairs of so-called conjugate variables (flow and efforts) with the property, that their product results in a physical quantity that can be interpreted as a power. Please note, that within a connector there should not disappear any power. This fact leads to a special symmetry within the governing differential equations. Hence, it seems also very interesting to study the integration process for this type of models.

6. Summary and Conclusions

The paper presented an approach of including discretized distributed parameter models into libraries of lumped parameter models for equation-based simulation tools. As an example, for a solid state body the authors showed how to import the Finite Element discretized model into a classical multi-body equation-based library. The result was a model that was able to describe the behaviour of the body in terms of its elastic deformations also for large motions. In order to achieve that goal, it was necessary to define appropriate connectors and to reduce the size of the spatially discretized model by a model order reduction algorithm. The generated model was produced using the modelling language Modelica that fully realizes the equation-based modelling paradigm and thus offers the opportunity of simply changing existing models according to the new requirements.

References

[1] A. C. Antoulas. *Approximation of large-scale dynamical systems*. Society for Industrial & Applied Mathematics, 2005.

[2] K.-J. Bathe. *Finite Element Procedures*. Prentice Hall, 1996.

[3] H. Bremer and F. Pfeiffer. *Elastische Mehrkörpersysteme*. Teubner, 1992

[4] V. Duindam, A. Macchelli, S. Stramigioli, and H. Bruyninckx (Eds.). *Modeling and Control of Complex Physical Systems*. Springer, 2009.

[5] O. Enge-Rosenblatt, P. Schneider, C. Clauss, and A. Schneider. Functional Digital Mock-up – Coupling of Advanced Visualization and Functional Simulation for Mechatronic System Design. *ASIM Workshop*, Ulm, 2010.

[6] R.W. Freund. Model reduction methods based on Krylov subspaces. *Acta Numerica*, 12:267–319, 2003.

[7] E. J. Haug. *Computer aided kinematics and dynamics of mechanical systems - 1*. Prentice Hall, 1989.

[8] A. Köhler. *Modellreduktion von linearen Deskriptorsystemen erster und zweiter Ordnung mit Hilfe von Block-Krylov-Unterraumverfahren*. Diplomarbeit. TU Dresden, Germany, 2006.

[9] A. Köhler, C. Clauss, S. Reitz, J. Haase, and P. Schneider. Snapshot - Based Parametric Model Order Reduction. *MATHMOD Wien*, February, 2009.

[10] A. Macchelli, A. J. van der Schaft, and C. Melchiorri. Distributed port-Hamiltonian formulation of infinite dimensional systems. In: *Proc. 16th International Symposium on Mathematical Theory of Networks and Systems*, MTNS 2004 (2004).

[11] K.W. Morton and D.F. Mayers. *Numerical Solution of Partial Differential Equations. An Introduction.* Cambridge University Press, 2005.

[12] L. Saldamli. *PDEModelica – A High-Level Language for Modeling with Partial Differential Equations*. Linköping Studies in Science and Technology, Dissertation No. 1022, Linköping, 2006.

[13] P. Schneider et. al. Functional Digital Mock-up - More Insight to Complex Multi-physical Systems. *Multiphysics Simulation - Advanced Methods for Industrial Engineering* 1st International Conference, Bonn, Germany, June 22-23, 2010, Proceedings.

[14] P. Schneider, P. Schwarz, and S. Wünsche. Beschreibungsmittel für komplexe Systeme. *40. Intern. Wiss. Kolloquium der TH Ilmenau*, September 7-9, 1995, Band 3, p. 102–108.

[15] A. A. Shabana. *Dynamics of Multibody Systems*. Cambridge University Press, 2nd Edition, 1998.

[16] A. L. Schwab and J. P. Meijaard. Beam Benchmark Problems for Validation of Flexible Multibody Dynamics Codes. *MULTIBODY DYNAMICS 2009, ECCOMAS Thematic Conference*, June 29 - July 2, 2009.

[17] P. Schwarz. Physically oriented modeling of heterogeneous systems. *3. IMACS Symp. MATHMOD* pp. 309-318, Wien, 2000.

[18] R. Schwertassek and O. Wallrapp. *Dynamik flexibler Mehrkörpersysteme*. Vieweg Verlag, 1999.

[19] A. Stork. *FunctionalDMU – Eine Initiative der Fraunhofer Gesellschaft*. 2006, URL: www.functionaldmu.org, seen at 17 May, 2010.

[20] R. E. Valembois, P. Fisette, and J. C. Samin. Comparison of Various Techniques for Modelling Flexible Beams in Multibody Dynamics. In: *Nonlinear Dynamics*, p. 367-397, Kluwer Acadamic Publishers, 1997.

Synchronous Events in the OpenModelica Compiler with a Petri Net Library Application

Willi Braun[1] Bernhard Bachmann[1] Sabrina Proß[1]

[1]Department of Applied Mathematics, University of Applied Sciences Bielefeld, Germany,
{wbraun,bernhard.bachmann,spross}@fh-bielefeld.de

Abstract

In this work an approach is presented that extends the OpenModelica Compiler (OMC) with an event handling module and controls events separately from the integrator. The aim of this extension is to improve the event handling controller of the OMC to handle all equations synchronously, resulting in an efficient simulation of hybrid dynamical systems. This improvements of the event handling allows to formulate the Petri Net library in optimal Modelica code.

Keywords Modelica, Hybrid Models, Petri Nets, OpenModelica, Synchronous

1. Introduction

In general, Modelica models are represented mathematically through differential-algebraic equations (DAEs). A special feature of the Modelica language is the ability to describe continuous and discrete processes in a so-called hybrid model. Hybrid models consist of continuous differential and algebraic as well as discrete equations. The latter introduces events during simulation.

Typical applications for hybrid models are electronic circuits or models with collisions of bodies. These models generate events which can change the behaviour of the system. Furthermore, there are also approaches within event-based modelling, e.g. with hybrid Petri Nets, which involve discrete and continuous places and transitions as well as stochastic transitions.

For the numerical simulation of hybrid systems, a special treatment of events is therefore needed. In addition to a robust numerical integration of the DAEs, the instant of time in which events modify the system should be approximately determined and events should be treated in the correct chronological sequence as they appear. Modelica requires that all equations are handled synchronously at all points in time, even more at events.

In OpenModelica a version of the DASSL algorithm with associated root finding (DASRT) is used, so that the event handling can not be considered independently from the solver [5]. The Petri Net Library [8] could not been expressed in optimal Modelica code since at that time the OMC did not treat discrete events synchronously. This paper describes the correct treatment of the event handling which has been implemented in the OMC, leading to an enhanced formulation of the Petri Net library.

2. Modelica Synchronous Data-flow Principle

A hybrid Modelica model consists of differential, algebraic and discrete equations. The discrete equations are not permanently active, they are only activated when an event occurs. It is important to keep all variables synchronously at all time points.

In order to solve Modelica models efficiently, all equations are sorted into the block-lower-triangular form. The Modelica synchronous principle states that at every time instant, the active equations express relations between variables which have to be fulfilled concurrently (cf. [7]). Based on this synchronous data-flow principle, all equations are considered active during sorting to ensure a correct order at all points of time. The idea of using the synchronous data flow principle in the context of hybrid systems was introduced in [4].

The following Modelica example illustrates how all equations are kept synchronously.

```
when y1 > 2 then
  y2 = f1(y3);
end when;
y3 = f3(y4);
when y0 > 0 then
  y4 = f2(u);
  y1 = f4(y3);
end when;
y0 = f5(u);
```

The example consists of five equations. The three `when`-equations are only active at the points of time at which

3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. October, 2010, Oslo, Norway.

the conditions are even fulfilled. The other two equations describe the continuous behavior. The order of evaluation plays an important role to ensure the synchronous principle.

The individual equations are sorted according to the contained variables. So it must be assumed that all equations are activated simultaneously during transformation into a block-lower-triangular form. Then the right evaluation order can be determined automatically. To sort the `when`-equations correctly, it has to be noted that they also depend on variables that occur in the condition of a `when`-expression.

If the block-lower-triangular transformation is performed on these principles, the evaluation results in the following specified order.

```
//known Variable: u
  y0 = f5(u);
  when y0 > 0 then
    y4 = f2(u);
  end when;
  y3 = f3(y4);
  when y0 > 0 then
    y1 = f4(y3);
  end when;
  when y1 > 2 then
    y2 = f1(y3);
  end when;
```

During continuous integration and also at events the sorting order is always correct because the discrete variables are kept constant during the continuous integration [6].

3. Hybrid Modelica Model represented as DAEs

Flat hybrid DAEs could represent continuous-time behavior and discrete-time behavior. This is done mathematically by the equation (1).

$$F(\underline{\dot{x}}(t), \underline{x}(t), \underline{u}(t), \underline{y}(t), \underline{q}(t_e), \underline{q_{pre}}(t_e), \underline{c}(t_e), \underline{p}, t) = 0 \tag{1}$$

This implicit equation (1) is transformed to the explicit representation of equation (2) by block-lower-triangular transformation.

$$\begin{pmatrix} \underline{\dot{x}}(t) \\ \underline{y}(t) \\ \underline{q}(t_e) \end{pmatrix} = \begin{pmatrix} \underline{f_s}(\underline{x}(t), \underline{u}(t), \underline{p}, \underline{q_{pre}}(t_e), \underline{c}(t_e), t) \\ \underline{f_a}(\underline{x}(t), \underline{u}(t), \underline{p}, \underline{q_{pre}}(t_e), \underline{c}(t_e), t) \\ \underline{f_q}(\underline{x}(t), \underline{u}(t), \underline{p}, \underline{q_{pre}}(t_e), \underline{c}(t_e), t) \end{pmatrix} \tag{2}$$

From this explicit form all necessary calculations can be educed for the simulation of hybrid models. This is done by formulating the continuous-time part, followed by the discrete-time part. Below are summarized the notation used in the following equations:

- $\underline{\dot{x}}(t)$, the differentiated vector of state variables of the model.
- $\underline{x}(t)$, the vector of state variables of the model, i.e., variables of type `Real` that also appear differentiated, meaning that `der()` is applied to them somewhere in the model.
- $\underline{u}(t)$, a vector of input variables, i.e., not dependent on other variables, of type `Real`. They also belong to the set of algebraic variables since they do not appear differentiated.
- $\underline{y}(t)$, a vector of Modelica variables of type `Real` which do not fall into any other category.
- $q(t_e)$, a vector of discrete-time Modelica variables of type discrete `Real`, `Boolean`, `Integer` or `String`. These variables change their value only at event instants, i.e., at points t_e in time.
- $q_{pre}(t_e)$, the values of q immediately before the current event occurred, i.e., at time t_e.
- $c(t_e)$, a vector containing all `Boolean` condition expressions evaluated at the most recent event at time t_e. This includes conditions from all `if`-equations and `if`-statements and `if`-expressions from the original model as well as those generated during the conversion of `when`-equations and `when`-statements.
- $p = p1, p2, \ldots$, a vector containing the Modelica variables declared as parameter or constant i.e., variables without any time dependency.
- t, the Modelica variable time, the independent variable of type `Real` implicitly occurring in all Modelica models.

3.1 Continuous Behavior

The continuous behavior of hybrid DAEs can be formulated with the following equations (3).

$$\begin{pmatrix} \underline{\dot{x}}(t) \\ \underline{y}(t) \end{pmatrix} = \begin{pmatrix} \underline{f_s}(\underline{x}(t), \underline{u}(t), \underline{q_{pre}}(t_e), \underline{c}(t_e), \underline{p}, t) \\ \underline{f_a}(\underline{x}(t), \underline{u}(t), \underline{q_{pre}}(t_e), \underline{c}(t_e), \underline{p}, t) \end{pmatrix} \tag{3}$$

The states $\underline{x}(t)$ are determined by an integration method, so that they are assumed to be known as the vectors $\underline{u}(t)$ and $\underline{p}$. For discrete variables and the condition expressions t_e is used instead of t to indicate that such variables may only change values at event points of time and are kept constant in the continuous parts of the simulation. Additional the values of $\underline{q}(t_e)$ and $\underline{q_{pre}}(t_e)$ are equivalent in the continuous parts of the simulation. Since every dependents of the functions $\underline{f_s}$ and $\underline{f_a}$ are known, the continuous behavior is fully described by (3).

3.2 Discrete Behavior

The discrete behavior is controlled by events. Events are triggered by the event conditions $\underline{c}(t_e)$ and can appear at any time as well as influence the system several times.

An event occurs when a condition of $\underline{c}(t_e)$ change its value at time t_e from `false` to `true` or the other way around. This occurs if and only if for a sufficient small ϵ, one condition in $\underline{c}(t_e)$ is changed, for e.g. $\underline{c}(t_e - \epsilon)$ is `false` and for $\underline{c}(t_e + \epsilon)$ is `true`. When an event occurs all caused changes in the system can be carried out. In addition, the entire system must be determined by the function (2) to guarantee the synchronism of all equations. However,

it is not enough to determine only the discrete variables by the function $\underline{f_q}$ at this point.

The problem to be solved here is the most accurate determination of the event time t_e. For this conditions $\underline{c}(t_e)$ can be divided into three groups.

1. Conditions $\underline{c_k}(t_e)$, which also depend on continuous variables.
2. Conditions $\underline{c_d}(t_e)$, that only depend on discrete variables.
3. Conditions $\underline{c_{\text{noEvent}}}(t)$, where the `noEvent()` operator is present.

If the `smooth` operator applies to a condition in $\underline{c}(t_e)$ this condition can be categorized depending on the order of the integration method by 1. or 3., respectively.

The second and third group of conditions are easy to handle, because if a condition in $\underline{c}(t_e)$ only depends on discrete variables, then they could only change at events and the conditions $\underline{c_d}(t_e)$ must be tested only at events. The conditions $\underline{c_{\text{noEvent}}}(t)$ result logically in no events. Thus, the equations which depend on conditions $\underline{c_{\text{noEvent}}}(t)$, will be determined during the continuous integration at the output points. Hence the variables that are determined by the function (4) should be treated appropriately, like algebraic variables.

$$\underline{q_{\text{noEvent}}(t)} := g(\underline{x}(t_e), \underline{u}(t_e), \underline{q_{pre}}(t_e), \underline{c_{\text{noEvent}}}(t_e), \underline{p}, t) \tag{4}$$

The event conditions $\underline{c_k}(t_e)$, that depend only on time, can be considered separately as they lead to time events. The presence of time events is known at the start of the simulation, thus they can be treated efficiently. This separation is not yet implemented in OpenModelica, but can easily be realized.

What remains is the group of conditions, that lead to state events. For this group of conditions a time-consuming search has to be performed. These conditions have to be checked during the continuous solution as described in the next section.

Additional, discontinuous changes can caused by the `reinit()` operator to the continuous states $\underline{x}(t)$. As for purely discrete conditions $\underline{c_d}(t_e)$ the `reinit()` operator can only be activated at event times t_e . This new allocation to the states could use the function (5).

$$\underline{x(t_e)} := \underline{f_x}(\underline{x}(t), \underline{\dot{x}}(t), \underline{u}(t), \underline{q}(t_e), \underline{q_{pre}}(t_e), \underline{c}(t_e), \underline{p}, t) \tag{5}$$

3.3 Crossing Functions

In order to evaluate and check conditions $\underline{c_k}(t_e)$ during the continuous solution, they are converted into continuous functions, so-called ZeroCrossing functions. Such functions have a root if the `Boolean` condition jumps from `false` to `true`. The relation $x > 2$, for example, changes its value from `false` to `true` if $x - 2$ changes their value from less zero to greater than zero, as shown in figure 1.

The values of ZeroCrossing functions are evaluated at the points t_i as this points are provided by the continuous solution. Through the continuous monitoring of ZeroCrossing functions the interval $[t_i, t_{i+1}]$ is obtained, where at least one of the ZeroCrossing function has a zero-crossing. To determine the zero-crossing more precisely, the state values are necessary for the entire interval. A continuous solution for that can be provided by interpolating methods within the interval limits. The order of the used interpolating method should match the order of the integration process, to get an error that only depends on the used step size (cf. [9, p. 197-215]).

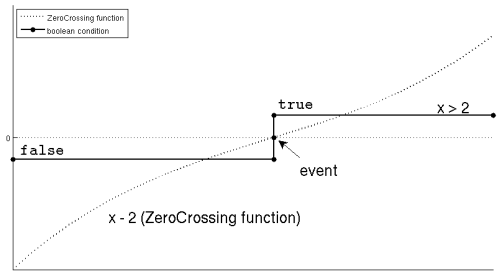

Figure 1. A boolean condition and its ZeroCrossing-function.

4. Hybrid DAE Solution Algorithm

A general approach for the simulation of hybrid systems has been developed by Cellier (cf. [2]). In the following a schematic Flowchart (see fig. 2) for the simulation is shown and each step is described.

First of all the simulation must be initialized consistently. For that the initial values are found with a simplex optimization method in OpenModelica (cf. [1]). By use of the initial conditions the initial values for the entire system can be determined with the function (2). This will also execute all initial events at time t_0.

After the initialization the main simulation loop starts with the continuous integration step that calculates the states $\underline{x}(t_{i+1})$. With the new values of $\underline{x}(t_{i+1})$, the functions $\underline{f_s}$ and $\underline{f_a}$ can be evaluated. Thus, the entire continuous system is determined.

The continuous integration step is accepted if none of the ZeroCrossing functions has a zero-crossing, i.e in $\underline{c}(t_{i+1})$ no value has changed compared to $\underline{c}(t_i)$. If no event has occurred the values can be saved and the next step can be performed.

However, if a value of $\underline{c}(t_{i+1})$ changes, an event occurred within the interval t_i and t_{i+1}. Then the exact time t_e has to be detected. Therefore a root finding method is performed on the ZeroCrossing functions of the corresponding conditions. If several ZeroCrossing functions apply the first occurring root is chosen as the next event time t_e.

The next step is to prepare the treatment of an event by evaluating the system just before an event at time $t_e - \epsilon$, and shortly after the event at $t_e + \epsilon$. Current derivative-free root finding methods work under the principle that the root is approximated through limits at the two sides,

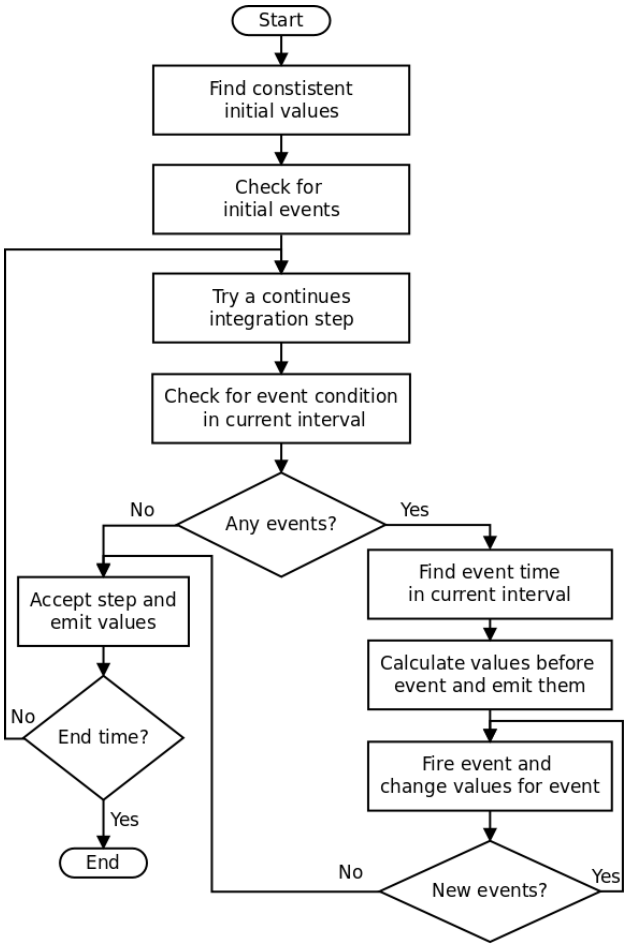

Figure 2. Schematic Flowchart for simulation hybrid models.

so that the delivered root lies somewhere in the interval $[t_e - \epsilon; t_e + \epsilon]$. Here ϵ is the tolerance level of the root finding method. Thus the necessary information to treat the event are available after the root is found.

The treatment of an event looks like that: The continuous part is evaluated at the time just before the event $t_e - \epsilon$ and all values are saved to provide them to the `pre()` operator. Then the entire system is evaluated by the function (2) at time $t_e + \epsilon$. At this point the causing event is handled and now further caused events are processed with the so-called EventIteration. Therefore the entire system constantly is re-evaluated, as long as there exist discrete variables q_j that satisfy $\texttt{pre}(q_j) \neq q_j$. Only if for all discrete variables $\texttt{pre}(q_j) = q_j$ is fulfilled, the EventIteration has reached a stable state and the next integration step can be performed.

5. Test and Evaluation in OpenModelica

In the following two test cases are presented to verify the implementation. The first model will show that `when`-equations are sorted properly and events are processed correctly. The next model is from [5] and an example for EventIteration, that works with only one re-evaluation through the correct order of all equations. Finally a hybrid Petri-Net model with its results is presented.

In order to check the correct sorting for `when`-equations the following test model is considered.

```
model when_sorting
  Real x;
  Real y;
  Boolean w(start=true);
  Boolean v(start=true);
  Boolean z(start=true);
equation
  when y > 2 and pre(z) then
    w = false;
  end when;
  when y > 2 and z then
    v = false;
  end when;
  when x > 2 then
    z = false;
  end when;
  when sample(0,1) then
    x = pre(x) + 1;
    y = pre(y) + 1;
  end when;
end when_sorting;
```

All variables in the model are discrete variables because all are contained within the `when`-equations. The variables `w` and `v` are taken to determine whether the system behaves correctly. The condition `sample(0,1)` creates events at points of time $0.0, 1.0, 2.0, \ldots$, and the variables `x` and `y` are incremented by one at this points. At time point 2.0 the variables `x` and `y` are set to 3, so the other `when`-equations are activated at point of time 2.0 firstly. The evaluation order of the equations is significant for this example. The order of evaluation has to be read on the basis of the following sorted adjacency matrix of the model.

$$\begin{array}{l} \\ y = pre(y) + 1 \\ w = false \\ x = pre(x) + 1 \\ z = false \\ v = false \end{array} \begin{array}{c} \begin{array}{ccccc} y & w & x & z & v \end{array} \\ \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 \\ 1 & \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 & 0 \\ 0 & 0 & 1 & \mathbf{1} & 0 \\ 1 & 0 & 0 & 1 & \mathbf{1} \end{pmatrix} \end{array}$$

The variable `w` is turned to `false` because the variables `y` and `pre(z)` are both `true` and are not dependent on the equation of variable `z`. The variable `v` is not turned because it depends on variable `z` and is always evaluated afterwards. Thus the condition for this equation can not be fulfilled. As a result, the equation that describes the variable `v` is not activated. However, the order of evaluation in this example is not unique, but it would not change the described behavior of the model. The results of the model are illustrated in figure 3.

The following example is developed in [5] for demonstration of the event iteration mechanism.

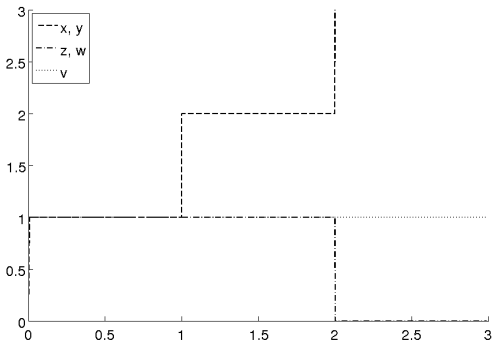

Figure 3. Results of the above example.

```
model EventIteration
  Real x(start = 1);
  discrete Real a(start = 1.0);
  Boolean z(start = false);
  Boolean h1,h2;
equation
  der(x) = a * x;
  h1 = x >= 2;
  h2 = der(x) >= 4;
  when h1 then
    y = true;
  end when;
  when y then
    a = 2;
  end when;
  when h2 then
    z = true;
  end when;
end EventIteration;
```

In the original version of the OMC which did not fulfill the synchronous data flow principle this example produced up to three EventIterations. However, if all equations are sorted in correct order and an event is handled like described above, the chain of events is treated at once. Therefore we consider again the sorted adjacency matrix to the model:

$$\begin{array}{l} \\ h1 = x >= 2 \\ y = true \\ a = 2 \\ der(x) = a * x \\ h2 = der(x) >= 4 \\ z = true \end{array} \begin{array}{c} \begin{array}{cccccc} h1 & y & a & dx & h2 & z \end{array} \\ \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 & 0 \\ 1 & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 1 & \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & 1 & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & 1 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 & 1 & \mathbf{1} \end{pmatrix} \end{array} \tag{6}$$

The first event appears if `x` reaches 2 and the condition `x>=2` is fulfilled. Thus the function (2) is executed and all further caused events are treated directly. So that the variable `z` is turned to `true` after the first evaluation of (2).

6. Petri Net Example in OpenModelica

The Petri Net library developed at the University of Applied Sciences Bielefeld consists of Modelica Models for the basic components of a Petri Net (see [8]). In the library models for discrete and continuous places and transitions as well as stochastic transitions are included. With these basic components Petri Net models can be created intuitively and quickly with a graphical representation. With hierarchical capabilities of Modelica large Petri Net models can be constructed easily.

In the following we present an example of a production process modeled by the Petri Net Library. Figure 4 shows the correnspond Petri Net of the production process of crude steel, compare [3]. At first, the iron ore is transported per ship from Brasilia to a stock at the port of Rotterdam. This trip takes generally 14 days. Every 24 days a ship arrives at the port of Rotterdam. But the exact time of arrival is uncertain. The trip can take a little bit longer or shorter because of nature or other conditions. This is modeled with the aid of a stochastic Transition (Transition ship). The time of arrival is a normal distributed random variable with the expectation value $m = 24$ and the standard deviation $s = 1$. A shipload contains $360.000t$ iron ore. For this reason `add1` of Transition ship is equal to 360.000. The stock at the port can contain at most $720.000t$ iron ore. Therefore, the maximal value of the Place `stock` is fixed to 720.000. The start value of this Place is 360.000.

At the next level the iron ore is loaded from the stock to several trains. A train can contain $5000t$ iron ore and the drive to the steel production in Duisburg takes 8 hours. The iron ore is delivered "just in time" to the production process. Hence, no other stock is needed. The discrete Transition train represents the transport from Rotterdam to Duisburg. The delay is $1/3$ day ($= 8$ hours) and `sub1 = add1 = 5000`. The iron ore (Place pro) and the coke (Place coke) are mixed in the sintering plant. It accrues the intermediate product sinter (Place `I1`). For one ton employed iron ore $0.2t$ coke is needed and $0.73t$ sinter is produced. This production step is modeled continuously by means of the Transition `Si`. The edge weightings are the following:

```
sub1 = 0.2 pro.t
sub2 = pro.t
add1 = 0.73 pro.t
```

The sinter is further processed in the blast furnace to hot metal (Place `I2`). In addition, the by-products slag (Place `slag`) and blast furnace dust (Place `dust1`) are produced. For one ton employed sinter $0.2t$ coke is needed and $0.1t$ slag, $0.65t$ hot metal and $0.01t$ blast furnace dust are produced. The Transition `Fu` displays this. The edges weightings are:

```
sub1 = 0.2 I1.t
sub2 = I1.t
add1 = 0.1 I1.t
add2 = 0.65 I1.t
add3 = 0.01 I1.t
```

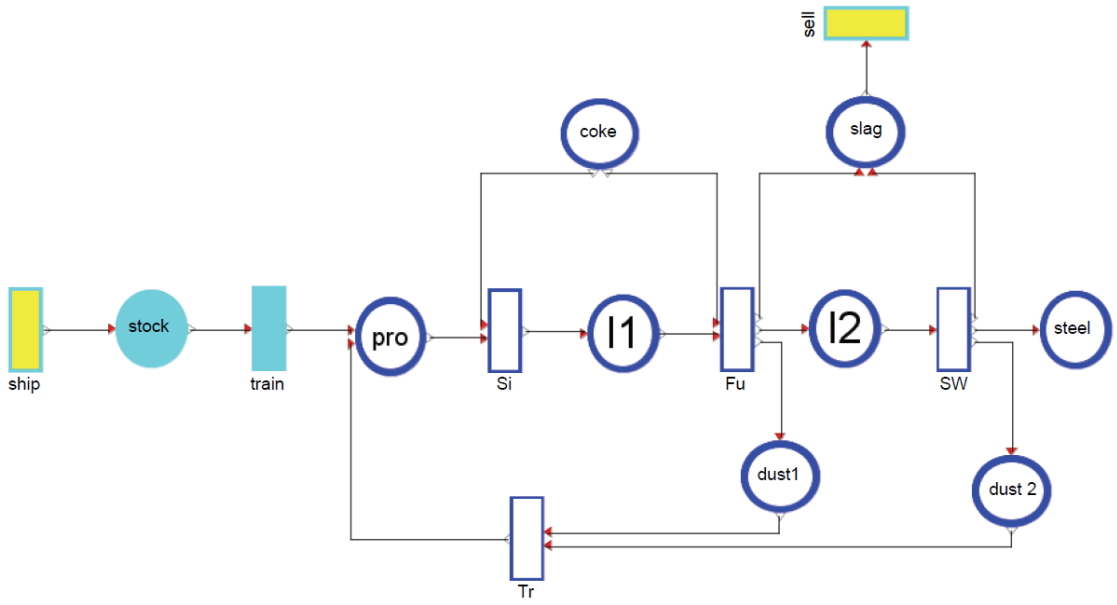

Figure 4. Steel production process.

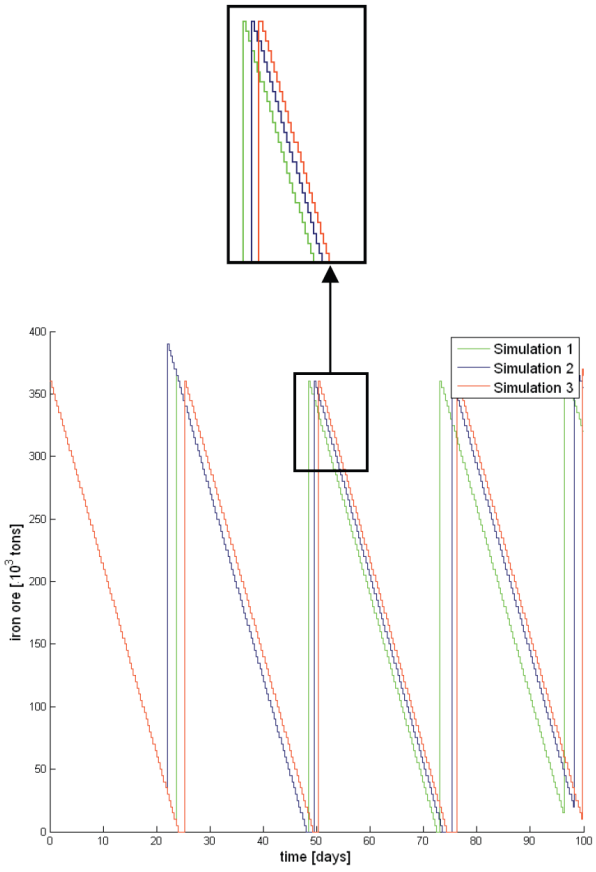

Figure 5. Three simulation results of the iron ore stock at the port Rotterdam.

The by-product slag is sold to building industry. When $50.000t$ slag are produced the company is informed but it is uncertain when the company arrives to pick up the slag and how long this procedure takes. This is modeled with a stochastic Transition with a normal distributed delay ($m = 1/2$ and $s = 1/8$) and `sub1 = 50.000`. In the last production step the hot metal is processed to crude steel (Place `steel`) in the steel works. Slag (Place `slag`) and converter dust (Place `dust2`) are the by-products here. For one ton employed hot metal $0.13t$ slag, $0.8t$ crude steel and $0.05t$ converter dust are produced. The Transition `SW` represents the steel works. The edge weightings are:

```
sub1 = I2.t
add1 = 0.13 I2.t
add2 = 0.8 I2.t
add3 = 0.05 I2.t
```

Iron ore can be substituted by blast furnace dust (Place `dust1`) and converter dust (Place `dust2`). This is modeled with the Transition `Tr` and the edges weightings are:

```
sub1 = dust1.t
sub2 = dust2.t
add1 = 0.1 (dust1.t +dust2.t)
```

Following, some simulation results are shown. Figure 5 displays three possible progressions of the stock of iron ore at the port of Rotterdam. Every progression is different because of the stochastic modeling. The stock is limited to $720.000t$ iron ore. Hence, this border is not exceed. The iron ore is loaded to trains. Every 8 hours a train drives with $5000t$ iron ore to Duisburg. These are the discrete stages in the magnification. The iron ore is exhausted in all simulations at specific time points:

Simulation1	Simulation2	Simulation3
48 - 48.5	48 - 49.5	24 - 25.25
72.5 - 73	73.5 - 75.4	49.25 - 50.31
		74.31 - 76.28

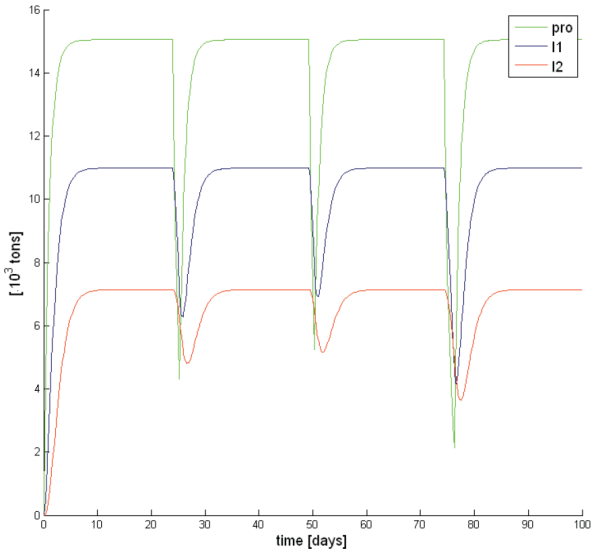

Figure 6. The progressions of iron ore (pro), sinter (I1) and hot metal (I2) (simulation 3).

This causes bottlenecks in the production process. The next figure 6 shows the progression of iron ore, sinter and hot metal of simulation 3. The decrease after day 24.3, 49.6 and 74.4 is caused by the exhaused stocks. Figure 7 illustrates the bottleneck in the production process of simulation 3, too. The exhausted stocks are reflected in the amount of crude steel. The production is decreased after every empty stock period.

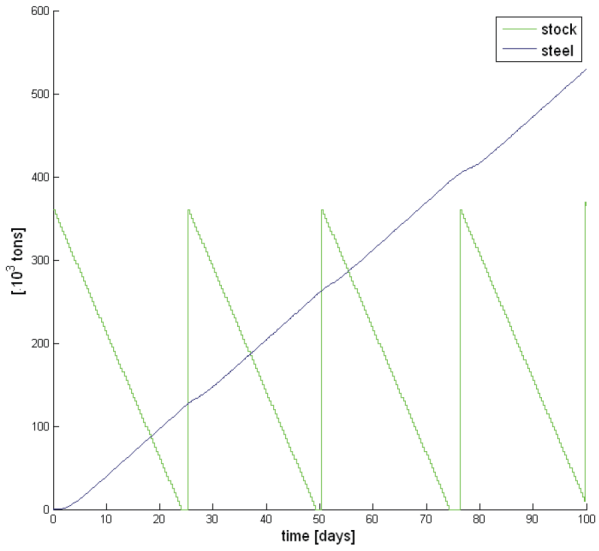

Figure 7. Stock of iron ore (stock) and produced crude steel (steel) by comparison (simulation 3).

The conclusion of these simulations is that the delivery period of iron ore has to be reduced. The new period has to be big enough that only small bottlenecks appear and small enough that no high stocks accumulate. If for example a period of 22.5 days is chosen, the probability of a bottleneck is 6.7% and the probability that this bottleneck takes longer than one day is 0.62%. Now is the task to find the "optimal" solution between bottlenecks and stock costs. Figure 8 shows three simulation results of the progression of the iron ore stock if the delivery period is 22.5 days.

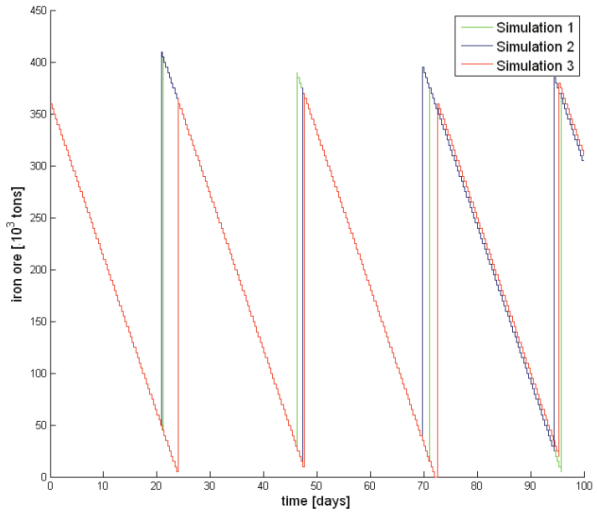

Figure 8. Three simulation results of the iron ore stock with a delivery period of 22.5.

7. Conclusion and Future Work

In this paper the algorithmic approach with respect to the synchronous event handling in the OpenModelica Compiler is stated. The advantage of this procedural method has been clearly demonstrated on some application examples.

Previously, the use of when-equations had to be avoided in the OpenModelica version of the Petri Net library. In order to get correct simulation results the equivalent formulation `if edge(b) then ... end if` had to be employed. This also yields for all other event driven Modelica libraries. Since this work this reformulation is unnecessary. The description of the Petri Net Library for OpenModelica has been optimized, so that now events can be specified with the aid of when-equations.

The current implementation represents the concept, but efficiency can be further improved in the near future. For example, an increase in performance can be achieved by more advanced root finding methods. Furthermore, the efficiency can be enhanced by handling time events separately with a suitable step-size control.

Acknowledgments

The German Ministry BMBF has partially funded this work (BMBF Förderkennzeichen: 01IS09029C) within the ITEA2 project OPENPROD (http://www.openprod.org).

References

[1] Bernhard Bachmann, Peter Aronsson, and Peter Fritzson. Robust initialization of differential algebraic equations. In *Proceedings 5th Modelica Conference, Vienna, Austria*, 2006.

[2] François E. Cellier. *Combined Continuous/Descrete System Simulation by use of digital computers: Techniques and Tools*. PhD thesis, ETH Zürich, 1979.

[3] Harald Dyckhoff and Thomas Stefan Spengler. *Produktionswirtschaft*. Springer-Verlag Berlin Heidelberg, 2005.

[4] Hilding Elmqvist, François E. Cellier, and Martin Otter. Object-oriented modeling of hybrid systems. In *European Simulation Symp.*, pages 31–41, 1993.

[5] Håkan Lundvall, Peter Fritzson, and Bernhard Bachmann. Event handling in the openmodelica compiler and runtime system. Technical report, PELAB,The Institute of Technology, Linköpings universitet, 2008.

[6] Modelica Association. *Modelica - A unified Object-Oriented Language for Physical Systems Modeling Language Specification - Version 3.2*, 03 2010.

[7] Martin Otter, Hilding Elmqvist, and Sven Erik Mattsson. Hybrid modeling in modelica based on the synchronous data flow principle. In *Proceedings of CACSD, Symposium on Computer Aided Control System Desig*, 1999.

[8] Sabrina Pross and Bernhard Bachmann. A petri net library for modeling hybrid systems in openmodelica. In *Proceedings 7th Modelica Conference, Como, Italy*, pages 454–463. The Modelica Association, 2009.

[9] Lena Wunderlich. *Analysis and Numerical Solution of Structured and Switched Differential-Algebraic Systems*. PhD thesis, TU Berlin, 2008.

Towards Efficient Distributed Simulation in Modelica using Transmission Line Modeling

Martin Sjölund[1] Robert Braun[2] Peter Fritzson[1] Petter Krus[2]

[1]Dept. of Computer and Information Science, Linköping University, Sweden,
{martin.sjolund,peter.fritzson}@liu.se
[2]Dept. of Management and Engineering, Linköping University, Sweden,
{robert.braun,petter.krus}@liu.se

Abstract

The current development towards multiple processor cores in personal computers is making distribution and parallelization of simulation software increasingly important. The possible speedups from parallelism are however often limited with the current centralized solver algorithms, which are commonly used in today's simulation environments. An alternative method investigated in this work utilizes distributed solver algorithms using the transmission line modeling (TLM) method. Creation of models using TLM elements to separate model components makes them very suitable for computation in parallel because larger models can be partitioned into smaller independent submodels. The computation time can also be decreased by using small numerical solver step sizes only on those few submodels that need this for numerical stability. This is especially relevant for large and demanding models. In this paper we present work in how to combine TLM and solver inlining techniques in the Modelica equation-based language, giving the potential for efficient distributed simulation of model components over several processors.

Keywords TLM, transmission lines, distributed modeling, Modelica, HOPSAN, parallelism, compilation

1. Introduction

An increasingly important way of creating efficient computations is to use parallel computing, i.e., dividing the computational work onto multiple processors that are available in multi-core systems. Such systems may use either a CPU [11] or a GPU using GPGPU techniques [14, 26]. Since multi-core processors are becoming more common than single-core processors, it is becoming important to utilize this resource. This requires support in compilers and development tools.

3rd International Workshop on Equation-Based Object-Oriented Languages and Tools. October, 2010, Oslo, Norway.

However, while parallelization of models expressed in *equation-based object-oriented* (EOO) languages is not an easily solved task, the increased performance if successful is important. A hardware-in-the-loop real-time simulator using detailed computationally intensive models certainly needs the performance to keep short real-time deadlines, as do large models that take days or weeks to simulate. There are a few common approaches to parallelism in programming:

- No parallelism in the programming language, but accessible via library calls. You can divide the work by executing several processes or jobs at once, each utilizing one CPU core.
- Explicit parallelism in the language. You introduce language constructs so that the programmer can express parallel computations using several CPU cores.
- Automatic parallelization. The compiler itself analyzes the program or model, partitions the work, and automatically produces parallel code.

Automatic parallelization is the preferred way because the users do not need to learn how to do parallel programming, which is often error-prone and time-consuming. This is even more true in the world of equation-based languages because the "programmer/modeler" can be a systems designer or modeler with no real knowledge of programming or algorithms.

However, it is not so easy to do automatic parallelization of models in equation-based languages. Not only is it needed to decide which processor to perform a particular operation on; it is also needed to determine in which order to schedule computations needed to solve the equation system.

This scheduling problem can become quite difficult and computationally expensive for large equation systems. It might also be hard to split the sequence of operations into two separate threads due to dependencies between the equations [2].

There are methods that can make automatic parallelization easier by introducing parallelism over time, e.g. distributing solver work over time [24]. However, parallelism

over time gives very limited speedup for typical ODE systems of equations.

A single centralized solver is the normal approach to simulation in most of today's simulation tools. Although great advances have been made in the development of algorithms and software, this approach suffers from inherent poor scaling. That is, execution time grows more than linearly with system size.

By contrast, distributed modeling, where solvers can be associated with or embedded in subsystems, and even component models, has almost linear scaling properties. Special considerations are needed, however, to connect the subsystems to each other in a way that maintains stability properties without introducing unwanted numerical effects. Technologies based on bilateral delay lines [3], also called transmission line modeling, TLM, have been developed for a long time at Linköping University. It has been successfully implemented in the HOPSAN simulation package, which is currently almost the only simulation package that utilizes the technology, within mechanical engineering and fluid power. It has also been demonstrated in [16] and subsequently in [5]. Although the method has its roots already in the sixties, it has never been widely adopted, probably because its advantages are not evident for small applications, and that wave-propagation is regarded as a marginal phenomenon in most areas, and thus not well understood.

In this paper we focus on introducing distributed simulation based on TLM technology in Modelica, and combining this with solver inlining which further contributes to avoiding the centralized solver bottleneck. In a future paper we plan to demonstrate these techniques for parallel simulation.

Summarizing the main contents of the paper.

- We propose using a structured way of modeling with model partitioning using transmission lines in Modelica that is compatible with existing Modelica tools (Section 6).
- We investigate two different methods to model transmission lines in Modelica and compare them to each other (Section 6).
- We show that such a system uses a distributed solver and may contain subsystems with different time steps, which may improve simulation performance dramatically (Section 7).
- We demonstrate that solver inlining and distributed simulation using TLM can be combined, and that the resulting simulation results are essentially identical to those obtained using the HOPSAN simulation package.

We use the Modelica language [8, 21] and the OpenModelica Compiler [9, 10] to implement our prototype, but the ideas should be valid for any similar language.

2. Transmission Line Element Method

A computer simulation model is basically a representation of a system of equations that model some physical phenomena. The goal of simulation software is to solve this system of equations in an efficient, accurate and robust way. To achieve this, the by far most common approach is to use a centralized solver algorithm which puts all equations together into a differential algebraic equation system (DAE) or an ordinary differential equation system (ODE). The system is then solved using matrix operations and numeric integration methods. One disadvantage of this approach is that it often introduces data dependencies between the central solver and the equation system, making it difficult to parallelize the equations for simulation on multi-core platforms. Another problem is that the stability of the numerical solver often will depend on the simulation time step.

An alternative approach is to let each component in the simulation model solve its own equations, i.e. a distributed solver approach. This allows each component to have its own fixed time step in its solvers. A special case where this is especially suitable is the transmission line element method. Such a simulator has numerically highly robust properties, and a high potential for taking advantage of multi-core platforms [15]. Despite these advantages, distributed solvers have never been widely adopted and centralized solvers have remained the de facto strategy on the simulation software market. One reason for this can perhaps be the rapid increase in processor speed, which for many years has made multi-core systems unnecessary and reduced the priority of increasing simulation performance. Modeling for multi-core-based simulation also requires applications of significant size for the advantages to become significant. With the recent development towards an increase in the number of processor cores rather than an increase in speed of each core, distributed solvers are likely to play a more important role.

The fundamental idea behind the TLM method is to model a system in a way such that components can be somewhat numerically isolated from each other. This allows each component to solve its own equations independently of the rest of the system. This is achieved by replacing capacitive components (for example volumes in hydraulic systems) with transmission line elements of a length for which the physical propagation time corresponds to one simulation time step. In this way a time delay is introduced between the resistive components (for example orifices in hydraulic systems). The result is a physically accurate description of wave propagation in the system [15]. The transmission line element method (also called TLM method) originates from the method of characteristics used in HYTRAN [17], and from Transmission Line Modeling [13], both developed back in the nineteen sixties [3]. Today it is used in the HOPSAN simulation package for fluid power and mechanical systems, see Section 3, and in the SKF TLM-based co-simulation package [25].

Mathematically, a transmission line can be described in the frequency domain by the four pole equation [27]. Assuming that friction can be neglected and transforming these equations to the time domain, they can be described according to equation 1 and 2.

$$p_1(t) = p_2(t-T) + Z_c q_1(t) + Z_c q_2(t-T) \quad (1)$$

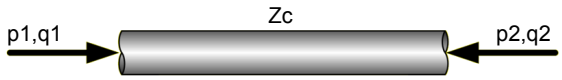

Figure 1. Transmission line components calculate wave propagation through a line using a physically correct separation in time.

$$p_2(t) = p_1(t-T) + Z_c q_2(t) + Z_c q_1(t-T) \quad (2)$$

Here p equals the pressure before and after the transmission line, q equals the volume flow and Zc represents the characteristic impedance. The main property of these equations is the time delay they introduce, representing the communication delay between the ends of the transmission line, see Figure 1. In order to solve these equations explicitly, two auxiliary variables are introduced, see equations 3 and 4.

$$c_1(t) = p_2(t-T) + Z_c q_2(t-T) \quad (3)$$

$$c_2(t) = p_1(t-T) + Z_c q_1(t-T) \quad (4)$$

These variables are called wave variables or wave characteristics, and they represent the delayed communication between the end nodes. Putting equations 1 to 4 together will yield the final relationships between flow and pressure in equations 5 and 6.

$$p_1(t) = c_1 + Z_c q_1(t) \quad (5)$$

$$p_2(t) = c_2 + Z_c q_2(t) \quad (6)$$

These equations can now be solved using boundary conditions. These are provided by adjacent (resistive) components. In the same way, the resistive components get their boundary conditions from the transmission line (capacitive) components.

One noteworthy property with this method is that the time delay represents a physically correct separation in time between components of the model. Since the wave propagation speed (speed of sound) in a certain liquid can be calculated, the conclusion is that the physical length of the line is directly proportional to the time step used to simulate the component, see equation 7. Note that this time step is a parameter in the component, and can very well differ from the time step used by the simulation engine. Keeping the delay in the transmission line larger than the simulation time step is important, to avoid extrapolation of delayed values. This means that a minimum time delay of the same size as the time step is required, introducing a modeling error for very short transmission lines.

$$l = ha = \sqrt{\frac{\beta}{\rho}} \quad (7)$$

Here, h represents the time delay and a the wave propagation speed, while β and ρ are the bulk modulus and the density of the liquid. With typical values for the latter two, the wave propagation speed will be approximately $1000\ m/s$, which means that a time delay of $1\ ms$ will represent a length of $1\ m$. [16]

3. HOPSAN

HOPSAN is a simulation software for simulation and optimization of fluid power and mechanical systems. This software was first developed at Linköping University in the late 1970's [7]. The simulation engine is based on the transmission line element method described in Section 2, with transmission lines (called C-type components) and restrictive components (called Q-type) [1]. In the current version, the solver algorithms are distributed so that each component uses its own local solvers, although many common algorithms are placed in centralized libraries.

In the new version of HOPSAN, which is currently under development, all equation solvers will be completely distributed as a result of an object-oriented programming approach [4]. Numerical algorithms in HOPSAN are always discrete. Derivatives are implemented by first or second order filters, i.e. a low-order rational polynomial expression as approximation, and using bilinear transforms, i.e. the trapetzoid rule, for numerical integration. Support for built-in compatibility between HOPSAN and Modelica is also being investigated.

4. Example Model with Pressure Relief Valve

The example model used for comparing TLM implementations in this paper is a simple hydraulic system consisting of a volume with a pressure relief valve, as can be seen in Figure 2. A pressure relief valve is a safety component, with a spring at one end of the spool and the upstream pressure, i.e., the pressure at the side of the component where the flow is into the component, acting on the other end, see Figure 3. The preload of the spring will make sure that the valve is closed until the upstream pressure reaches a certain level, when the force from the pressure exceeds that of the spring. The valve then opens, reducing the pressure to protect the system.

In this system the boundary conditions are given by a constant prescribed flow source into the volume, and a constant pressure source at the other end of the pressure relief valve representing the tank. As oil flows into the volume the pressure will increase at a constant rate until the reference pressure of the relief valve is reached. The valve then opens, and after some oscillations a steady state pressure level will appear.

A pressure relief valve is a very suitable example model when comparing simulation tools. The reason for this is that it is based on dynamic equations and also includes several non-linearities, making it an interesting component to study. It also includes multiple physical domains, namely hydraulics and mechanics. The opening of a relief valve can be represented as a step or ramp response, which can be analyzed by frequency analysis techniques, for example using bode plots or Fourier transforms. It also includes several physical phenomena useful for comparisons, such as wave propagations, damping and self oscillations. If the complete set of equations is used, it will also produce non-linear phenomena such as cavitation and hysteresis, although these are not included in this paper.

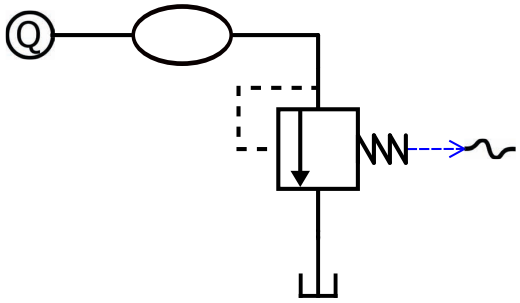

Figure 2. The example system consists of a volume and a pressure relief valve. Boundary conditions is represented by a constant flow source and a constant pressure source.

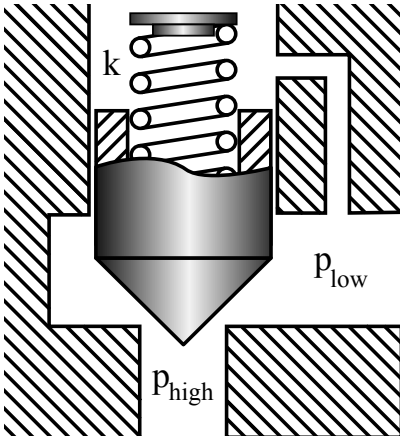

Figure 3. A pressure relief valve is designed to protect a hydraulic system by opening at a specified maximum pressure.

The volume is modeled as a transmission line, in HOPSAN known as a C-type component. In practice this means that it will receive values for pressure and flow from its neighboring components (flow source and pressure relief valve), and return characteristic variables and impedance. The impedance is calculated from bulk modulus, volume and time step, and is in turn used to calculate the characteristic variables together with pressures and flows. There is also a low-pass damping coefficient called α, which is set to zero and thereby not used in this example.

```
mZc = mBulkmodulus/mVolume * mTimestep;
c10 = p2 + mZc * q2;
c20 = p1 + mZc * q1;
c1 = mAlpha*c1 + (1.0-mAlpha)*c10;
c2 = mAlpha*c2 + (1.0-mAlpha)*c20;
```

The pressure relief valve is a restrictive component, known as Q-type. This means that it receives characteristic variables and impedance from its neighboring components, and returns flow and pressure. Advanced models of pressure relief valves are normally *performance oriented*. This means that parameters that users normally have little or no knowledge about, such as the inertia of the spool or the stiffness of the spring are not needed as input parameters but are instead implicitly included in the code. This is however complicated and not very intuitive. For this reason a simpler model was created for this example. It is basically a first-order force equilibrium equation with a mass, a spring and a force from the pressure. Hysteresis and cavitation phenomena are also excluded from the model.

The first three equations below calculate the total force acting on the spool. By using a second-order filter, the x position can be received from Newton's second law. The position is used to retrieve the flow coefficient of the valve, which in turn is used to calculate the flow using a turbulent flow algorithm. Pressure can then be calculated from impedance and characteristic variables according to transmission line modeling.

```
mFs = mPilotArea*mPref;
p1 = c1 + q1*Zc1;
Ftot = p1*mPilotArea - mFs;
x0 = mFilter.value(Ftot);
mTurb.setFlowCoefficient(mCq*mW*x0);
q2 = mTurb.getFlow(c1,c2,Zc1,Zc2);
q1 = -q2;
p1 = c1 + Zc1*q1;
p2 = c2 + Zc2*q2;
```

5. OpenModelica and Modelica

OpenModelica [9, 10] is an open-source Modelica-based modeling and simulation environment, whereas Modelica [21] is an equation-based, object-oriented modeling/programming language. The Modelica Standard Library [22] contains almost a thousand model components from many different application domains.

Modelica supports event handling as well as delayed expressions in equations. We will use those properties later in our implementation of a distributed TLM-style solver. It is worth mentioning that HOPSAN may access the value of a state variable, e.g. `x`, from the previous time step. This value may then be used to calculate derivatives or do filtering since the length of time steps is fixed.

In standard Modelica, it is possible to access the previous value before an event using the `pre()` operator, but impossible to access solver time-step related values, since a Modelica model is independent of the choice of solver. This is where sampling and delaying expressions comes into play. Note that while `delay(x,0)` will return a delayed value, if the solver takes a time step > 0, it will extrapolate information. Thus, it needs to take an infinite number of steps to simulate the system, which means a delay time > 0 needs to be used.

6. Transmission Lines in an Equation-based Language

There are some issues when trying to use TLM in an equation-based language.

TLM has been proven to work well using fixed time steps. In Modelica however, events can happen at any time. When an event is triggered due to an event-inducing expression changing sign, the continuous-time solver is temporarily stopped and a root-finding solution process is started in order to find the point in time where the event

occurs. If the event occurs e.g. in the middle of a fixed time step, the solver will need to take a smaller (e.g. half) time step when restarted, i.e. some solvers may take extra time steps if the specified tolerance is not reached. However, this occurs only for hybrid models. For pure continuous-time models which do not induce events, fixed steps will be kept when using a fixed step solver.

The delay in the transmission line can be implemented in several ways. If you have a system with fixed time steps, you get a sampled system. Sampling works fine in Modelica, but requires an efficient Modelica tool since you typically need to sample the system quite frequently. An example usage of the Modelica `sample()` built-in function is shown below. Variables defined within when-equations in Modelica (as below) will have discrete-time variability.

```
when sample(-T,T) then
  left.c = pre(right.c) + 2 * Zc * pre(
     right.q);
  right.c = pre(left.c) + 2 * Zc * pre(
     left.q);
end when;
```

Modelica tools also offer the possibility to use delays instead of sampling. If you use delays, you end up with continuous-time variables instead of discrete-time ones. The methods are numerically very similar, but because the variables are continuous when you use delay, the curve will look smoother.

```
left.c = delay(right.c + 2 * Zc * right
    .q, T);
right.c = delay(left.c + 2 * Zc * left.
    q, T);
```

Finally, it is possible to explicitly specify a derivative rather than obtaining it implicitly by difference computations relating to previous values (delays or sampling). This then becomes a transmission line without delay, which is a good reference system.

```
der(left.p) = (left.q+right.q)/C;
der(right.p) = der(left.p);
```

Figure 4 contains the results of simulating our example system, i.e., the pressure relief valve from section 4. Figures 5 and 6 are magnified versions that show the difference between our different TLM implementations. The models used to create the Figures, are part of the Modelica package `DerBuiltin` in Appendix A.

If you decrease the delay in the transmission even closer to zero (it is now 10^{-4}), the signals are basically the same (as would be expected). It does however come at a significant increase in simulation times and decreased numerical stability. This is not acceptable if stable real-time performance is desired. We use the same step size as the delay of the transmission line since that is the maximum allowed time step using this method, and better shows numerical issues than a tiny step size.

Due to the nature of integrating solvers, we calculate the value `der(x)`, and use `reinit()` when `der(x)`

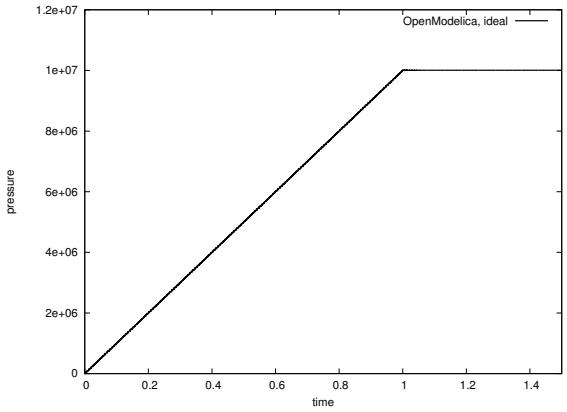

Figure 4. Pressure increases until the reference pressure of 10 MPa is reached, where the relief valve opens.

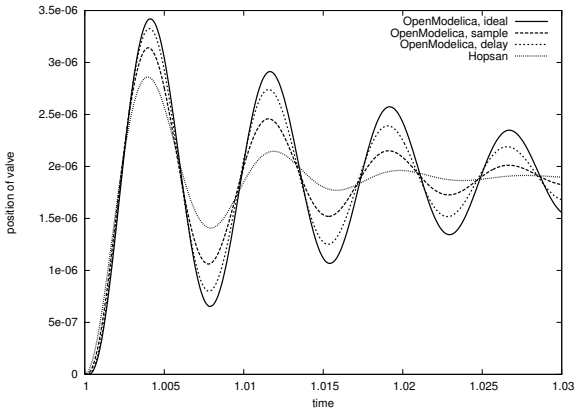

Figure 5. Comparison of spool position using different TLM implementations.

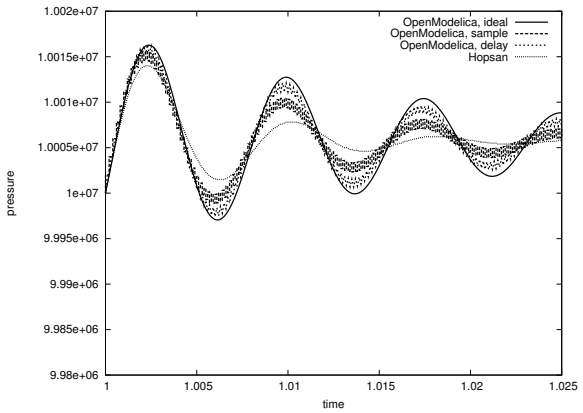

Figure 6. Comparison of system pressure using different TLM implementations.

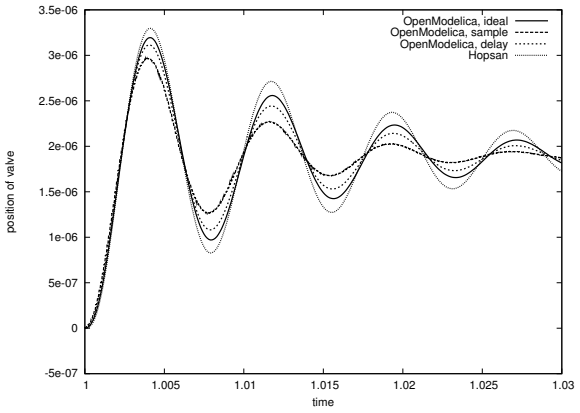

Figure 7. Comparison of spool position with inlined explicit euler.

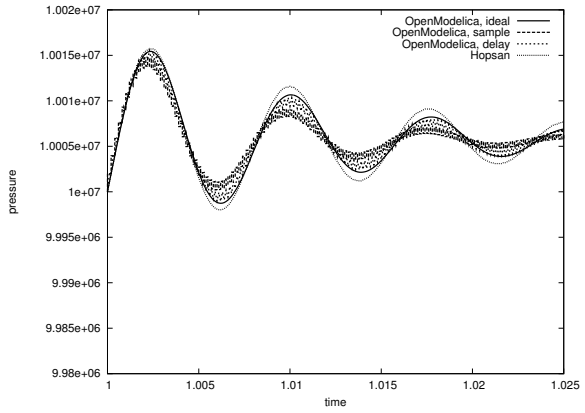

Figure 8. Comparison of system pressure with inlined explicit euler.

Table 1. Performance comparison between different models in the DerBuiltin and DerInline packages.

Method	Builtin (sec)	Inlined (sec)
OpenModelica Delay	0.13	0.40
OpenModelica Ideal	0.04	0.27
OpenModelica Sample	3.65	63.63
Dymola Ideal	0.64	0.75
Dymola Sample	1.06	1.15

changes sign. The OpenModelica DASSL solver cannot be used in all of these models due to an incompatibility with the `delay()` operator (the solver does not limit its step size as it should). DASSL is used together with sampling since the solver does limit its step size if a zero crossing occurs; in the other simulations the Euler solver is used.

Because of these reasons we tried to use another method of solving the equation system, see package `DerInline` in Appendix A. We simply inlined a derivative approximation `(x-delay(x,T))/T` instead of `der(x)`, which is much closer to the discrete-time approximation used in the HOPSAN model. This is quite slow in practice because of the overhead delay adds, but it does implicitly inline the solver, which is a good property for in parallelization.

If you look at Figures 7 and 8, you can see that all simulations now have the same basic shape. In fact, the OpenModelica ones have almost the same values. The time step is still 10^{-4}, which means you get the required behavior even without sacrificing simulation times.

Even in this small example, the implementation using delays has 1 state variable, while the ideal, zero-delay, implementation has 3 state variables. This makes it easier to automatically parallelize larger models since the centralized solver handles fewer calculations. When inlining the `der()` operator, we end up with 0 continuous-time state variables.

Table 1 contains some performance numbers on the models used. At this early stage in the investiagtion the numbers are not that informative for several reasons.

We only made single-core simulations so far. Models that have better parallelism will get better speedups when we start doing multi-core simulations.

The current OpenModelica simulation runtime system implementation does not have special efficient implementations of time events or delayed expressions.

The inlined solver uses `delay` explicitly instead of being an actual inlined solver. This means it needs to search an array for the correct value rather than accessing it directly, resulting in an overhead that will not exist once inline solvers are fully implemented in OpenModelica.

We used the `-noemit` flag in OpenModelica to disable generation of result files. Generating them takes between 20% and 90% of total simulation runtime depending on solver and if many events are generated.

Do not compare the current Dymola [6] performance numbers to OpenModelica. We run Dymola inside a Windows virtual machine, while we run OpenModelica on Linux.

The one thing that the performance numbers really tells you is not to use sampling in OpenModelica until performance is improved, and that the overhead of inlining the derivative using `delay` is a lot lower in Dymola than it is in OpenModelica.

7. Distributed Solver

The implementation using an inlined solver in Section 6 is essentially a distributed solver. It may use different time steps in different submodels, which means a system can be simulated using a very small time step only for certain components. The advantage of such a distributed system becomes apparent in [16].

In the current OpenModelica implementation this is not yet taken advantage of, i.e., the states are solved in each time step regardless.

8. Related Work

Several people have performed work on parallelization of Modelica models [2, 18, 19, 20, 23, 28], but there are still many unsolved problems to address.

The work closest to this paper is [23], where Nyström uses transmission lines to perform model partitioning for parallelization of Modelica simulations using computer clusters. The problem with clusters is the communication

overhead, which is huge if communication is performed over a network. Real-time scheduling is also a bit hard to reason about if you connect your cluster nodes through TCP/IP. Today, there is an increasing need to parallelize simulations on a single computer because most CPUs are multi-core. One major benefit is communication costs; we will be able to use shared memory with virtually no delay in interprocessor communication.

Another thing that is different between the two implementations is the way TLM is modeled. We use regular Modelica models without function calls for communication between model elements. Nyström used an external function interface to do server-client communication. His method is a more explicit way of parallelization, since he looks for the submodels that the user created and creates a kind of co-simulation.

Inlining solvers have also been used in the past to introduce parallelism in simulations [18].

9. Further Work

To progress further we need to introduce replace the use of the `delay()` operator in the delay lines with an algorithm section. This would make the initial equations easier to solve, the system would simulate faster, and it would retain the property that connected subsystems don't depend on each other.

Once we have partitioned a Modelica model into a distributed system model, we will be able to start simulating the submodels in parallel, as described in [12, 16].

Some of the problems inherent in parallelization of models expressed in EOO languages are solved by doing this partitioning. By partitioning the model, you essentially create many smaller systems, which are trivial to schedule on multi-core systems.

To progress this work further a larger more computationally intensive model is also needed. Once we have a good model and inlined solvers, we will work on making sure that compilation and simulation scales well both with the target number of processors and the size of the problem.

10. Conclusions

We conclude that all implementations work fine in Modelica.

The delay line implementation using delays is not considerably slower than the one using the `der()` operator, but can be improved by using for example algorithm sections here instead. Sampling also works fine, but is far too slow for real-time applications. The delay implementation should be preferred over using `der()`, since the delay will partition the whole system into subsystems, which are easy to parallelize.

Approximating integration by inline euler using the delay operator is not necessary to ensure stability although it produces results that are closer to the results of the same simulation in HOPSAN. When you view the simulation as a whole, you can't see any difference (Figure 4).

References

[1] *The* HOPSAN *Simulation Program, User's Manual*. Linköping University, 1985. LiTH-IKP-R-387.

[2] Peter Aronsson. *Automatic Parallelization of Equation-Based Simulation Programs*. Doctoral thesis No 1022, Linköping University, Department of Computer and Information Science, 2006.

[3] D. M. Auslander. Distributed System Simulation with Bilateral Delay-Line Models. *Journal of Basic Engineering*, Trans. ASME:195–200, 1968.

[4] Mikael Axin, Robert Braun, Petter Krus, Alessandro dell'Amico, Björn Eriksson, Peter Nordin, Karl Pettersson, and Ingo Staack. Next Generation Simulation Software using Transmission Line Elements. In *Proceedings of the Bath/ASME Symposium on Fluid Power and Motion Control (FPMC)*, Sep 2010.

[5] JD Burton, KA Edge, and CR Burrows. Partitioned Simulation of Hydraulic Systems Using Transmission-Line Modelling. In *ASME WAM*, 1993.

[6] Dassault Systèmes. Dymola 7.3, 2009.

[7] Björn Eriksson, Peter Nordin, and Petter Krus. HOPSAN, A C++ Implementation Utilising TLM Simulation Technique. In *Proceedings of the 51st Conference on Simulation and Modelling (SIMS)*, October 2010.

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

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

[10] Peter Fritzson et al. Openmodelica 1.5.0 system documentation, June 2010.

[11] Jason Howard et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108 –109, 7-11 2010.

[12] Arne Jansson, Petter Krus, and Jan-Ove Palmberg. Real Time Simulation Using Parallel Processing. In *The 2nd Tampere International Conference on Fluid Power*, 1991.

[13] P. B. Johns and M. A. O'Brien. Use of the transmission line modelling (t.l.m) method to solve nonlinear lumped networks. *The Radio and Electronic Engineer*, 50(1/2):59–70, 1980.

[14] David B. Kirk and Wen-Mei W. Hwu. *Programming Massively Parallel Processors: A Hands-On Approach*. Morgan Kaufmann Publishers, 2010.

[15] Petter Krus. Robust System Modelling Using Bi-lateral Delay Lines. In *Proceedings of the 2nd Conference on Modeling and Simulation for Safety and Security (SimSafe)*, Linköping, Sweden, 2005.

[16] Petter Krus, Arne Jansson, Jan-Ove Palmberg, and Kenneth Weddfelt. Distributed Simulation of Hydromechanical Systems. In *The Third Bath International Fluid Power Workshop*, 1990.

[17] Air Force Aero Propulsion Laboratory. Aircraft hydraulic system dynamic analysis. Technical report, Air Force Aero

Propulsion Laboratory, AFAPL-TR-76-43, Ohio, USA, 1977.

[18] Håkan Lundvall. *Automatic Parallelization using Pipelining for Equation-Based Simulation Languages*. Licentiate thesis No 1381, Linköping University, Department of Computer and Information Science, 2008.

[19] Håkan Lundvall, Kristian Stavåker, Peter Fritzson, and Christoph Kessler. Automatic Parallelization of Simulation Code for Equation-based Models with Software Pipelining and Measurements on Three Platforms. *Computer Architecture News. Special Issue MCC08 – Multi-Core Computing*, 36(5), December 2008.

[20] Martina Maggio, Kristian Stavåker, Filippo Donida, Francesco Casella, and Peter Fritzson. Parallel Simulation of Equation-based Object-Oriented Models with Quantized State Systems on a GPU. In *Proceedings of the 7th International Modelica Conference*, September 2009.

[21] Modelica Association. The Modelica Language Specification version 3.2, 2010.

[22] Modelica Association. Modelica Standard Library version 3.1, 2010.

[23] Kaj Nyström and Peter Fritzson. Parallel Simulation with Transmission Lines in Modelica. In Christian Kral and Anton Haumer, editors, *Proceedings of the 5th International Modelica Conference*, volume 1, pages 325–331. Modelica Association, September 2006.

[24] Thomas Rauber and Gudula Rünger. Parallel execution of embedded and iterated runge-kutta methods. *Concurrency - Practice and Experience*, 11(7):367–385, 1999.

[25] Alexander Siemers, Dag Fritzson, and Peter Fritzson. Meta-Modeling for Multi-Physics Co-Simulations applied for OpenModelica. In *Proceedings of International Congress on Methodologies for Emerging Technologies in Automation (ANIPLA)*, November 2006.

[26] Ryoji Tsuchiyama, Takashi Nakamura, Takuro Iizuka, Akihiro Asahara, and Satoshi Miki. *The OpenCL Programming Book*. Fixstars Corporation, 2010.

[27] T. J. Viersma. *Analysis, Synthesis and Design of Hydraulic Servosystems and Pipelines*. Elsevier Scientific Publishing Company, Amsterdam, The Netherlands, 1980.

[28] Per Östlund. Simulation of Modelica Models on the CUDA Architecture. Master's thesis, Linköping University, Department of Computer and Information Science, 2009.

A. Pressure Relief Valve - Modelica Source Code

```
package TLM
package Basic
connector Connector_Q
  output Real p;
  output Real q;
  input Real c;
  input Real Zc;
end Connector_Q;
connector Connector_C
  input Real p;
  input Real q;
  output Real c;
  output Real Zc;
end Connector_C;
model FlowSource
  Connector_Q source;
  parameter Real flowVal;
equation
  source.q = flowVal;
  source.p = source.c + source.q*
     source.Zc;
end FlowSource;
model PressureSource
  Connector_C pressure;
  parameter Real P;
equation
  pressure.c = P;
  pressure.Zc = 0;
end PressureSource;
model HydrAltPRV
  Connector_Q left;
  Connector_Q right;
  parameter Real Pref = 20000000;
  parameter Real cq = 0.67;
  parameter Real spooldiameter = 0.01;
  parameter Real frac = 1.0;
  parameter Real W = spooldiameter*frac
     ;
  parameter Real pilotarea = 0.001;
  parameter Real k = 1e6;
  parameter Real c = 1000;
  parameter Real m = 0.01;
  parameter Real xhyst = 0.0;
  constant Real xmax = 0.001;
  constant Real xmin = 0;
  parameter Real T;
  parameter Real Fs = pilotarea*Pref;
  Real Ftot = left.p*pilotarea - Fs;
  Real Ks = cq*W*x;
  Real x(start = xmin);
  parameter Integer one = 1;
  constant Boolean useDerInlineDelay;
  Real xfrac = x*Pref/xmax;
  Real v = if useDerInlineDelay then (
     x-delay(x,T))/T else der(xtmp);
  Real a = if useDerInlineDelay then (
     v-delay(v,T))/T else der(v);
  Real v2 = c*v;
  Real x2 = k*x;
  Real xtmp;
equation
  left.p = left.c + left.Zc*left.q;
  right.p = right.c + right.Zc*right.q;
  left.q  = -right.q;
  right.q = sign(left.c-right.c) * Ks *
       (sqrt(abs(left.c-right.c)+((
     left.Zc+right.Zc)*Ks)^2/4) - Ks*(
     left.Zc+right.Zc)/2);
  xtmp = (Ftot - c*v - m*a)/k;
```

```
  x = if noEvent(xtmp < xmin) then xmin
      else if noEvent(xtmp > xmax)
     then xmax else xtmp;
end HydrAltPRV;
end Basic;
package Continuous
extends Basic;
model Volume
  parameter Real V;
  parameter Real Be;
  parameter Real Zc = Be*T/V;
  parameter Real T;
  Connector_C left(Zc = Zc);
  Connector_C right(Zc = Zc);
equation
  left.c = delay(right.c+2*Zc*right.q,T
     );
  right.c = delay(left.c+2*Zc*left.q,T)
     ;
end Volume;
end Continuous;
package ContinuousNoDelay
extends Basic;
model Volume
  parameter Real V;
  parameter Real Be;
  parameter Real Zc = Be*T/V;
  parameter Real T;
  parameter Real C =V/Be;
  Connector_C left(Zc = Zc);
  Connector_C right(Zc = Zc);
protected
  Real derleftp;
equation
  derleftp = (left.q+right.q)/C;
  derleftp = der(left.p);
  derleftp = der(right.p);
end Volume;
end ContinuousNoDelay;
package Discrete
extends Basic;
model Volume
  parameter Real V;
  parameter Real Be;
  parameter Real Zc = Be*T/V;
  parameter Real T = 0.01;
  Connector_C left(Zc = Zc);
  Connector_C right(Zc = Zc);
equation
  when sample(-T,T) then
    left.c = pre(right.c)+2*Zc*pre(
       right.q);
    right.c = pre(left.c)+2*Zc*pre(
       left.q);
  end when;
end Volume;
end Discrete;
end TLM;
```

```
package BaseSimulations
  constant Boolean useDerInlineDelay;
  model HydrAltPRVSystem
    replaceable model Volume =
       TLM.Continuous.Volume;
    parameter Real T = 1e-4;
    Volume volume(V=1e-3,Be=1e9,T=T);
    TLM.Continuous.FlowSource
       flowSource(flowVal = 1e-5);
    TLM.Continuous.PressureSource
       pressureSource(P = 1e5);
    TLM.Continuous.HydrAltPRV hydr(Pref
       =1e7,cq=0.67,spooldiameter
       =0.0025,frac=1.0,pilotarea=5
       e-5,xmax=0.015,m=0.12,c=400,k
       =150000,T=T,useDerInlineDelay=
       useDerInlineDelay);
  equation
    connect(
       flowSource.source,volume.left);
    connect(volume.right,hydr.left);
    connect(
       hydr.right,pressureSource.pressure
       );
  end HydrAltPRVSystem;
  model DelayDassl
    extends HydrAltPRVSystem;
  end DelayDassl;
  model DelayEuler
    extends HydrAltPRVSystem;
  end DelayEuler;
  model NoDelay
    extends HydrAltPRVSystem(redeclare
       model Volume =
       TLM.ContinuousNoDelay.Volume);
  end NoDelay;
  model Disc
    extends HydrAltPRVSystem(redeclare
       model Volume =
       TLM.Discrete.Volume);
  end Disc;
end BaseSimulations;
package DerBuiltin
  extends BaseSimulations(
     useDerInlineDelay = false);
  redeclare class extends
     HydrAltPRVSystem
  equation
    when (hydr.v > 0) then
      reinit(hydr.xtmp, max(
         hydr.xmin,hydr.xtmp));
    elsewhen (hydr.v < 0) then
      reinit(hydr.xtmp, min(
         hydr.xmax,hydr.xtmp));
    end when;
  end HydrAltPRVSystem;
end DerBuiltin;
package DerInline
```

```
  extends BaseSimulations(
     useDerInlineDelay = true);
end DerInline;
```

Compilation of Modelica Array Computations into Single Assignment C for Efficient Execution on CUDA-enabled GPUs

Kristian Stavåker[2] Daniel Rolls[1] Jing Guo[1] Peter Fritzson[2] Sven-Bodo Scholz[1]

[1]School of Computer Science, University of Hertfordshire, United Kingdom,
{d.s.rolls,j.guo,s.scholz}@herts.ac.uk
[2]Programming Environment Laboratory, Department of Computer Science, Linköping University, Sweden
{peter.fritzson,kristian.stavaker}@liu.se

Abstract

Mathematical models, derived for example from discretisation of partial differential equations, often contain operations over large arrays. In this work we investigate the possibility of compiling array operations from models in the equation-based language Modelica into Single Assignment C (SAC). The SAC2C SAC compiler can generate highly efficient code that, for instance, can be executed on CUDA-enabled GPUs. We plan to enhance the open-source Modelica compiler OpenModelica, with capabilities to detect and compile data parallel Modelica for-equations/array-equations into SAC WITH-loops. As a first step we demonstrate the feasibility of this approach by manually inserting calls to SAC array operations in the code generated from OpenModelica and show how capabilities and runtimes can be extended. As a second step we demostrate the feasibility of rewriting parts of the OpenModelica simulation runtime system in SAC. Finally, we discuss SAC2C's switchable target architectures and demonstrate one by harnessing a CUDA-enabled GPU to improve runtimes. To the best of our knowledge, compilation of Modelica array operations for execution on CUDA-enabled GPUs is a new research area.

Keywords Single Assignment C, Modelica, data parallel programming, OpenModelica, CUDA, GPU, SAC

1. Introduction

Mathematical models, derived for example from discretisation of partial differential equations, can contain computationally heavy operations over large arrays. When simulating such models, using some simulation tool, it might be beneficial to be able to compute data parallel array operations on SIMD-enabled multicore architectures.

3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. October, 2010, Oslo, Norway.

One opportunity for data parallel execution is making use of graphics processing units (GPUs) which have in recent years become increasingly programmable. The theoretical processing power of GPUs has far surpassed that of CPUs due to the highly parallel structure of GPUs. GPUs are, however, only good at solving certain problems of data parallel nature. Compute Unified Device Architecture (CUDA) [12] is a software platform for Nvidia GPUs that simplifies the programming of their GPUs.

This paper is about unifying three technologies which will be briefly introduced. These are OpenModelica, SAC2C and CUDA. OpenModelica [14] is a compiler for the object-oriented, equation-based mathematical modeling language Modelica [11, 3]. SAC2C [20] is a compiler for the Single Assignment C [19] functional array programming language for efficient multi-threaded execution. We are interested in using SAC2C's CUDA backend [7] that will enable Modelica models to benefit from NVidia graphics cards for faster simulation. Even without this backend SAC2C can generate highly efficient code for array computations, see for instance [17]. We want to investigate the potential of producing SAC code with OpenModelica where opportunities for data parallelism exist.

Work has been planned to enhance the OpenModelica compiler with capabilities to detect and compile arrays of equations defined in Modelica using for-loops into SAC code. From now on these for-loops will be referred to as for-equations. The overall goal of this investigation is to get a clear overview of the feasibility of this technique before any further work. In this paper we investigate how the OpenModelica runtime system and generated code can be amended to call SAC compiled libraries. This is achieved by manually inserting calls to SAC in the code generated from OpenModelica for array based operations. We also examine the feasibility of rewriting parts of the OpenModelica simulation runtime system in SAC. We perform measurements of this new integrated runtime system with and without CUDA and perform stand-alone measurements of CUDA code generated with SAC2C.

Prior work exists on the generation of parallel executable code from equation-based (Modelica) models [1, 10]. In these publications a task graph of the entire equation

system was first generated and then distributed and scheduled for execution. Ways to inline the solver and pipeline computations were also investigated in [10]. However, no handling of data parallel array operations for the purpose of parallel execution in the context of Modelica was done in any of these publications. For work on parallel differential equation solver implementations in a broader context than Modelica see [15, 9, 16].

The remaining sections of the paper are organized as follows. Section 2 introduces the model we wish to simulate thus giving a clear overview of the case study we will use throughout the rest of the paper. In Section 3 we discuss the OpenModelica compiler and the compilation and simulation of Modelica code and also briefly discuss the proposed changes of the OpenModelica compiler needed for the Modelica to SAC compilation. Section 4 contains a description of SAC, gives SAC code that OpenModelica could eventually produce and gives results and analysis from the first experiments of integrating SAC code with OpenModelica. In Section 5 we give an overview of CUDA, how the SAC2C compiler generates CUDA code, results from experiments and an analysis of how this fits in to the overall goals of this paper. Finally, in Section 6, we draw some conclusions and discuss future work.

2. Case Study

In this section we introduce the Modelica model we wish to simulate. The model has a parameter that can be altered to increase or decrease the default number of state variables. The model introduced here is compiled by the OpenModelica compiler into C++ code and linked with a runtime system. The runtime system will simulate the model in several time steps and each time step involves some heavy array computations. Simulation involves, among other things, computing the values of the time-dependent state variables for each time step from a specified start to stop time. Time-independent algorithm sections and functions are also allowed in Modelica.

2.1 One-dimensional Wave Equation PDE Model

The wave equation is an important second-order linear partial differential equation for waves, such as sound waves, light waves and water waves. Here we study a model of a duct whose pressure dynamics is given by the wave equation. This model is taken from [3] (page 584). The present version of Modelica cannot handle partial differential equations directly since there is only the notion of differentiation with respect to time built into the language. Here we instead use a simple discretisation scheme represented using the array capabilities in Modelica. Research has been carried out on introducing partial differential equations into Modelica, see for instance [18].

The one-dimensional wave equation is given by a partial differential equation of the following form:

$$\frac{\partial^2 p}{\partial t^2} = c^2 \frac{\partial^2 p}{\partial x^2} \ . \tag{1}$$

where $p = p(x,t)$ is a function of both space and time and c is a velocity constant. We consider a duct of length 10 and let $-5 \leq x \leq 5$ describe its spatial dimension. We discretize the problem in the spatial dimension and approximate the spatial derivatives using difference approximations with the approximation:

$$\frac{\partial^2 p}{\partial t^2} = c^2 \frac{p_{i-1} + p_{i+1} - 2p_i}{\Delta x^2} \tag{2}$$

where $p_i = p(x_i + (i-1) \cdot \Delta x, t)$ on an equidistant grid and Δx is a small change in distance. We assume an initial pressure of 1. We get the following Modelica model where the pressure to be computed is represented as a one-dimensional array p of size n, where the array index is the discretized space coordinate along the x-coordinate, and the time dependence is implicit as is common for a continuous-time Modelica variable.

```
1  model WaveEquationSample
2    import Modelica.SIunits;
3    parameter SIunits.Length L = 10 "Length of duct";
4    parameter Integer n = 30 "Number of sections";
5    parameter SIunits.Length dl = L/n "Section length";
6    parameter SIunits.Velocity c = 1;
7    SIunits.Pressure[n] p(each start = 1.0);
8    Real[n] dp(start = fill(0,n));
9  equation
10   p[1] = exp(-(-L/2)^2);
11   p[n] = exp(-(L/2)^2);
12   dp = der(p);
13   for i in 2:n-1 loop
14     der(dp[i]) = c^2 * (p[i+1] - 2 * p[i] + p[i-1]) / dL^2;
15   end for;
16 end WaveEquationSample;
```

On line 1 we declare that our entity should be a model named 'WaveEquationSample'. This model basically consists of two sections: a section containing declarations of parameters and variables (lines 3 to 8) followed by an equation section (lines 9 to 15). A parameter is constant for each simulation run but can be changed between different simulation runs. On line 2 we import the package $SIunits$ from the Modelica standard library.

The two arrays p and dp declared on lines 7 and 8 are arrays of state variables. We can tell that they are arrays of state variables since they occur in derivative expressions in the equation section, thus their values will evolve over time during the simulation run.

The first two equations on lines 10 and 11 state that the first and last pressure value should have a constant value, given by exponent expressions. The third equation on line 12 states that an element in the dp array is equal to the derivative of the corresponding element in the p array. With the present OpenModelica version this equation will result in n scalar equations; we view this kind of equation as an implicit for-equation. The fourth equation on lines 13 to 15 is a for-equation that will result in $n-2$ scalar equations.

3. OpenModelica

OpenModelica is an open source implementation of a Modelica compiler, simulator and development environment for research as well as for educational and industrial purposes. OpenModelica is developed and supported by an international effort, the Open Source Modelica Consortium (OSMC) [14]. OpenModelica consists of a Modelica compiler, OMC, as well as other tools that form an environment for creating and simulating Modelica models.

3.1 The OpenModelca Compilation Process

Due to the special nature of Modelica, the compilation process of Modelica code differs quite a bit from programming languages such as C, C++ and Java. Here we give a brief overview of the compilation and simulation process for generating sequential code. For a more detailed description the interested reader is referred to [2] or [3].

The OpenModelica front-end will first instantiate the model, which includes among other things the removal of all object-oriented structure, and type checking of all equations, statements, and expressions. The output from the OpenModelica front-end is an internal data structure with separate lists for variables, equations, functions and algorithm sections.

For-equations are currently expanded into separate equations. This means that currently each is analysed independently. This is inefficient for large arrays. Thus, for our purpose it would be better if the structure of for-equations is kept throughout the compilation process instead of being expanded into scalar equations.

From the internal data structure executable simulation code is generated. The mapping of time-invariant parts (algorithms and functions) into executable code is performed in a relatively straightforward manner: Modelica assignments and functions are mapped into assignments and functions respectively in the target language of C++. The WaveEquationSample model does not contain any algorithm sections or functions and hence the result of instantiating the WaveEquationSample model in section 2 is one list of parameters, one list of state variables and one list of equations.

The handling of equations is more complex and involves, among other things, symbolic index reduction, topological sorting according to the causal dependencies between the equations and conversion into assignment form. In many cases, including ours, the result of the equation processing is an explicit ordinary differential equation (ODE) system in assignment form. Such a system can be described mathematically as follows.

$$\dot{x} = f(x(t), y(t), p, t) \quad x(t = t_0) = x_0 \; . \tag{3}$$

Here $x(t)$ is a vector of state variables, $\dot{x}$ is a vector of the derivatives of the state variables, $y(t)$ is a vector of input variables, p is a vector of time-invariant parameters and constants, x_0 is a vector of initial values, f denotes a system of statements, and t is the time variable. Simulation corresponds to solving this system with respect to time using a numerical integration method, such as Euler, DASSL or Runge-Kutta.

The output from the OpenModelica back-end consists of a source file containing the bulk of the model-specific code, for instance a function for calculating the right-hand side f in the equation system 3; a source file that contains code for compiled Modelica functions; and a file with initial values of the state variables and of constants/parameters along with other settings that can be changed at runtime.

The ODE equation system in sorted assignment form ends up in a C++ function named `functionODE`. This function will be called by the solver one or more times in each time step (depending on the solver). With the current OpenModelica version, `functionODE` will simply contain a long list of statements originating from the expanded for-equations but work is in progress to be able to keep for-equations throughout the compilation process.

3.1.1 Compilation of WaveEquationSample Model

In this section we illustrate the present OpenModelica compilation process, with the help of the WaveEquationSample model from section 2. In the next section we will discuss how OpenModelica has to be altered if we wish to compile Modelica for-equations into SAC WITH-loops. By instantiating WaveEquationSample we get the following system of equations. All high-order constructs have been expanded into scalar equations and array indices start at 0.

```
p[0] = exp(-(-L / 2.0) ^ 2.0);
p[n-1] = exp(-(L / 2.0) ^ 2.0);
der(p[0]) = p[0];
  .
  .
  .
der(p[n-1]) = p[n-1];
der(dp[0]) = 0;
der(dp[1]) = c^2.0 * ((p[2]+(-2.0*p[1]+p[0])) * dL^-2.0);
  .
  .
  .
der(dp[n-2]) = c^2.0 * ((p[n-1]+(-2.0*p[n-2]+p[n-3])) * dL^-2.0);
der(dp[n-1]) = 0;
```

The above equations corresponds to line 10 to 15 in the original WaveEquationSample model. The rotated ellipsis denotes lines of code that are not shown. The assignments to zero are later removed from the system since they are constant (time independent). From the instantiated code above we can define the following four expressions (where $0 \leq Y \leq n-1$ and $2 \leq X \leq n-3$):

EXPRESSION 3.1.
```
p[Y]
```

EXPRESSION 3.2.
```
c^2.0*((p[2] + (-2.0*p[1] + p[0]))*dL^-2.0)
```

EXPRESSION 3.3.
```
c^2.0*((p[X+1] + (-2.0*p[X] + p[X-1]))*dL^-2.0)
```

EXPRESSION 3.4.
```
c^2.0*((p[n-1]+ (-2.0*p[n-2] + p[n-3]))*dL^-2.0)
```

These expressions correspond roughly to the different types of expressions that occur in the right-hand side of the equation system. The generated code will have the following structure in pseudo code where ... denotes ranges.

```
void functionODE(...) {
  // Initial code
  tmp0 = exp((-pow((L / 2.0), 2.0)));
  tmp1 = exp((-pow(((-L) / 2.0), 2.0)));

  stateDers[0 ... (NX/2)-1] = Expression 3.1;

  stateDers[NX/2] = Expression 3.2;

  stateDers[(NX/2 + 1) ... (NX - 2)] = Expression 3.3;

  stateDers[NX-1] = Expression 3.4;
}
```

The state variable arrays p and dp in the original model have been merged into one array named $stateVars$. There is also a corresponding $stateDers$ array for the derivatives of the state variable arrays. The constant `NX` defines the total number of state variables. The actual generated code (in simplified form) for `functionODE` will look like this:

```
void functionODE(...) {
//--- Initial code ---//
//---
tmp0 = exp((-pow((L / 2.0), 2.0)));
tmp1 = exp((-pow(((-L) / 2.0), 2.0)));

stateDers[0]=stateVars[0 + (NX/2)];
.
.
.
stateDers[(NX/2)-1]=stateVars[((NX/2)-1) + (NX/2)];


stateDers[NX/2] = (c*c) * (stateVars[((NX/2)+1)-(NX/2)]+
                    ((-2.0 * stateVars[(NX/2)-(NX/2)])+
                                        tmp1))/(dL*dL);

stateDers[NX/2 + 1]=(c*c)*(stateVars[((NX/2)+2)-(NX/2)]+
                ((-2.0 * stateVars[((NX/2)+1)-(NX/2)])+
             stateVars[(NX/2)-(NX/2)]))/(dL*dL);
.
.
.
stateDers[NX - 2]= (c*c)*(stateVars[(NX - 1)-(NX/2)]+
                ((-2.0 * stateVars[(NX - 2)-(NX/2)])+
             stateVars[(NX - 3)-(NX/2)]))/(dL*dL);

stateDers[NX-1] = (c*c)*(tmp0 + ((-2.0 *
                stateVars[(NX-1)-(NX/2)])+
          stateVars[(NX-2)-(NX/2)]))/(dL*dL);
//---
//--- Exit code ---//
}
```

This function obviously grows large as the number of state variables increases. Our intention is to rewrite this code and since it is potentially data-parallel we can use the language SAC for this.

3.1.2 Proposed Compilation Process

Several changes to the compilation process have to be performed in order to compile for-equations into SAC WITH-loops. A Modelica for-equation should have the same semantic meaning as before. Right now a for-equation is first expanded into scalar equations. These scalar equations are merged with all other equations in the model and the total set of equations are sorted together. So equations inside the original loop body might end up in different places in the resulting code. This leads to restrictions on what kind of for-equations should be possible to compile into WITH-loops, at least for-equations containing only one equation inside the body should be safe.

In the front-end of the compiler expansion of for-equations into scalar equations should be disabled. We would then get the following code with the WaveEquationSample model.

```
1  p[0] = exp(-(-L / 2.0) ^ 2.0);
2  p[n-1] = exp(-(L / 2.0) ^ 2.0);
3  for i in 0:n-1 loop
4    der(p[i]) = dp[i];
5  end for;
6  der(dp[0]) = 0;
7  for i in 1:n-2 loop
8    der(dp[i]) = c^2.0 * ((p[i+1]+(-2.0*p[i]+p[i-1])) * dL^-2.0);
9  end for;
10 der(dp[n-1]) = 0;
```

New internal data structures that represents a for-equation should be added; one for each internal intermediate form. In the equation sorting phase it might be possible to handle a for-equation as one equation. The equations inside the loop body have to be studied for possible dependencies with other equations outside the loop. The main rule imposed on Modelica models is that there are as many equations as there are unknown variables; a model should be balanced. Checking whether a model is balanced or not can be done by counting the number of equations and unknown variables inside the loop body and adding these numbers with the count from the rest of the model. In the final code generation phase of the compilation process a Modelica for-equation should be mapped into a SAC WITH-loop. This mapping, as long as all checks have proved successful, is relatively straightforward.

4. Single Assignment C

SAC combines a C-like syntax with Matlab-style programming on n-dimensional arrays. The functional underpinnings of SAC enable a highly optimising compiler such as SAC2C to generate high performance code from such generic specifications. Over the last few years several auto-parallelising backends have been researched demonstrating the strengths of the overall approach. These backends include POSIX-thread based code for shared memory multicores [6], CUDA based code for GPGPUs [7] as well as backends for novel many core architectures such as the Microgrid architecture from the University of Amsterdam [8]. All these backends demonstrate the strength of the SAC approach when it comes to auto-parallelisation (see [5, 4, 17] for performance studies).

4.1 Data Parallelism and SAC

Almost all syntactical constructs from SAC are inherited from C. The overall policy in the design of SAC is to enforce that whatever construct looks like C should behave in the same way as it does in C [6].

The only major difference between SAC and C is the support of non-scalar data structures: In C all data structures are explicitly managed by the programmer. It is the programmers responsibility to allocate and deallocate memory as needed. Sharing of data structures is explicit through the existence of pointers which are typically passed around as arguments or results of functions.

In contrast, SAC provides n-dimensional arrays as stateless data structures: there is no notion of pointers whatsoever. Arrays can be passed to and returned from functions in the same way as scalar values can. All memory related issues such as allocations, reuse and deallocations are handled by the compiler and the runtime system. Jointly the compiler and the runtime system ensure that memory is being reused as soon as possible and that array updates are performed in place whenever possible.

The interesting aspect here is that the notion of arrays in SAC actually matches that of Modelica perfectly. Both languages are based on the idea of homogeneously nested arrays, i.e., the shape of any n-dimensional array can always

be described in terms of an n-element shape vector which denotes the extents of the array with respect to the individual axes. All array elements are expected to be of the same element type. Both languages do consider 0-dimensional arrays scalars. The idea of expressing array operations in a combinator style is promoted by both languages.

To support such a combinator style, SAC comes with a very versatile data-parallel programming construct, the WITH-loop. In the context of this paper, we will concentrate our presentation on one variant of the WITH-loop, the `modarray` WITH-loop. A more thorough discussion of SAC is given in [19]. A `modarray` WITH-looptake the general form

with {
($lower_1$ <= *idx_vec* < $upper_1$) : $expr_1$;
⋮
($lower_n$ <= *idx_vec* < $upper_n$) : $expr_n$;
} : **modarray** (*array*)

where *idx_vec* is an identifier and $lower_i$ and $upper_i$, denote expressions for which for any i $lower_i$ and $upper_i$ should evaluate to vectors of identical length. $expr_i$ denote arbitrary expressions that should evaluate to arrays of the same shape and the same element type. Such a WITH-loop defines an array of the same shape as *array* is, whose elements are either computed by one of the expressions $expr_i$ or copied from the corresponding position of the array *array*. Which of these values is chosen for an individual element depends on its location, i.e., it depends on its index position. If the index is within at least one of the ranges specified by the lower and upper bounds $lower_i$ and $upper_i$, the expression $expr_i$ for the highest such i is chosen, otherwise the corresponding value from *array* is taken.

As a simple example, consider the WITH-loop

```
1  with {
2    ([1] <= iv < [4]) : a[iv] + 10;
3    ([2] <= iv < [3]) : 0 * a[iv];
4  } : modarray( a)
```

It increments all elements from index `[1]` to `[3]` by `10`. The only exception is the element at index position `[2]`. As the index `[2]` lies in both ranges the expression associated with the second range is being taken, i.e., it is replaced by `0`. Assuming that `a` has been defined as `[ 0, 1, 2, 3, 4]`, we obtain `[0, 11, 0, 13, 4]` as a result.

Note here, that selections into arrays as well as the WITH-loops themselves are shape-generic, i.e., they can be applied to arrays of arbitrary rank. Assuming that the same WITH-loop is computed with `a` being defined as

```
1  a = [ [  0,  1,  2,  3,  4],
2        [  5,  6,  7,  8,  9],
3        [ 10, 11, 12, 13, 14],
4        [ 15, 16, 17, 18, 19],
5        [ 20, 21, 22, 23, 24]];
```

this would result in an array of the form

```
1      [ [  0,  1,  2,  3,  4],
2        [ 15, 16, 17, 18, 19],
3        [  0,  0,  0,  0,  0],
4        [ 25, 26, 27, 28, 29],
5        [ 20, 21, 22, 23, 24]]
```

Note also, that it was crucial to use `0 * a[iv]` in the second range to make a shape-generic application possible. If we had used `0` instead, the shapes of the expressions would have been detected as incompatible and the application to the array `a` of rank 2 would have been rendered impossible.

4.2 SAC2C, a highly optimising compiler for SAC

SAC2C (see [20] for details) is a compiler for SAC which compiles SAC programs into concurrently executable code for a wide range of platforms. It radically transforms high level programs into efficiently executable C code. The transformations applied do not only frequently eliminate the need to materialise arrays in memory that hold intermediate values but they also attempt to get rid of redundant computations and small memory allocated values as well. Its primary source of concurrency for auto-parallelisation are the WITH-loops. They are inherently data-parallel and, thus, constitute a formidable basis for utilising multi- and many-core architectures. Details of the compilation process can be found in various papers [19, 6].

In order to hook up compiled SAC code into an existing C or C++ application, the SAC2C toolkit also supplies an interface generator named SAC4C. It enables the creation of a dynamically linked library which contains C functions that can be called from C directly.

4.3 Writing OpenModelica Generated Code in SAC

Section 3.1 defined four index expressions for defining the state derivatives array. Their placement into the generated array, `stateDers`, can be represented in SAC as WITH-loop partitions in the following way

```
1  with {
2   ([0] <= iv < [NX/2]) :  Expression 3.1;
3
4   ([NX/2] <= iv <= [NX/2]) :  Expression 3.2;
5
6   ([NX/2] < iv < [NX-1]) :  Expression 3.3;
7
8   ([NX-1] <= iv <= [NX-1]) :  Expression 3.4;
9  } : modarray(stateVars)
```

In legal SAC syntax this can be written as the following.

```
1  with {
2   ([0] <= iv < [NX/2]) :
3       stateVars[iv + (NX/2)];
4
5   ([NX/2] <= iv <= [NX/2]) :
6       (c * c) * (stateVars[(iv+1) - (NX/2)] +
7       ((-2d * stateVars[iv - (NX/2)]) + tmp1))
8       / (dL * dL);
9
10  ([NX/2] < iv < [NX-1]) :
11      (c * c) * (stateVars[(iv+1) - (NX/2)] +
12      ((-2d * stateVars[iv - (NX/2)]) +
13      stateVars[iv-1 - (NX/2)]))
14      / (dL * dL);
15
16  ([NX-1] <= iv <= [NX-1]) :
17      (c * c) *
18      (tmp0 + ((-2d * stateVars[iv - (NX/2)]) +
19      stateVars[iv-1 - (NX/2)])) / (dL * dL);
20 } : modarray(stateVars)
```

The above code defines two single elements within the result array and two large sub-arrays. The equivalent code in OpenModelica-generated C++ and the array the code populates grows linearly with number of state variables.

We modified the OpenModelica-generated C++ code so that instead of computing the ODE system in OpenModelica generated C++, a function call is made in `functionODE` to a SAC function containing the above WITH-loop. Both pieces of code are semantically equivalent. The code was patched so that using a pre-processor macro either the original OpenModelica produced code is invoked or a call is made to a dynamically linked library implemented in SAC that produces the same result.

In the above strategy we make at least one call to SAC in each time step. One alternative to the above strategy of writing this piece of code in SAC is to write the whole solver, or at least the main solver loop, in SAC. We did this for the Euler solver where the main loop looks as follows.

```
1  while (time < stop)
2  {
3    states = states + timestep * derivatives;
4    derivatives = functionODE(states, c, 1, dL);
5    time = time + timestep;
6  }
```

This simple SAC codeblock moves the Euler solver forward in time in steps of `timestep`. Note here that `states` and `derivatives` are arrays and not scalars. The arithmetic operations applied to these arrays are applied to each element with each array. In each time step state variables and state derivatives are calculated. Within each step the SAC version of `functionODE` is invoked.

In the following section we outline experiments where firstly the WITH-loop and secondly the complete SAC Euler solver including the WITH-loop are integrated with OpenModelica.

4.4 Experiments Using SAC with OpenModelica Generated Code

All experiments in this paper were run on CentOS Linux with Intel Xeon 2.27GHz processors and 24Gb of RAM, 32kb of L1 cache, 256Kb of L2 cache per core and 8Mb of processor level 3 cache. SAC2C measurements were run with version 16874 and svn revision number 5625 of OpenModelica was used. C and C++ compilations were performed with Gcc 4.5. The model we used was the WaveEquationSample model introduced in Section 2. The experiments in this section all run sequential code.

Since OpenModelica does not yet have awareness of data parallel constructs inherent in for-equations in the equation section of models it was only feasible to run the compiler for relatively small problem sizes. As mentioned earlier, equations over arrays are expanded into one equation for each element. Even when making some modifications to the OpenModelica code base for the sake of the experiment we were only able to increase the n defined in the original Modelica model to numbers in the low thousands. Anything above this size becomes increasingly infeasible in compile time and resource limits are met at runtime. These problem sizes are big enough still to demonstrate the feasibility of linking SAC modules with C++ code. Computational Fluid Dynamics simulations for instance may however operate on very large arrays.

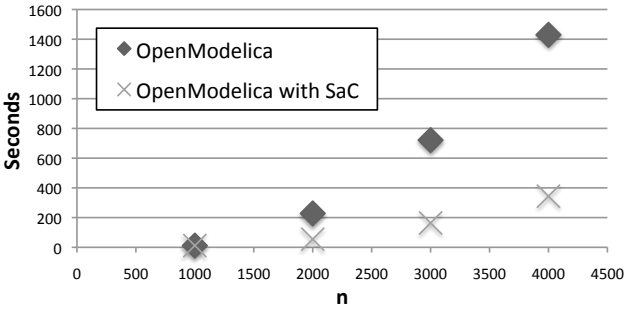

Figure 1. The WaveEquationSample run for different number of sections (n) with functionODE implemented as pure OpenModelica-generated C++ code and as OpenModelica-generated C++ code with functionODE implemented in SAC. Start time 0.0, stop time 1.0, step size 0.002 and without CUDA.

4.4.1 Invoking a SAC WITH-loop from OpenModelica

For our first experiment we altered `functionODE` from the code produced by the OpenModelica compiler so that instead of computing the state derivatives in normal sequential C++ code, a call to SAC code is made. Our SAC code consists primarily of the WITH-loop from Section 4.3. Since in this first experiment only `functionODE` is modified the loop from Section 4.3 is inside the OpenModelica-generated C++ code in this example. The new C++ code that calls the SAC code includes copying of the states array before it is passed to SAC and copying of the array returned by SAC to the state derivatives array in the C++ code. Some copying is required currently because SAC allocates an array with the result. This creates a penalty for the SAC implementation. In a future OpenModelica compiler it is hoped this allocation can be delegated to SAC so that the copying can be removed.

Whilst OpenModelica does an efficient job of taking models and writing code that can make use of different runtime solvers to solve these models, no provisions exist yet for creating highly data parallel code from obviously data parallel models. Our first result shows that if the compiler were to produce SAC code it would be possible to produce code that can feasibly operate on the large arrays that are inevitably required. This in itself can broaden the range of models that OpenModelica could be used to handle.

Figure 1 shows the time taken to simulate the models by running the OpenModelica programs with the two above-described setups for increasing values of n. The experiments were run with the default OpenModelica solver which is the DASSL solver. The simulation was run with timestep 0.002, start time 0 and stop time 1. The results show significant improvements in speed of execution of the SAC implementation already as n raises to values above 1000. For many desired simulations these are relatively small numbers.

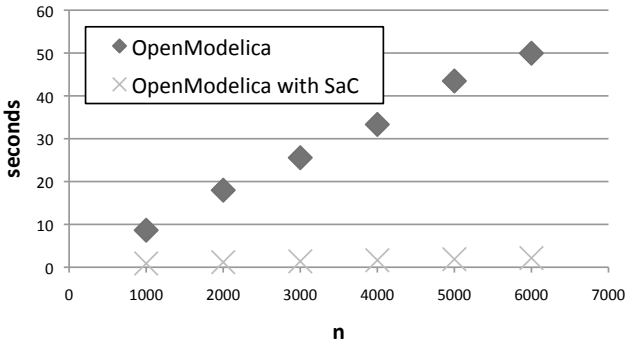

Figure 2. The WaveEquationSample run for different number of sections (n) with functionODE and Euler loop implemented as pure OpenModelica-generated C++ code and as OpenModelica-generated C++ code with functionODE and Euler loop implemented in SAC. Start time 0.0, stop time 100.0, step size 0.002 and without CUDA.

4.4.2 Linking Euler SAC2C Generated Libraries with OpenModelica Generated Code

For a second experiment we used the SAC-implemented Euler solver code consisting primarily of the for-loop in Section 4.3. This code makes calls to the WITH-loop from our previous experiment. For this experiment the OpenModelica Euler solver was used instead of DASSL since the code is simpler but the algorithm is well known and performs a (simple) time step integration, and is hence appropriate for a feasibility study.

This time it was the OpenModelica solver that was patched rather than OpenModelica-generated code. The simulation was run from 0 to 100 seconds with steps of 0.002 seconds. We patched the OpenModelica-solver code so as not to allocate large blocks of memory for the request. This allowed us to run for larger values of n and more time steps. In addition the OpenModelica compiler was patched to not output intermediate time-steps and the SAC code behaves in the same way. As before the SAC version of the code includes additional memory allocation and copying of the data structures passed to and received from the module that will be removed in the future.

Figure 2 shows the time taken to calculate derivatives by running two patched versions of the WaveEquationSample model generated with OpenModelica for increasing values of n.

When using the SAC libraries the performance benefits are significant. We attribute this to OpenModelica's current implementation. Currently globally defined pointers into globally defined arrays are referred to in the code. An array is calculated that is dependent on values in another array and each element of each array is referenced using the globally defined pointers. We believe that the C compiler was unable to efficiently optimise this code where array sizes were large. Improving this will require some changes to the OpenModelica compiler and runtime system which will in themselves certainly have performance benefits.

The model for this experiment operates on a vector of state values. Some computational fluid dynamics applications operate on three-dimensional state spaces. In terms of

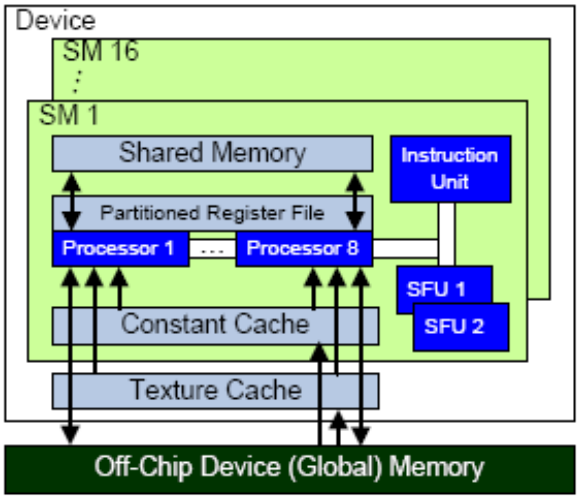

Figure 3. CUDA-enabled GPU hardware architecture.

OpenModelica models these may manifest as three-level nested for-equations. These could map perfectly into SAC where a lot of work [19] has already gone into optimisation for the efficient execution of multi-dimensional arrays taking into account memory access patterns and potential vectorisations. Any future integration of SAC2C into OpenModelica would inevitably make use of these optimisations.

The patches and command line calls used in the experiments in this section can be found in [20].

5. Compute Unified Device Architecture

In recent years, the processing capability of graphics processing units (GPUs) has improved significantly so that they are used to accelerate both scientific kernels and real-world computational problems. Two main features of these architectures render them attractive: large numbers of cores available and their low cost per MFLOP compared to large-scale super-computing systems. Their peak performance figures have already exceeded that of multi core CPUs while being available at a fraction of the cost. The appearance of programming frameworks such as CUDA (Compute Unified Device Architecture) from Nvidia minimises the programming effort required to develop high performance applications on these platforms. To harness the processing power of modern GPUs, the SAC compiler has a CUDA backend which can automatically parallelise WITH-loops and generate CUDA executables. We will briefly introduce the CUDA architecture and programming model before demonstrating the process of compiling the computational kernel of the case study example into a CUDA program.

5.1 Hardware Architecture

Figure 3 shows a high-level block diagram of the architecture of a typical CUDA-enabled GPU. The card consists of an array of Streaming Multiprocessors (SM). Each of these SMs typically contains 8 Streaming Processors (SP).

The organisation of the memory system within CUDA is hierarchical. Each card has a device memory which is common to all streaming multiprocessors, connected through a shared bus, as well as externally. To minimise the contention at that bus, each streaming multiprocessor has a relatively small local memory referred to as Shared Memory shared across all streaming processors. In addition to this, each streaming processor has a set of registers.

5.2 The CUDA Programming Model

The CUDA programming model assumes that the system consists of a host, which is a traditional CPU, and one or more CUDA-enabled GPUs. Programs start running sequentially at the host and call CUDA thread functions to execute parallelisable workloads. The host needs to transfer all the data that is required for computation by the CUDA hardware to the device memory via the system bus. The code that is to be executed by the cores is specified in a function-like unit referred-to as a *kernel*. A large number of threads can be launched to perform the same kernel operation on all available cores at the same time, each operating on different data. Threads in CUDA are conceptually organised as a 1D or 2D grid of blocks. Each block within a grid can itself be arranged as a 1D, 2D or 3D array of cells with each cell representing a thread. Each thread is given a unique ID at runtime which can be used to locate the data upon which they should perform the computation. After each kernel invocation, blocks are dynamically created and scheduled onto multiprocessors efficiently by the hardware.

5.3 Compiling SAC into CUDA

Most of the high level array operations in SAC are a composition of the fundamental language construct - the data parallel WITH-loop. The CUDA backend of the SAC compiler identifies and transforms parallelizable WITH-loops into code that can be executed on CUDA-enabled graphic GPUs. Here we demonstrate the process of compiling the computational kernel of the wave equation PDE model, expressed as a WITH-loop, into equivalent CUDA program (See Figure 4). The compilation is a two-staged process:

- **Phase I**: This phase introduces host-to-device and device-to-host transfers for data arrays referenced in and produced from the WITH-loop. In the example shown, array `stateVars` introduces host-to-device transfers. The final result computed within the GPU, the array $\texttt{stateDers}^{\texttt{D}}$, introduces a device-to-host transfer.
- **Phase II**: This phase lifts computations performed inside each generator as a separate CUDA kernel. In this example, four kernels (i.e. `k1`, `k2`, `k3` and `k4`) are created, each corresponds to one WITH-loop generator.

CUDA kernels are invoked with a special syntax specifying the CUDA grid/block configuration. In the example, each kernel invocation creates a thread hierarchy composed of a one-dimensional grid with one-dimensional blocks in it. Device array variables $\texttt{stateDers}^{\texttt{D}}$ and $\texttt{stateVars}^{\texttt{D}}$, along with scalars(`c`, `dL`, `tmp0`, `tmp1`), are passed as pa-

```
stateDers = with {
       ( [0] <= iv < [256]) :
          stateVars[iv+256];
       ( [256] <= iv < [257]) :
          (c*c)*(stateVars[(iv+1)-256]+
          ((-2f*stateVars[iv-256])+tmp1))/(dL*dL);
       ( [257] <= iv < [511]) :
          (c*c)*(stateVars[(iv+1)-256]+
          ((-2f*stateVars[iv-256])+
          stateVars[(iv-1)-256]))/(dL*dL);
       ( [511] <= iv <= [512]) :
          (c*c)*
          (tmp0*((-2f*stateVars[iv-256])+
          stateVars[(iv-1)-256]))/(dL*dL);
   } : modarray(stateVars);
```

⇓ ⇓ ⇓

```
stateVars^D = host2device( stateVars);
stateDers^D = with {
       ( [0] <= iv < [256]) :
          stateVars^D[iv+256];
       ( [256] <= iv < [257]) :
          (c*c)*(stateVars^D[(iv+1)-256]+
          ((-2f*stateVars^D[iv-256])+tmp1))/(dL*dL);
       ( [257] <= iv < [511]) :
          (c*c)*(stateVars^D[(iv+1)-256]+
          ((-2f*stateVars^D[iv-256])+
          stateVars^D[(iv-1)-256]))/(dL*dL);
       ( [511] <= iv < [512]) :
          (c*c)*
          (tmp0*((-2f*stateVars^D[iv-256])+
          stateVars^D[(iv-1)-256]))/(dL*dL);
   } : modarray(stateVars^D);
stateDers = device2host( stateDers^D);
```

⇓ ⇓ ⇓

```
stateVars^D = host2device( stateVars);
dim3 grid1(16), block1(16);
k1<<<grid1, block1>>>
   ( stateDers^D, stateVars^D);
dim3 grid2(1), block2(16);
k2<<<grid2, block2>>>
   ( stateDers^D, stateVars^D, c, dL, tmp1);
dim3 grid3(16), block3(16);
k3<<<grid3, block3>>>
   ( stateDers^D, stateVars^D, c, dL);
dim3 grid4(1), block4(16);
k4<<<grid4, block4>>>
   ( stateDers^D, stateVars^D, c, dL, tmp0);
stateDers = device2host( stateDers^D);

__global__ void k1( float *stateDers^D,
                    float *stateVars^D) {
   int ivs = blockIdx.x*blockDim.x+threadIdx.x;
   stateDers^D[ivs] = stateVars^D[ivs+256];
}

__global__ void k2( float *stateDers^D,
                    float *stateVars^D) {
                    float c, float dL, float tmp1) {
   int ivs = blockIdx.x*blockDim.x+threadIdx.x+256;
   stateDers^D[ivs] = (c*c)*(stateVars^D[(ivs+1)-256]+
                ((-2f*stateVars^D[ivs-256])+tmp1))
                /(dL*dL);
}

__global__ void k3( float *stateDers^D,
                    float *stateVars^D) {
                    float c, float dL) {
   int ivs = blockIdx.x*blockDim.x+threadIdx.x+257;
   stateDers^D[ivs] = (c*c)*(stateVars^D[(ivs+1)-256]+
                ((-2f*stateVars^D[ivs-256])+
                stateVars^D[(ivs-1)-256]))/(dL*dL);
}

__global__ void k4( float *stateDers^D,
                    float *stateVars^D) {
                    float c, float dL, float tmp0) {
   int ivs = blockIdx.x*blockDim.x+threadIdx.x+511;
   stateDers^D[ivs] = (c*c)*
                (tmp0*((-2f*stateVars^D[ivs-256])+
                stateVars^D[(ivs-1)-256]))/(dL*dL);
}
```

Figure 4. Compiling an example WITH-loop to CUDA.

rameters to the kernels. Each thread running the kernel calculates the linear memory offset of the data being accessed using a set of built-in variables `blockIdx`, `blockDim` and `threadIdx`. This offset is then used to access array `stateVars`^{D}. The final result is written into `stateDers`^{D}.

5.4 Running SAC modules with CUDA

For experiments with CUDA we used CUDA 3.0 and a Tesla C1060. C and C++ compilations were performed with GCC 4.5 except for those invoked by CUDA which used GCC 4.3 since CUDA currently does not support the latest GCC compilers.

In the following experiment more SAC code was written to call the SAC code from Section 4.4.2. When doing this we were able to increase n to much higher numbers to simulate future potential of models with very large numbers of states. Our SAC program simply calls the SAC Euler code from Section 4.4.2 in the same way that OpenModelica called it in our previous experiment. The values used to call the function are explicitly hidden from the compiler so that SAC2C cannot make use of this knowledge when optimising the function. Due to current limitations in older Nvidia cards like the Tesla in our experiment we have used floats not doubles for this experiment. When running on newer cards this modification is not necessary. All parameters used for this experiment match the previous experiment except that we were able to raise n by a factor of one-thousand. It is currently not feasible to raise n this high with code generated with the current OpenModelica compiler since it tries to generate instructions for each extra computation required rather than using a loop and because it allocates large blocks of memory that depend on the size of n. With the exception of these two changes the SAC code matches that of the previous experiment.

Two versions of SAC programs were compared. In one SAC2C was invoked with an option specifying that the code should be highly optimised by the C compiler. In the other an option was added to invoke the SAC2C CUDA backend. The code was linked against both the CUDA libraries and libraries for our SAC module. The SAC code used for the CUDA and non-cuda library versions is identical. As before the patches and command line calls used in the experiment can be found at [20].

The results from the experiment are shown in Figure 5. In both cases time increases linearly as n increases. SAC with CUDA performs significantly better than SAC without CUDA. This is because in each iteration of derivative calculation the input array, i.e. the state variables, is a function of only the `timestep` and the derivative array computed in the previous iterations. This means both arrays can be retained on the GPU main memory without the need of transferring back to the host. The CUDA backend is capable of recognising this pattern and lifting both transfers before and after the WITH-loop (see Figure 4) out of the main stepping loop. With application of this optimisation, each iteration contains pure computation without the overhead of host-device data transfers. Moreover, the large problem sizes provide the CUDA architecture with abundance of data parallelism to exploit to fully utilise the available re-

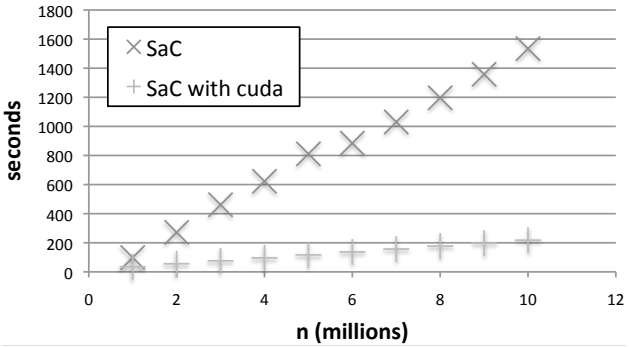

Figure 5. WaveEquationSample state derivative calculations embedded with an Euler loop, both written entirely in SAC, run sequentially and with CUDA for varying number of sections (n). Start time 0.0, stop time 100.0 and step size 0.002.

sources. Given the stencil-like access pattern in the computational kernel, potential data reuse can be exploited by utilizing CUDA's on-chip shared memory. This continued work will further improve the performance.

All experiments with SAC in this paper produced code to either run sequentially or with the CUDA target. For future work we'd like to try the experiments with SAC's already mature pthread target which has already shown positive results [17]. Work is underway to produce a target of C code with OpenMP directives to make use of parallelisation work in C compilers. All these projects and future projects provide interesting potential for future work.

Note that the experiments have demonstrated a benefit when using SAC that materialises for each group of time steps for which intermediate steps are stored. The storing of both states and state derivatives between intermediate time-steps can be a requirement for users of these models and the effect on performance as the number of save points is increased is the next obvious study. When interfacing with SAC there are two ways of doing this. One is to call a SAC function for every group of steps for which a save point is desired. If OpenModelica were to no longer allocate memory for the entire result and instead write the result to file periodically then this method would be the most scalable but it would give SAC2C little opportunity for optimisation. Alternatively one call to the SAC module could be made and SAC could return all desired save points. This would give SAC2C the best chance for optimisation but would have the constraint that the result from the function call would need to be small enough to fit into memory. Ideally a hybrid approach might be desired.

6. Conclusions

Modelica code often contains large arrays of variables and operations on these arrays. In particular it is common to have large arrays of state variables. As of today the OpenModelica compiler has limited support for executing array operations efficiently or for exploiting parallel architectures by, for instance using CUDA-enabled GPU-cards. This is something we hope will be improved in future versions of the compiler and runtime system.

In this work we have investigated ways to make use of the efficient execution of array computations that SAC and SAC2C offer, in the context of Modelica and OpenModelica. We have shown the potential of generating C++ code from OpenModelica that can call compiled SAC binaries for execution of heavy array computations. We have also shown that it is possible to rewrite the main simulation loop of the runtime solver in SAC, thus avoiding expensive calls to compiled SAC binaries in each iteration.

In doing this we have shown the potential for the use of SAC as a backend language to manage the efficient execution of code fragments that the OpenModelica compiler can identify as potentially data parallel. To the best of our knowledge this has not been done with a Modelica compiler before. The integration with SAC allowed experiments to be run with a larger number of state variables than was previously feasible. Moreover, we have shown that the SAC2C compiler can both produce efficient sequential code and produce code targeted for an underlying architecture supporting parallel execution. In this case we exploited the potential of a GPGPU. SAC2C can do this without any changes to the SAC code itself.

Nvidia has recently released the new Fermi architecture [13] which has several improvements which are important in the area of mathematical simulation, a cache hierarchy, more shared memory on the multiprocessors and support for running several kernels at a time.

The next planned stage in this on-going project is to enhance the OpenModelica compiler to pass for-equations through the compiler and to generate SAC source code and compile it automatically.

Acknowledgments

Partially funded by the European FP-7 Integrated Project Apple-core (FP7-215216 — Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs).

Partially funded by the European ITEA2 OPENPROD Project (Open Model-Driven Whole-Product Development and Simulation Environment) and CUGS Graduate School in Computer Science.

References

[1] Peter Aronsson. *Automatic Parallelization of Equation-Based Simulation Programs*. PhD thesis, Linköping University, 2006. Dissertation No. 1022.

[2] Francois Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer, 2006.

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

[4] Clemens Grelck. Implementing the NAS Benchmark MG in SAC. In Viktor K. Prasanna and George Westrom, editors, *16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida, USA*. IEEE Computer Society Press, 2002.

[5] Clemens Grelck and Sven-Bodo Scholz. HPF vs. SAC — A Case Study. In Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismüller, editors, *Euro-Par 2000 Parallel Processing, 6th International Euro-Par Conference (Euro-Par'00), Munich, Germany*, volume 1900 of *Lecture Notes in Computer Science*, pages 620–624. Springer-Verlag, Berlin, Germany, 2000.

[6] Clemens Grelck and Sven-Bodo Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[7] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. Towards Compiling SaC to CUDA. In Zoltan Horváth and Viktória Zsók, editors, *10th Symposium on Trends in Functional Programming (TFP'09)*, pages 33–49. Intellect, 2009.

[8] Stephan Herhut, Carl Joslin, Sven-Bodo Scholz, and Clemens Grelck. Truly nested data-parallelism. compiling sac to the microgrid architecture. In: IFL'09: Draft Proceedings of the 21st Symposium on Implementation and Application of Functional Languages. SHU-TR-CS-2009-09-1, Seton Hall University, South Orange, NJ, USA., 2009.

[9] Matthias Korch and Thomas Rauber. Scalable parallel rk solvers for odes derived by the method of lines. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par*, volume 2790 of *Lecture Notes in Computer Science*, pages 830–839. Springer, 2003.

[10] Håkan Lundvall. *Automatic Parallelization using Pipelining for Equation-Based Simulation Languages, Licentiate thesis 1381*. Linköping University, 2008.

[11] Modelica and the Modelica Association, accessed August 30 2010. `http://www.modelica.org`.

[12] CUDA Zone, accessed August 30 2010. `http://www.nvidia.com/object/cuda_home_new.html`.

[13] Fermi, Next Generation CUDA Architecture, accessed August 30 2010. `http://www.nvidia.com/object/fermi_architecture.html`.

[14] The OpenModelica Project, accessed August 30 2010. `http://www.openmodelica.org`.

[15] Thomas Rauber and Gudula Rünger. Iterated runge-kutta methods on distributed memory multiprocessors. In *PDP*, pages 12–19. IEEE Computer Society, 1995.

[16] Thomas Rauber and Gudula Rünger. Parallel execution of embedded and iterated runge-kutta methods. *Concurrency - Practice and Experience*, 11(7):367–385, 1999.

[17] Daniel Rolls, Carl Joslin, Alexei Kudryavtsev, Sven-Bodo Scholz, and Alexander V. Shafarenko. Numerical simulations of unsteady shock wave interactions using sac and fortran-90. In *PaCT*, pages 445–456, 2009.

[18] Levon Saldamli. *PDEModelica A High-Level Language for Modeling with Partial Differential Equations*. PhD thesis, Linköping University, 2006. Dissertation No. 1016.

[19] Sven-Bodo Scholz. Single Assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[20] Single Assignment C Homepage, accessed august 30 2010. `http://www.sac-home.org`.

An XML representation of DAE systems obtained from continuous-time Modelica models

Roberto Parrotto[1] Johan Åkesson[2,4] Francesco Casella[3]

[1]Master's student - Politecnico di Milano, Italy
roberto.parrotto@gmail.com
[2]Department of Automatic Control, Lund University and Modelon AB, Sweden
johan.akesson@control.lth.se
[3]Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy
casella@elet.polimi.it
[4]Modelon AB, Lund, Sweden

Abstract

This contribution outlines an XML format for representation of differential-algebraic equations (DAE) models obtained from continuous time Modelica models and possibly also from other equation-based modeling languages. The purpose is to offer a standardized model exchange format which is based on the DAE formalism and which is neutral with respect to model usage. Many usages of models go beyond what can be obtained from an execution interface offering evaluation of the model equations for simulation purposes. Several such usages arise in the area of control engineering, where dynamic optimization, Linear Fractional Transformations (LFTs), derivation of robotic controllers, model order reduction, and real time code generation are some examples. The choice of XML is motivated by its de facto standard status and the availability of free and efficient tools. Also, the XSLT language enables a convenient specification of the transformation of the XML model representation into other formats.

Keywords DAE representation, XML design

1. Introduction

Equation-based, object-oriented modeling languages have become increasingly popular in the last 15 years as a design tool in many areas of systems engineering. These languages allow to describe physical systems described by differential algebraic equations (DAE) in a convenient way, promoting re-use of modeling knowledge and a truly modular approach. The corresponding DAEs can be used for different purposes: simulation, analysis, model reduction, optimization, model transformation, control system synthesis, real-time applications, and so forth. Each one of these activities involves a specific handling of the corresponding differential algebraic equations, by both numerical and symbolic algorithms. Moreover, specialized software tools which implement these algorithms may already exist, and only require the equations of the model to be input in a suitable way.

The goal of this paper is to define an XML-based representation of DAE systems obtained from object-oriented models written in Modelica [16], which can then be easily transformed into the input of such tools, e.g. by means of XSLT transformations.

The first requirement of this system representation is to be as close as possible to a set of scalar mathematical equations. Hierarchical aggregation, inheritance, replaceable models, and all kinds of complex data structures are a convenient means for end-users to build and manage models of complex, heterogeneous physical systems, but they are inessential for the mathematical description of its behavior. They will therefore be eliminated by the Modelica compiler in the flattening process before the generation of the sought-after XML representation. However, the semantics of many Modelica models is in part defined by user-defined functions described by algorithms working on complex data structures. It is therefore necessary to describe Modelica functions conveniently in this context.

The second requirement of the representation is to be as general as possible with respect to the possible usage of the equations, which should not be limited to simulation. A few representative examples include:

- off-line batch simulation;
- on-line real-time simulation;
- dynamic optimization [3];
- transformation of dynamic model with nonlinearities and/or uncertain parameters into Linear Fractional Representation formalism [7];
- linearization of models and computation of transfer functions for control design purposes;

3rd International Workshop on Equation-Based Object-Oriented Languages and Tools. October, 2010, Oslo, Norway.

- model order reduction, i.e., obtaining models with a smaller number of equations and variables, which approximate the input-output behavior around a set of reference trajectories [8];
- automatic derivation of direct/inverse kinematics and advanced computed torque and inverse dynamics controllers in robotic systems [6].

From this point of view, the proposed XML representation could also be viewed as a standardized interface between multiple Modelica front-end compilers and multiple symbolic/numerical back-ends, each specialized for a specific purpose.

In addition, the XML representation could also be very useful for treating other information concerning the model, for example using an XML schema (DTD or XSD) for representing the simulation results, or the parameter settings. In those cases, using a well accepted standard will result in great benefits in terms of interoperability for a very wide spectrum of applications.

Previous efforts have been registered to define standard XML-based representations of Modelica models. One idea, explored in [14, 11], is to encode the original Modelica model using an XML-based representation of the abstract syntax tree, and then process the model through, e.g., XSLT transformations. Another idea is to use an XML database for scalable and database-friendly parameterization of libraries of Modelica models [17, 15].

The goal of this paper is instead to use XML to represent the system equations at the lowest possible level for further processing, leaving the task of handling aggregated models, reusable libraries etc. to the object-oriented tool that will eventually generate the XML representation of the system. In particular, this paper extends and complements ideas and concepts first presented in [5]. A similar approach has been followed earlier by [4], but has apparently remained limited to the area of chemical engineering applications.

The paper is structured as follows: in Section 2, a definition of the XML schema describing a DAE system is given. Section 3 briefly describes a test case in which a model is exported from JModelica.org platform and imported in the tool ACADO in order to solve an optimization problem. Section 4 ends the paper with concluding remarks and future perspectives.

2. XML schema representation of DAE systems

The goal of the present work is to define a representation of a DAE system which can be easily transformed into the input format of different purpose tools and then reused. A representation as close as possible to the mathematical formulation of equations is a solution general enough to be imported from the most of the tools and neutral with respect of the possible usage. For this reason concepts such as aggregation and inheritance proper of equation based object-oriented models have to be avoided in the representation.

A DAE system consists of a system of differential algebraic equations and it can be expressed in vector form as:

$$F(\dot{x}, x, u, w, t, p) = 0 \tag{1}$$

where $\dot{x}$ are the derivatives of the states, x are the states, u are the inputs, w are the algebraic variables, t is the time and p is the set of the parameters.

The schema does not enforce the represented DAEs to have index-1, but this would be the preferable case, so that the x variables can have the proper meaning of states, i.e., it is possible to arbitrarily select their initial values. Preferring the representation of models having index 1 is acceptable considering that most of the applications for DAE models require an index-1 DAE as input. In addition, in case the equations of the original model have higher index, usually an index-1 DAE can be obtained by index reduction, so the representation of index-1 DAEs doesn't drastically restrict the possible applications range.

The formulation provided in equation (1) is very general and useful for viewing the problem as one could see it written on the paper, but it is not directly usable for inter-tools exchange of models. It is then necessary to provide a standardized mathematical representation of the DAE systems that relies on a standard technology: this justifies the choice of the XML standard as a base for our representation. Hence, a formulation that better suits with our goal is proposed.

Given the sets of the involved variables

- $\dot{x} \in \mathbb{R}^n$: vector of time-varying state derivative variables
- $x \in \mathbb{R}^n$: vector of time-varying state variables
- $u \in \mathbb{R}^m$: vector of time-varying input variables
- $w \in \mathbb{R}^r$: vector of time-varying algebraic variables
- $p \in \mathbb{R}^k$: vector of bound time invariant variables (parameters and constants)
- $q \in \mathbb{R}^l$: vector of unknown time invariant variables (unknown parameters)
- $t \in \mathbb{R}$: time variable

it is possible to define the three following different subsets for the equations composing the system. The system of dynamic equations is given by

$$F(x, \dot{x}, u, w, p, q, t) = 0 \tag{2}$$

where $F \in \mathbb{R}^{n+r}$. These equations determine the values of all algebraic variables w and state variable derivatives $\dot{x}$, given the states x, the inputs u, the parameters p and q, and the time t. The parameter binding equations are given by

$$p = G(p) \tag{3}$$

where $G \in \mathbb{R}^k$. The system of parameter binding equations is assumed to be acyclic, so that it is possible to compute all the parameters by suitably re-ordering these equation into a sequence of assignments, e.g. via topological sorting. The

DAE initialization equations are given by

$$H(x, \dot{x}, u, w, p, q) = 0. \quad (4)$$

Ideally, for index-1 systems, $H \in \mathbb{R}^{n+l}$, i.e., H provides $n+q$ additional equations, yielding a well posed initialization problem with $2n + r + l$ unknowns and $2n + r + l$ equations. The initialization system is thus obtained by aggregating the dynamic equations (2) and the initialization equations (4) and determines the values of the states, state derivatives, algebraic variables and free parameters at some initial time t_0.

2.1 General design issues

The main goal is to have a schema:

- neutral with respect of the model usage;
- easy to use, read and maintain;
- easy to extend.

To achieve the first goal a representation as close as possible to the mathematical one of the DAE is required, as discussed in the previous section. To achieve the other required properties, a design based on modularity yields a result easier to read and extend. The proposed design provides one different vocabulary (namespace) for every section of the schema. In this way, if a new section will be required, for example to represent information useful for a special purpose, and a new module can be added without modify the base schema.

The Functional Mock-up Interface for Model Exchange 1.0 (FMI 1.0)[12] has been chosen as a starting point for the schema, with the main advantage of basing the work on an already accepted standard for model exchange. The FMI 1.0 specification already provides a schema containing a representation of the scalar variables involved in the system. This schema has been extended according to our goals by adding a qualified name representation for the variable identifiers, and by appending a specification of the DAE system.

The new modules composing the schema with the corresponding namespace prefixes are:

- expressions module (`exp`)
- equations module (`equ`)
- functions module (`fun`)
- algorithms module (`fun`)

All these modules, whose detailed information are given in the next paragraphs, are imported in the FMI schema, to construct the composite schema.

2.2 FMI schema and variable definitions

The FMI standard is a result of the ITEA2 project MODELISAR. The intention is that dynamic system models of different software systems can be used together for simulation. The FMI defines an interface to be implemented by an executable called FMU (Functional Mock-up Unit). The FMI functions are called by a simulator to create one or more instances of the FMU, called models, and to run these models, typically together with other models. An FMU may either be self-integrating (co-simulation) or require the simulator to perform numerical integration. Alternatively, tools may be coupled via co-simulation with network communication. The intention is that a modeling environment can generate C-code of a dynamic system model that can be utilized by other modeling and simulation environments. The model is then distributed in packages containing the C-code of the dynamic system, an XML-file containing the definition of all variables in the model, and other model information. For the present work, the FMI XML schema for description of model variables has been reused and extended.

The FMI XML schema already provides elements and attributes to represent general information about the model, such as name, author, date, generating tool, vendor annotations, but the core is the representation of the scalar variables defined in the model. It is important to notice that the FMI project is developed for the exchange of models for simulation purpose only, and not all the information present in the schema should be used in our case. Thus it is necessary to point out how to correctly use it for the purposes of this work. Firstly, the FMI schema allows the definition of Real, Integer, Boolean, String and Enumeration scalar variables. In our case, the equations (2) - (4) are all real-valued, and all time varying variables are real variables. The scalar variables definition provided by the FMI XML schema also includes attributes describing the causality (input, output, internal) and variability (constant, parameter, discrete, continuous) of the variable. Since our representation is concerned with continuously time varying DAEs only, then the definition of discrete variables should not be allowed. A full documentation of the FMI XML schema is available in [12].

The proposed representation should be neutral with respect to the application context. This also means that variable identifiers should be represented in a general way. It may happen that the tool exporting the model accepts identifiers with special characters that the importing tool does not allow. Furthermore, in the definition of user-defined functions (see a detailed discussion in Section 2.4) more complex types than scalar variables, such as array and records, are allowed. The index of an array can be a general expression, and representing the array's element by a string, e.g. `"x[3*5]"`, would require to write an ad-hoc parsing module in the importing tools. In the same manner the exporting tool can support a notation to describe array subscripts or record fields that is different from the one used by the importing tool.

For all these reasons a structured representation for qualified names, that includes only the necessary information and avoid language dependent notations is introduced. A complex type `QualifiedName` is then defined and it will be used as a standard representation for names in all the schema. The `QualifiedName` complex type expects that the identifier is broken in a list of parts, represented by `QualifiedNamePart` elements. `QualifiedNamePart` holds a string attribute "name"

and an optional element `ArraySubscripts`, to represent the indices of the array element. `ArraySubscripts` elements provide a list of elements, one for each index of the array (e.g. a matrix has an `ArraySubscripts` element with two children). Each index is generally an expression, represented by `IndexExpression`, but usually languages support definition of array variables with undefined dimensions, represented by an `UndefinedIndex` element. Conventionally, the first element of an array has index 1. In the proposed representation, definition of array variables is allowed only in user-defined functions.

Hence, the original representation of scalar variables provided by the FMI XML schema is extended in order to support the definition of variable names as qualified names, that will be the standard representation of identifiers in the whole schema.

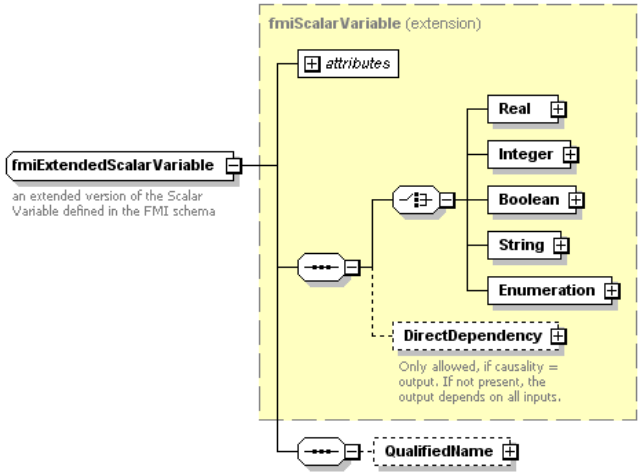

Figure 1. Scalar variable definition extended from the original FMI definition

2.3 Expressions

All the expressions are collected in the `exp` namespace. The elements in the `exp` namespace represent all the mathematical scalar expressions of the system:

- basic algebraic operators like `Add`, `Mul`, `Sub`, `Div` and the factor function `Pow`.
- Basic logical comparison operators like >, >=, <, <=, ==.
- Basic logical operators like `And`, `Or` and `Not`.
- Built-in trigonometric functions (`Sin`, `Cos`, `Tan`, `Asin`, `Acos`, `Atan`, `Atan2`), hyperbolic functions (`Sinh`, `Cosh`, `Tanh`), the exponential function (`Exp`), the logarithmic functions (`Log`, `Log10`), the square root function (`Sqrt`), the absolute value function `Abs`, the sign function `Sign`, the `Min` and the `Max` function.
- The derivative operator `Der`.
- Function calls referring to user-defined functions.
- Variable identifiers, including the time variable.
- Real, integer, boolean, string literals.

In addition to the previous basic expressions, some special non-scalar expressions are included in the `exp` namespace: `Range`, `Array`, `UndefinedDimension` and `RecordConstructor`.

The `Range` element defines an interval of values and it can be used only in for loop definitions, inside algorithms of user-defined functions or as an argument of array constructors.

Array variable definitions and uses are allowed only within user-defined functions. It is possible to use the element `UndefinedDimension` in array variable definitions when the dimension is not known a priori. The `Array` element can be used as a constructor of an array of scalar variables in the left hand side of user-defined function call equations. Multidimensional arrays can be built by iteratively applying the one-dimensional array constructor.

As for arrays, record variables can be defined and used only in user-defined functions. The `RecordConstructor` element can be used in the left hand side of user-defined function calls, where it should be seen as a collection of scalar elements. Both record variables used in functions and record constructors used in the left hand side of equations should be compatible with a given definition of record type. The `RecordList` element, that is referenced in the main schema, should contains the definition of all the records used in the XML document, each one stored in a different `Record` element. All the elements and complex types relevant to records definition are stored in the `fun` namespace, since they are mostly related to the use of functions.

The detailed explanation of how to use `Array` and `RecordConstructor` in the left hand side of a user-defined function call equations is given in Section 2.4.

In the design of the schema, whenever a valid element is supposed to be a general expression, a wildcard element in the `exp` namespace is used, in order to simplify the representation extensibility. As a result, when a new expression is needed, it is sufficient to create a new element in the `exp` vocabulary and it will be automatically available in all the rest of the schema.

2.4 Functions

A function is a portion of code which performs a procedural computation and is relatively independent of the remaining model. A function is defined by:

- input variables, possibly with default values
- output variables
- protected variables (i.e. variables visible only within the context of the function)
- an algorithm that computes outputs from the given inputs, possibly using protected variables.

The algorithm can operate on structured variables such as arrays and records, e.g. by means of `for` loops. Differently from the variables used in equations, which can always be expressed as scalars, it is then required that input, output and protected variables of a function can also be arrays or

records, so that the algorithm can keep its original structure.

Whereas in the formulation of the equations (2) - (4) only scalar variables are involved, a detailed discussion on the use of calls for any possible cases in which the function involves non-scalar inputs or outputs is then required.

Function calls with non-scalar inputs

If an input of a function is not a scalar, it will be represented by keeping its structure, possibly using array or record constructors, but populating it with its scalar elements. In this way, it is possible to keep track of the structure of the arguments, which can then be mapped to efficient data structures in the target code.

For example, given the following definition of a record R and a function $F1$:

```
Record R
  Real X;
  Real Y[3];
End R;

Function F1
  Input R X;
  Output Real Y;
End F1;
```

A function call to $F1$ may be used as an expression in this case, since the function has only one scalar output.

```
F(R(x,{y[1], y[2], y[3]})) - 3 = 0
```

where x, y[1], y[2], y[3] are real scalar variables, `R(args)` denotes a constructor for the `R` record type, and `{var1, var2,...,varN}` represents an array constructor.

Function calls with a single non-scalar output

Auxiliary variables can be introduced to handle this case, making it possible to always have scalar equations and at the same time avoiding unnecessary duplicated function calls.

Considering the following definition of the function $F2$:

```
Function F2
  Input Real X;
  Output Real Y[3];
End F2;
```

The equation $x + F(y) * F(z) = 0$ (a scalar product) can be mapped to:

```
({aux1, aux2, aux3}) = F(y);
({aux4, aux5, aux6}) = F(z);
x + aux1*aux4 + aux2*aux5 + aux3*aux6 = 0
```

where y and z are real scalar variables.

Similarly the equation $y+F(x)-F(-3*x) = 0$, where y is an array of three real elements is mapped to:

```
({aux1, aux2, aux3}) = F(x);
({aux4, aux5, aux6}) = F(-3*x);
y[1] + aux1 - aux4 = 0;
y[2] + aux2 - aux5 = 0;
y[3] + aux3 - aux6 = 0;
```

This strategy also applies to arguments using records, or combinations of arrays and records.

Auxiliary variables are here treated as all the other scalar variables, including their declaration.

Function calls with multiple outputs

In this case, the function calls can be invoked in the following form:

$$(out1, out2, ..., outN) = f(in1, in2, ...inM) \quad (5)$$

where `out1, out2, ..., outN` can be scalar variable identifiers, array or record constructors populated with scalar variables identifiers, empty arguments, or any possible combination of these elements. So, it is not possible to write any expression on the left-hand side, nor to put the equation in residual form. Rather, this construct is used as a mechanism dedicated to handle function calls with multiple outputs while preserving the scalarized structure of system of equations.

Function with multiple outputs, cannot be used in expressions.

Given the following definition of a record type `R1` and a function `F3`:

```
Record R1
  Real X;
  Real Y[2,2];
End R1;

Function F3
  input Real x;
  output Real y;
  output R1 r;
End F3;
```

an example of call to the function $F3$ is

```
(var1,R1(var2,{{var3,var4},{var5,var6}}))
  = F1(x)
```

where `x`, `var1`, `var2`, `var3`, `var4`, `var5`, `var6`, `var7` are real scalar variables.

The proposed representation of function calls is preferable to a full scalarization of the arguments,which does not preserve any structure, and thus would require multiple implementations for the same function, e.g. if it is called in many places with different array sizes of the inputs. This solution would lead to less efficient implementations in most target languages.

Concerning the XML schema implementation, all the elements and complex types regarding user-defined function are collected in the `fun` namespace.

The main element of the `fun` namespace is `Function`, which contains the whole definition of the function, including the name, three lists of variables (respectively outputs, inputs and protected variables), the algorithm and, optionally, the definition of inverse and derivative functions. `OutputVariable`, `InputVariable` and `ProtectedVariable` elements are defined by means of the `FunctionVariable` complex type.

It is allowed, but not mandatory, to embed the definition of possible inverse and derivative functions in the `InverseFunction` and `DerivativeFunction` elements of a function definition. The information stored in

these two elements could be used for optimization purposes by the importing tool.

The elements and complex types used in the description of algorithms are defined in a different schema module than the `Function` element, but also under the `fun` namespace. The allowed statements are:

- assignments
- conditional `if` statements with `elseif` and `else` branches
- `while` and `for` loops
- function calls of user-defined functions

2.5 Equations

Complex types and elements related to the equations of the DAE system are collected under the `equ` namespace.

Once the expressions have been defined, mapping the mathematical formulation of the binding equations (3) to the XML schema is straightforward. In the `equ` namespace a complex type `BindingEquation` is defined. It provides an element `Parameter` of `QualifiedName` type that represents the left hand side of the equation, and a `BindingExp` element that represents the right hand side of the equation. An element `BindingEquations` represents the set of all the binding equations and it accepts a list, possibly empty, of `BindingEquation` elements defined as `BindingEquation` complex type.

Equations in residual form are represented by the complex type `AbstractEquation`. This type of equations provide a subtraction node to represent an equation in $exp1 - exp2 = 0$ form.

The initial equations set (4) is represented by the element `InitialEquation`, that collects a list, possibly empty, of `Equation` elements defined as `Abstract-Equation` complex type.

The set of dynamic equations (2) is mapped to the `DynamicEquation` element. According to the considerations expressed in Section 2.4, equations resulting from a call to a function with multiple outputs are not suitable for representation in residual form. Thus a complex type for mapping an equation of the form (5) is given by the complex type `FunctionCallEquation`. The left hand side of the equation (5) is represented by a set of `OutputArgument` elements, defined by the `FunctionCallLeft` complex type, that can have as children scalar variable identifiers, array or record constructors populated with scalar variables identifiers, empty arguments, or any combination thereof. The right hand side is a `FunctionCall` element. It is important to notice that this element represents a set of scalar equations, one for each scalar variable in the left hand side (except for empty arguments).

Hence, the `DynamicEquations` elements contain a list of `Equation` elements of `AbstractEquation` type, which represent equations in residual form, and `FunctionCallEquation` elements, which represent equations on the form (5).

The `BindingEquation`, `DynamicEquations` and `InitialEquations` elements are directly referenced and used in the main schema.

2.6 Overall result and extensibility

Having defined the new modules, they are imported in the FMI XML schema. The elements required to be visible in the main schema are then directly referenced. The resulting overall DAE representation is given in Figure 2. In the same way, the schema could be extended by adding new information, possibly according to special purposes, developing a new separate module and referencing the main element in the XML schema, without changing the current definitions. An extension of the schema representing the formulation of optimization problems has been already developed.

Figure 2. Overall structure of the DAE XML schema

3. Test implementation

As a first implementation, an XML export module has been implemented in the JModelica.org platform [13, 1], in order to generate XML documents representing DAE systems derived from Modelica models valid with respect of the proposed schema. In addition, an extension of the XML schema for representing optimization problems has been developed and the XML code export has been implemented in the Optimica compiler, which is part of the JModelica.org compiler.

The extension for optimization problems provides elements for the representation of the objective function, the interval of time on which the optimization is performed and the constraints. The boundary values of the optimization interval, t_0 and t_f , can either be fixed or free. The constraints

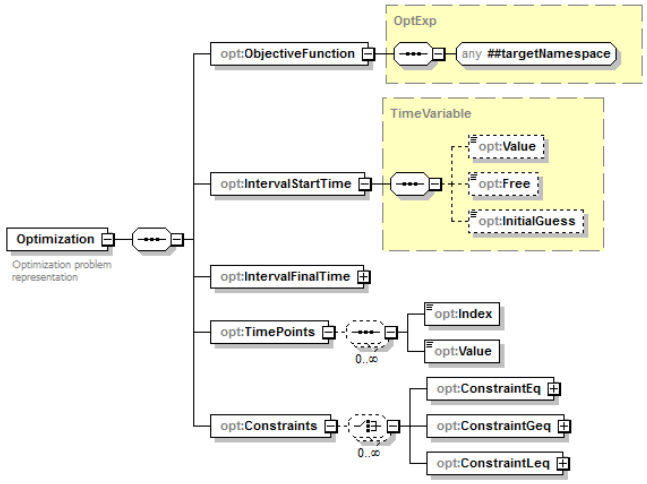

Figure 3. XML schema extension for optimization problems

include inequality and equality path constraints, but also point constraints are supported. Point constraints are typically used to express initial or terminal constraints, but can also be used to specify constraints for time points in the interior of the interval. An overall view of the extension is given in Figure 3.

Before exporting a Modelica model in XML format, a pre-processing phase is necessary. Firstly, the model should be "flattened" in order to have the system resulting from both equations of every single component and the connection equations. In this system the variables should all be scalarized. Parameters that are used to define array sizes, sometimes referred to as structural parameters, are evaluated in the scalarization and can not be changed after generation of the XML code.

Furthermore, the functions should be handled as explained in section 2.4. If the system is a higher index DAE, an index reduction operation can be performed, to obtain the final index-1 DAE.

It is interesting to notice how the XML schema has been easily implemented in the compiler. In fact, the structured representation of an XML schema is suitable to be mapped to the abstract syntax tree of the flattened model. A function that writes the corresponding XML representation has been implemented in each node class exploiting the aspect-oriented design allowed by JastAdd, used for the JModelica.org compiler construction [2]. Hence, traversing the abstract syntax tree of the flattened model is equivalent to traversing the XML document representing the same model. In the same way, importing an XML representation of the model could be done traversing the XML document and building a node of the syntax tree corresponding to each XML node. This remains to be done.

For the test case, the ACADO toolkit [10] has been chosen as importing tool. ACADO Toolkit is a software environment, not specifically related to Modelica, that collects algorithms for automatic control and dynamic optimization. It provides a general framework for using a great variety of algorithms for direct optimal control, including model predictive control, state and parameter estimation and robust optimization.

The Van der Pol oscillator model (it can be found with further explanations in [13]) has been exported from the JModelica.org platform and imported into ACADO. The goal is to solve an optimal control problem with respect to the constraint $u <= 0.75$ acting on the control signal while minimizing a quadratic cost function in the interval $t \in [0, 20]$.

The optimal control problem has been parameterized as an non-linear problem using direct multiple shooting (with condensing) and solved by an SQP method (sequential quadratic programming) based on qpOASES [9] as a QP solver.

The results are given in Figure 4. The same results can be obtained by solving the problem by means of a collocation method available in JModelica.org.

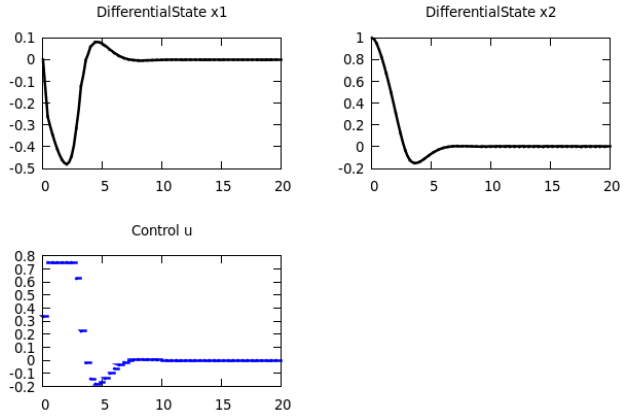

Figure 4. Van der Pol optimization problem results from ACADO

4. Conclusions and future perspectives

In this paper, an XML representation of continuous time DAEs obtained from continuous-time Modelica models has been proposed. The test implementation on the JModelica.org platform has shown the possibility to use the XML representation to export Modelica models and then reuse them in another non-Modelica tool. In the same manner, many other possible applications could be considered [6].

A future version of the schema could extend the representation to hybrid DAE systems. In this case the concept of discontinuous expressions, discrete variables, discrete equations and events should be introduced.

An interesting perspective could be to explore to which extent the proposed DAE representation could be used to describe flattened models written using other equation-based, object-oriented languages, possibly by introducing additional features that are not needed to handle models obtained from Modelica, in the same spirit of the CapeML initiative [4].

Finally, it would also be interesting to investigate the possibility to aggregate models represented by different XML documents. In this case every XML document would

represent a sub-model and an interface to allow more sub-models to be connected should be designed.

References

[1] Johan Åkesson, Karl-Erik Årzén, Magnus Gäfvert, Tove Bergdahl, and Hubertus Tummescheit. Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problem. *Computers and Chemical Engineering*, January 2010. Doi:10.1016/j.compchemeng.2009.11.011.

[2] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Implementation of a Modelica compiler using JastAdd attribute grammars. *Science of Computer Programming*, 75(1):21–38, 2010.

[3] L.T. Biegler, A.M. Cervantes, and A. Wächter. Advances in simultaneous strategies for dynamic process optimization. *Chemical Engineering Science*, 57(4):575–593, 2002.

[4] Christian H. Bischof, H. Martin Bücker, Wolfgang Marquardt, Monika Petera, and Jutta Wyes. Transforming equation-based models in process engineering. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, Lecture Notes in Computational Science and Engineering, pages 189–198. Springer, 2005.

[5] F. Casella, F. Donida, and Åkesson. An XML representation of DAE systems obtained from Modelica models. In *7th Modelica conference*, September, 20-22 2009.

[6] F. Casella, F. Donida, and M. Lovera. Beyond simulation: Computer aided control system design using equation-based object oriented modelling for the next decade. In *2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, July, 8 2008.

[7] F. Casella, F. Donida, and M. Lovera. Automatic generation of LFTs from object-oriented non-linear models with uncertain parameters. In *6th Vienna International Conference on Mathematical Modeling*, February, 11-13 2009.

[8] Francesco Casella, Filippo Donida, and Gianni Ferretti. Model order reduction for object-oriented models: a control systems perspective. In *Proceedings MATHMOD 09 Vienna*, pages 70–80, Vienna, Austria, Feb. 11–13 2009.

[9] H.J. Ferreau, H.G. Bock, and M. Diehl. An online active set strategy to overcome the limitations of explicit MPC. *International Journal of Robust and Nonlinear Control*, 18(8):816–830, 2008.

[10] KU Leuven. ACADO toolkit Home Page. `http://www.acadotoolkit.org/`.

[11] J. Larsson. A framework for simulation-independent simulation models. *Simulation*, 82(9):563–379, 2006.

[12] Modelisar. Functional Mock-up Interface for Model Exchange, 2010. `http://www.functional-mockup-interface.org`.

[13] Modelon AB. JModelica.org Home Page, 2010. `http://www.jmodelica.org`.

[14] A. Pop and P. Fritzson. ModelicaXML: A Modelica XML representation with applications. In *3rd Modelica conference*, November, 3-4 2003.

[15] U. Reisenbichler, H. Kapeller, A. Haumer, C. Kral, F. Pirker, and G. Pascoli. If we only had used XML... In *5th Modelica conference*, September, 4-5 2006.

[16] The Modelica Association. Modelica - a unified object-oriented language for physical systems modeling, 2009. `http://www.modelica.org/documents/ModelicaSpec32.pdf`.

[17] M. Tiller. Implementation of a generic data retrieval API for Modelica. In *4th Modelica conference*, March, 7-8 2005.

Towards a Computer Algebra System with Automatic Differentiation for use with Object-Oriented modelling languages

Joel Andersson Boris Houska Moritz Diehl

Department of Electrical Engineering and Optimization in Engineering Center (OPTEC), K.U. Leuven, Belgium,
{joel.andersson,boris.houska,moritz.diehl}@esat.kuleuven.be

Abstract

The Directed Acyclic Graph (DAG), which can be generated by object oriented modelling languages, is often the most natural way of representing and manipulating a dynamic optimization problem. With this representation, it is possible to step-by-step reformulate an (infinite dimensional) dynamic optimization problem into a (finite dimensional) non-linear program (NLP) by parametrizing the state and control trajectories.

We introduce CasADi, a minimalistic computer algebra system written in completely self-contained C++. The aim of the tool is to offer the benifits of a computer algebra to developers of C++ code, without the overhead usually associated with such systems. In particular, we use the tool to implement automatic differentiation, AD.

For maximum efficiency, CasADi works with two different graph representations interchangeably: One supporting only scalar-valued, built-in unary and binary operations and no branching, similar to the representation used by today's tools for automatic differentiation by operator overloading. Secondly, a representation supporting matrix-valued operations, branchings such as if-statements as well as function calls to arbitrary functions (e.g. ODE/DAE integrators).

We show that the tool performs favorably compared to CppAD and ADOL-C, two state-of-the-art tools for AD by operator overloading. We also show how the tool can be used to solve a simple optimal control problem, minimal fuel rocket flight, by implementing a simple ODE integrator with sensitivity capabilities and solving the problem with the NLP solver IPOPT. In the last example, we show how we can use the tool to couple the modelling tool JModelica with the optimal control software ACADO Toolkit.

Keywords computer algebra system, automatic differentiation, algorithmic differentiation, dynamic optimization, Modelica

3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. October, 2010, Oslo, Norway.

1. Introduction

Simulation of dynamic systems formulated in languages for physical modelling in continuous time typically amounts to solving initial-value problems in ordinary differential equations. This, in turn, requires the repeated evaluation of the ODE right-hand-sides, root-finding functions corresponding to hybrid behavior and user output functions.

With the rising interest of employing the developed dynamic models not only for simulation purposes, but also for dynamic optimization (i.e. optimization problems where the dynamic system enters as a constraint), it becomes critically important to have efficient ways of accurately evaluating not only the these functions, but also their derivatives. Gradient information is needed by implicit methods for integrating ODEs as well as in both simultaneous (e.g. direct collocation) and sequential methods (single shooting, multiple shooting) for dynamic optimization, the two families of methods that have been the most successful in solving large-scale dynamic optimization problems [4].

While dynamic optimization can certainly be useful for users of modelling languages, there are also large synergies for developers of optimization routines. The Directed Acyclic Graph (DAG) representation is used by computer algebra systems as well as object oriented modelling languages and is often the most convenient way to represent a complex non-linear function. The representation provides more information for the optimization routine than a set of "black-box" functions for evaluating nonlinear functions (objective function, constraints, etc.) and their derivatives in addition to the sparsity structure of Hessians and Jacobians, which is the representation used in most of today's tools for nonlinear optimization.

Optimization routines can benefit from the DAG representation since it gives a possibility to manipulate the graph and replace certain nodes in order to give the (dynamic or not dynamic) optimization problem a more favorable form. Examples of this is replacing *internal switches* by *external switches* to form a mixed-integer optimal control problem (MIOCP) when dynamic constraints are present or as a mixed-integer nonlinear problem (MINLP) when this is not the case. Another example, used by software such as CVX [8], is the possibility of reformulating a convex optimization problem, with an infavorable structure, into an equivalent convex problem in form required by numerical

solvers [7]. Finally, we can also easily parametrize the control and/or state trajectory, reducing our dynamic optimization problem into either a parameter estimation problem or even a nonlinear program (NLP) [14].

A second way in which optimization routines can use the DAG representation is through *structure exploitation*. For example, the *Lifted Newton method* [12] offers a way to treat the intermediate variables of a nonlinear optimization problem as degrees of freedom of the problem *without* increasing the size of the problem. Including intermediate variables in the problem formulation is known to increase both the convergence rate and the area of attraction for nonlinear problems. The Lifted Newton method works with a problem formulation which is easily obtained from a linear ordering of the nodes of the DAG.

1.1 Automatic differentiation

Automatic differentiation (or, alternatively, *algorithmic* differentiation), AD, is a technique for evaluating derivatives of complex functions that has proved very useful in non-linear optimization.

The technique delivers derivatives, up to machine precision, of arbitrary differentiable functions for a small multiple of the cost of evaluating the original function. In the *forward mode*, the technique is able to deliver directional derivatives of an arbitrary vector-valued function with a cost of the same order as the cost of evaluating the original function. In the *reverse mode*, on the other hand, it is for the cost of one evaluation possible to get the derivative in an all directions for a scalar function[1].

Efficiently implementing AD, especially in the reverse mode, is a complex task and it is advisable to use one of the existing software implementations dedicated for this. These software tools are typically divided into AD by *operator overloading* and AD by *source code transformation*, see [9] for details.

Since modelling languages such as Modelica typically involve a step of C-code generation, the natural approach of using AD has been to apply one of the existing AD tools to this code.

Automatic differentiation can also be implemented inside a computer algebra system (CAS), first demonstrated by Monagan and Neuenschwander [13].

2. Proposed AD framework for object oriented modelling tools

The method proposed in this paper follows the CAS/AD approach, but instead of using an existing computer algebra system, it implements a minimalistic computer algebra system in C++ using operator overloading. The main difference from the conventional operator overloading approach for AD is that we do not attempt to maintain a linear ordering of the nodes when the graph is constructed. Instead, we generate an ordering after the graph has been constructed, which can be done efficiently using linear time algorithms, described in Section 2.2.

Also different from conventional AD tools is that we shall build a graph of sparse, matrix-valued operations, instead of just scalar operations, and allow the graph to contain ”switches” in the form of if-statements, for-loops etc. This means that there is no need to reconstruct the graph or its linear ordering when a switch fires. When there are a lot of switching events relative to the number of function evaluations, this method promises to be significantly faster.

When calculating the full Hessian or Jacobian (as opposed to only a directional derivative), CasADi uses the *Sparse Jacobian/Hessian* methods rather than compression techniques ([9]). The former methods are more efficient from a theoretical point of view, but not widely used since having to keep track of dependencies for each evaluation node results in significant overhead. CasADi is able to avoid this overhead by resorting to source code transformation, whenever multiple derivative directions (forward or adjoint) are involved.

Given the linear ordering of a graph, it is generally straightforward to generate source code for evaluation of functions and derivatives, but in general we will stop short of doing so and instead evaluate the expressions on a *virtual machine*. This eliminates the need of a C compiler in the loop, which has large practical consequences, and also saves us the trouble of working with potentially very large files. As we shall see, a DAG with millions of nodes can be constructed and sorted within seconds and evaluated in milliseconds, but generating a C source file of it may require hundreds of MB, which have to be written to disk. With the current approach, the graph representation never leave RAM.

2.1 Forming the graph

Consider the following recursion describing the horizontal motion of a ball under the action of friction:

$$s_{k+1} = s_k + v_k, \quad k = 0, \ldots, N-1 \tag{1}$$

$$v_{k+1} = v_k - v_k * v_k \tag{2}$$

For given s_0, v_0 and N, this recursion defines a function $f : \mathbf{R}^2 \rightarrow \mathbf{R}^2$:

$$[s_N, v_N] = f(s_0, v_0; N) \tag{3}$$

Figure 1 shows the DAG, also referred to as the *computational graph* in automatic differentiation terminology [5], of this expression when $N = 2$.

A DAG like this can easily be constructed in an object-oriented programming language like C++. The basic building block can be an object that contains the elementary op-

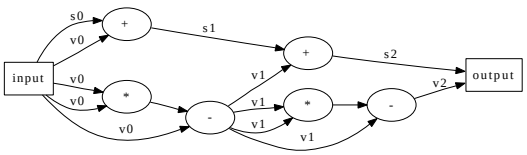

Figure 1. The DAG for the ball example ($N = 2$)

[1] when multiple directions (for the forward mode) or multiple outputs (for the reverse mode) are involved, the process must be repeated for each direction (output), unless the Jacobian has some special structure

eration to be performed as well as references to its dependent nodes. Since a node can have an arbitrary number of nodes pointing to it, we advocate the use of smart pointers (or, more precisely *shared pointers*) in order to be able to keep track of the ownerships of the nodes [2].

More complex functions, which similarly to the ball example arise after parametrization of an optimal control problem may have millions of nodes and it is important to find efficient ways to form and manipulate such graphs.

2.2 Topological sorting

Before a function can be evaluated, we have to order the nodes of the DAG in the order of dependencies, known as finding a *topological sorting* in graph theory. This step is not necessary in AD by operator overloading, since a good linear ordering can be recorded during the process of constructing the graph. In AD by source code transformation, the compiler is responsible for finding this ordering.

A topological ordering can however also be found cheaply after the DAG has been formed, which greatly facilitates the implementation. We propose to use a modified *breadth-first search*, described in the following, to obtain a good topological ordering. Our implementation of this sorting depends on another sorting, the *depth-first search*, which we will desribe first.

2.2.1 Depth-first search

One way of finding a linear ordering is by so-called *depth-first search* [6], which can be implemented in the following way, where top$(\cdot)$ refers to the topmost element of a *stack*[2]:

- Add the output nodes to a stack S
- Create an empty list A for the linear ordering
- While S is nonempty
  - If top(S) has not yet been visited
    - Mark the top(S) as visited
    - Add non-visited nodes to S on which top(S) depends
  - else
    - Add top(S) to A
    - Remove top(S) from S

The algorithm visits each node at most one time and has thus linear complexity in the number of nodes. Figure 2 shows an ordering obtained by this algorithm for the ball example. The example illustrates two problems with this sorting. Firstly, one needs to store v_1 in memory until we calculate s_2. Similarly, for a large N, we would need to keep $v_1, \ldots, v_{N-1}$ in memory at the time we evaluate v_N, since they will be needed later.

Secondly, several operations could be done in parallel, e.g. operation $\{1\}$ and $\{5\}$, in the graph.

2.2.2 Breadth-first search

These problems lead us to look at another ordering, namely the *breadth-first search* [6]. The standard algorithm for this ordering cannot readily be applied to the graph, since we are not able to iterate over *the nodes that depend on a given node*, but only over *the nodes that a given node depends on*. This can, however, be solved with the following algorithm:

- Find a topological sorting A by a depth-first search
- Create a vector of dependency levels L. An operation associated with one level depends only on the results from previous levels and can thus be evaluated independently. Constants and inputs have level 0.
- For $i = \text{begin}(A), \ldots, \text{end}(A)$
  - Find $l_{\text{max},i}$, the maximum level of any of i's dependent nodes
  - Assign level$(i) := 1 + l_{\text{max},i}$
- Sort the nodes by their level using a *bucket sort*

The result of the sorting, for the ball example, is shown in Figure 3. Since all nodes of one level can be evaluated independently of each other, the sorting is suitable for parallelization. Like the depth-first search, this algorithm will run in linear time.

A problem with the breadth-first search is that it tends to evaluate nodes earlier than they are actually needed. For example, when the expression is a simple sum of squares, $f(x_1, \ldots, x_N) = x_1^2 + \ldots + x_N^2$, all multiplications will take place on level 1, creating unnecessary memory overhead. We can solve this by iterating over the levels in reverse order and "move up" dependent nodes as much as possible. This operation has also linear complexity.

3. Software implementation

The proposed algorithms have been been implemented in the open-source C++ tool CasADi, which will be released under the LGPL licence. A stated goal of the tool has been to keep the data structures as transparent as possible to allow a user to easily extend the code with methods from the field of computer algebra, as well as numerical optimization.

The focus of the code is to generate highly efficient runtime code and for this purpose we propose to use a combination of two different DAG representations, one DAG which is restricted to built-in binary (and unary) scalar functions and a general one representing a matrix syntax tree of sparse matrix operations as well as nodes corresponding to switches, loops, function evaluations, element access and concatenation, similar to the graph representation used in a modelling language such as Modelica.

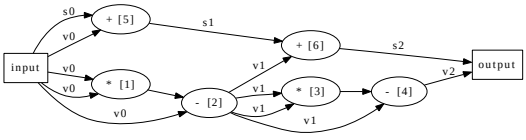

Figure 2. Linear ordering (in brackets) from a depth-first search for the ball example

[2] with stack is meant a *last in, first out* data structure

3.1 A scalar DAG representation

The purpose of the scalar DAG representation is to represent and evaluate an algorithm containing a series of elementary operations (`+`,`-`,`*`,`/`, `*`) and built-in functions from the C's `math.h` library (`floor`, `pow`, `sqrt`, etc.). No branching (if-statements) is allowed. With these restrictions, it is possible for the AD algorithm to access memory in a strictly sequential manner, also for the reverse AD algorithm, which is the reason that existing AD tools usually *only* allows operations of this form [9].

With these restrictions, the class hierarchy simply becomes, with the most important members in brackets:

- Expression class, `SX` [pointer to an `SXNode`]
- Abstract base class for all nodes, `SXNode` [reference counter]
  - Symbolic scalar [name of the variable]
  - Constant [value of the constant]
  - Binary Node [two `SX` instances, index of a binary function]

The graph is represented by a smart pointer class, called `SX`, and a polymorphic node class consisting of an abstract base class and three derived classes, a node corresponding to a constant node, a node corresponding to a variable and a node corresponding to a binary operation. We have gathered all binary operations in a single node rather than deriving a class for each binary operation. The binary nodes, in turn, has members of the class `SX`, correspond to its dependent nodes. This simple representation is flexible enough to be able to construct trees of millions of variables in C++. Using the topological sorting outlined in the previous section, we obtain a linear ordering of the nodes equivalent to the *trace* of automatic differentiation tools [9].

3.2 A matrix DAG representation

The scalar DAG representation is designed to be very efficient for code made up entirely by standard unary and binary operations. In a well-designed code, the lion's share of the calculation time should indeed in sections of this type, so it makes sense to specialize the solver to be efficient in this case.

To be able to represent more general functions, we propose to use a second, much more general, DAG representation. In this representation, the graph is made up by, possibly sparse, *matrix operations* and we include nodes for elementwise operations as well as operations such as matrix products. With this we can also represent a much wider range of operations such as vertical and horizontal concatenation, element access, function evaluation, loops

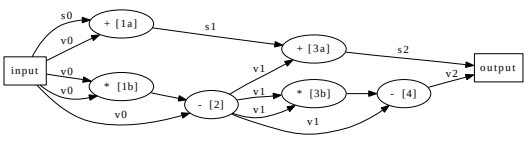

Figure 3. Linear ordering (in brackets) from a breadth-first search for the ball example

(for, while) and switches (if-statements, etc.). This is much more general than is allowed by AD tools, which generally freezes all the if-statements during the *tracing* step and inlines all loops to create a representation similar to the scalar DAG in the previous section. With the proposed approach, we instead leave it to the user to decide which loops and function calls actually should be inlined. By partially inlining the code, we arrive with an approach which is equivalent to the *checkpointing schemes* used by standard AD tools [9].

We node that both forward and adjoint AD readily generalizes to work with graphs of this form. Many operations, like element access, just translate to linear matrix operations (which are often sparse). Other operations, such as matrix products and function evaluations, have explicit chain rules defined for them.

When evaluating a function represented by the Matrix DAG on a multi-core (shared memory) architecture, independent nodes, as obtained from the breadth-first-search, can be evaluated in parallel (multiple threads) using OpenMP.

The implementation of the matrix expression class, `MX`, is similar to the `SX` class above with the following differences.

- The Binary Node class becomes polymorphic, with one derived node for each type of operation. This makes it possible for the binary nodes to contain more information, such as pointers to functions or more than two arguments.
- The *MX* class contains arrays for the evaluation, and the evaluation takes place by looping over a vector of pointers to the nodes, rather than in a separate data structure.

4. Numerical tests

4.1 An AD example: determinant calculation

We first test the algorithm on an AD benchmark, namely the calculation of the adjoint derivivative of the determinants of matrices of different sizes. The determinant is calculated by expansion along rows, an exponentially expensive calculation.

This example is implemented in the speed benchmark collection of CppAD. We use this implementation to test the performance of CasADi against CppAD as well as ADOL-C, [3]. Figure 4 shows the speed, in number of solves per second, for the three tools for matrices of sizes ranging from 1-by-1 to 9-by-9. The tests have been performed on an Dell Latitude E6400 laptop with an Intel Core Duo processor of 2.4 GHz, 4 GB of RAM, 3072 KB of L2 Cache and 128 kB if L1 cache. The operator system is Linux.

We use the latest version of ADOL-C at the time of writing, version 2-1-5, and the latest version of CppAD, released 17 June 2010.

For the current test, CasADi outperforms the CppAD and ADOL-C for sizes up to 8-by-8 (corresponding to some 100.000 elementary operations), but we want to stress that

the test only covers one single example and we make no claims that these results will hold generally. More tests are needed to better assess the performance of the solver.

4.2 Dynamic optimization example: minimal fuel rocket flight

We then study an example from optimal control, namely the minimal fuel flight of a rocket described by the following continuous time model:

$$\dot{s} = v \tag{4}$$
$$\dot{v} = (u - \alpha\, v^2)/m \tag{5}$$
$$\dot{m} = -\beta\, u^2 \tag{6}$$
$$\tag{7}$$

We assume that the rocket starts at rest at ($s(0) = 0$, $v(0) = 0$) with mass $m(0) = 1$. We simulate for $T = 10$ seconds and require that the rocket lands at rest ($v(T) = 0$) at $s(T) = 10$. The optimization problem is to minimize the fuel consumption $m(0) - m(T)$.

4.2.1 Solution approach

We discretize the control into $N_u = 1000$ piecewise constant controls and solve the problem using a *single-shooting approach*. The single-shooting approach requires a sensitivity-capable integrator which we can easily construct in CasADi with a few lines of code using an *explicit Euler approach* with 1000 steps per control interval of equal length (we wish to stress that there are certainly much better ways of solving this optimal control problem, but our purpose here is only to illustrate our AD approach).

We show two different ways of solving the problem using CasADi, with scalar graphs and a combination of scalar and matrix graphs (next section). In both approaches we arrive at a non-linear programming problem (NLP) [14] of the form:

$$\begin{array}{cc} \text{minimize:} & \sum_{i=0}^{999} u_i^2 \\ u_0, \ldots, u_{999} & \\ \text{subject to:} & g(u) = 0, \\ & -10 \leq u \leq 10 \end{array} \tag{8}$$

where $g(u) : \mathbf{R}^{1000} \rightarrow \mathbf{R}^2$ is a non-linear function that we shall construct. This NLP is then solved by the non-linear optimization code Ipopt [16], which requires not only function evaluation, but also the gradient of the objective function and the Jacobian of the constraints. We use CasADi to obtain this information. Since there are 1000 variables but only two constraints, it makes sense to use the adjoint mode AD to calculate the Jacobian of g.

4.2.2 Using scalar graphs

In the first approach, we use two nested for loops to calculate one large graph with around 13 million nodes (13 being the number of elementary operations in each step). This is the approach taken by default by existing AD tools. In the CasADi notation, we get:

```
SX s = 0, v = 0, m = 1;
for(int k=0; k<1000; ++k){
  for(int j=0; j<1000; ++j){
    s += dt*v;
    v += dt / m * (u[k] - alpha * v*v);
    m += -dt * beta*u[k]*u[k];
  }
}
```

where SX is the name of the symbolic expression type used in CasADi, which can be used in the same way as C/C++ `double`. We assume that the input u is stored in a vector of symbolic variables of length 1000. After this recursion, the expression for g is then simply obtained as $g(u) = [s - 10; v]$.

Ipopt requires 11 iterations to solve this problem and the total solution time was 19.6 seconds out of which 7.8 seconds was needed for the function evaluations (the lion's share of the rest being needed to form and sort the graphs). A single function evaluation, corresponding to around 13 million elementary operations, takes about 0.27 seconds, or around 20 ns per elementary operation.

4.2.3 Using scalar and matrix graphs

The approach above is basically to *inline everything* and it is clear that the graphs will soon become too large. We therefore show a second way to represent the same function based on a two level approach. We first use the above approach over an interval with a constant control only:

```
SX s_0("s_0"), v_0("v_0"), m_0("m_0");
SX u("u");
SX s = s_0, v = v_0, m = m_0;
for(int j=0; j<1000; ++j){
  s += dt*v;
  v += dt / m * (u[k] - alpha * v*v);
  m += -dt * beta*u*u;
}
```

which we use to generate an function (an *integrator*) with some 13 thousand nodes (instead of 13 million). This function will take as input (s_0,v_0,m_0 and u) and return the three outputs (s,v,m). We then evaluate this function 1000 times using out matrix graph representation:

```
MX X = {0,0,1}; // initial value
for(int k=0; k<1000; ++k){
  // Integrate
  vector<MX> input = {U[k],X};
  X = integrator(input);
}
```

where MX is the name of CasADi's matrix expression class. The second loop will be represented by a graph with about 2000 nodes, 1000 "element access" nodes (`U[k]`) and 1000 "function evaluation" nodes. For Ipopt, both approaches are equivalent, they are simply two ways of calculating the same function g, so the optimization results are indeed identical. The significantly lower memory need in the second approach, however, enables us to construct much larger problems (e.g. taking 100 times more steps) without running out of memory.

The solution of the problem using the scalar and matrix graph combination took 10.4 seconds, out of which 9.7

seconds was needed for the function and derivative evaluation. The example clearly shows how this approach makes forming the graphs significantly faster (less than one second instead of 11), but in terms of solution times for a single function evaluation, the scalar graph approach is still significantly quicker, as it requires less operations.

4.3 A Modelica example: Van der Pol oscillator

In the third example, we will use CasADi as an interface between two optimization tools, the Optimal control package ACADO Toolkit [11] and the Modelica-compiler JModelica [1]. We wish to solve the following optimal control problem describing a Van der Pol oscillator:

$$\begin{array}{ll} \text{minimize:} & \\ x_1(\cdot), x_2(\cdot), u(\cdot) & \displaystyle\int_0^{20} e^{p_3}\left(x_1^2 + x_2^2 + u^2\right) dt \\ & \\ & \dot{x_1} = (1 - x_2^2)\, x_1 - x_2 + u \\ \text{subject to:} & \dot{x_2} = p_1\, x_1 \\ & x_1(0) = 0, \quad x_2(0) = 1 \\ & u \leq 0.75 \end{array} \tag{9}$$

where the p_0, p_1, p_2 are parameters and $x_1(\cdot)$ and $x_2(\cdot)$ are state variables (time dependent) and $u(\cdot)$ is a control.

The example is taken from the JModelica benchmark collection and has been implemented there in the Optimica extension of Modelica [2]. We use the newly added XML export functionality of JModelica to export the optimal control problem in a fully symbolic form. This XML code is then parsed in CasADi using the XML parser TinyXML [15].

We use the optimal control package ACADO Toolkit to solve the optimal control problem coupled to CasADi for evaluating the objective function, the ODE right hand side of their and derivatives. ACADO Toolkit uses a multiple-shooting methods to discretize the optimal control problem to a non-linear program (NLP) and then solves the NLP using an Sequential Quadratic Programming (SQP) method [14]. Using a limited memory Hessian approximation and initialized with an zero control, 26 iterations were needed to solve the problem.

Figure 5 shows the state and control trajectories for the optimal solution as obtained by the tool coupling. The results agree with those obtained by JModelica's built-in optimal control problem solver, which is based on direct collocation [2].

5. Conclusions and outlook

The directed graph is a natural way of representing a non-linear function and this formulation can be used not only to formulate an optimal control problem, but also to reformulate the problem into the canonical form used by current state-of-the-art solvers for large-scale optimization problem. Automatic differentiation, both in forward and in reverse mode, can be implemented efficiently directly on the graph instead of taking the detour over generating C-code and then using an existing AD tool for iteratively getting generating linear orderings corresponding to different branches.

We have presented CasADi, an AD tool using a function representation borrowed from the field of computer algebra, also found in object oriented modelling languages, and certainly much richer than that of most existing AD tools. The tool has been coupled to optimal control software ACADO Toolkit, the nonlinear programming solver Ipopt [16] as well as the CVodes of the Sundials suite [10], but we want to stress that CasADi is *not* intended to be just an interface between optimization tools and modelling environments. The idea is instead to actually implement the optimization algorithms using the graph representation and exploit the structure as much as possible.

The main scope of the tool thus starts off where current tools, e.g. JModelica, generate C-code to be used in a numerical solver. Using the graph representation, it is possible to step-by-step reformulate the infinite dimensional, possibly non-smooth OCP into a nonlinear problem (NLP), even going as far as to even solve the NLP. The latter is of particular interest if we wish to generate code for a, say, nonlinear model predictive control to be used on embedded systems.

Since the graphs after parameterizing states as well as controls will be significantly larger than the graphs needed to represent the optimal control problem, it makes sense to have the graph constructed in a language such as C++, rather than, say, Java or a scripting language such as Python. Given the excellent possibilities to interface C++ to other languages, this should not be an issue.

Acknowledgments

This research was supported by the Research Council KUL via the Center of Excellence on Optimization in Engineering EF/05/006 (OPTEC, http://www.kuleuven.be/optec/), GOA AMBioRICS, IOF-SCORES4CHEM and PhD/postdoc/fellow grants, the Flemish Government via FWO (PhD/postdoc grants, projects G.0452.04, G.0499.04, G.0211.05,G.0226.06, G.0321.06, G.0302.07, G.0320.08, G.0558.08, G.0557.08, research communities ICCoS, ANMMM, MLDM) and via IWT (PhD Grants, McKnow-E, Eureka-Flite+), Helmholtz Gemeinschaft via vICeRP, the EU via ERNSI, Contract Research AMINAL, as well as the Belgian Federal Science Policy Office: IUAP P6/04 (DYSCO, Dynamical systems, control and optimization, 2007-2011).

References

[1] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Implementation of a modelica compiler using jastadd attribute grammars. *Science of Computer Programming*, 75(1-2):21 – 38, 2010. Special Issue on ETAPS 2006 and 2007 Workshops on Language Descriptions, Tools, and Applications (LDTA '06 and '07).

[2] Andrei Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2008.

[3] Andreas Griewank and David Juedes and Jean Utke. Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw.*, 22(2):131–167, 1996.

[4] Lorenz T. Biegler. An overview of simultaneous strategies for dynamic optimization. *Chemical Engineering and Processing: Process Intensification*, 46(11):1043–1053, 2007. Special Issue on Process Optimization and Control in Chemical Engineering and Processing.

[5] Christian H. Bischof and Paul D. Hovland and Boyana Norris. Implementation of automatic differentiation tools (invited talk). *j-SIGPLAN*, 37(3):98–107, March 2002. Proceedings of the 2002 ACM SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02).

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.

[7] M. Grant and S. Boyd. Graph implementations for nonsmooth convex programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent Advances in Learning and Control*, Lecture Notes in Control and Information Sciences, pages 95–110. Springer-Verlag Limited, 2008.

[8] M. Grant and S. Boyd. CVX: Matlab software for disciplined convex programming, version 1.21. `http://cvxr.com/cvx`, May 2010.

[9] Andreas Griewank and Andrea Walther. *Evaluating Derivatives*. SIAM, 2008.

[10] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software*, 2005.

[11] Boris Houska, Hans Joachim Ferreau, and Moritz Diehl. Acado toolkit - an open-source framework for automatic control and dynamic optimization. *Optimal Control Methods and Application*, 2010.

[12] Jan Albersmeyer and Moritz Diehl. The Lifted Newton Method and Its Application in Optimization. *SIAM J. Optim*, 20(3):1655–1684, 2010.

[13] Monagan, Michael B. and Neuenschwander, Walter M. GRADIENT: algorithmic differentiation in Maple. In *ISSAC '93: Proceedings of the 1993 international symposium on Symbolic and algebraic computation*, pages 68–76, New York, NY, USA, 1993. ACM.

[14] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, August 2000.

[15] Lee Thomason, Yves Berquin, and Andrew Ellerton. Tinyxml, version 2.6.0. `http://www.grinninglizard.com/tinyxml/`, May 2010.

[16] A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.

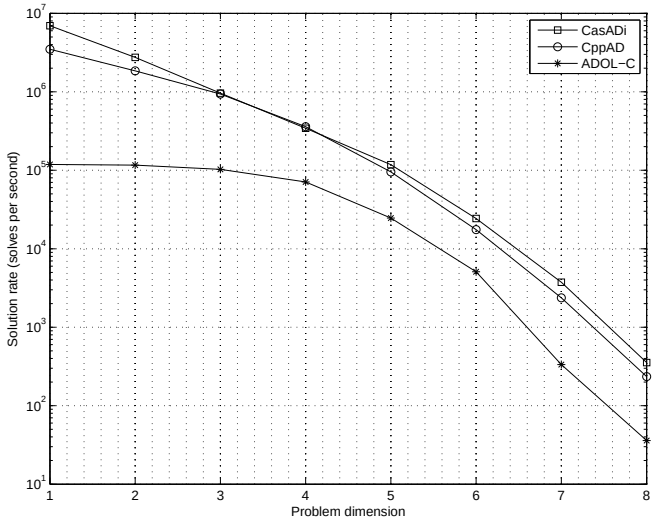

Figure 4. AD benchmark test: determinant by minor expansion

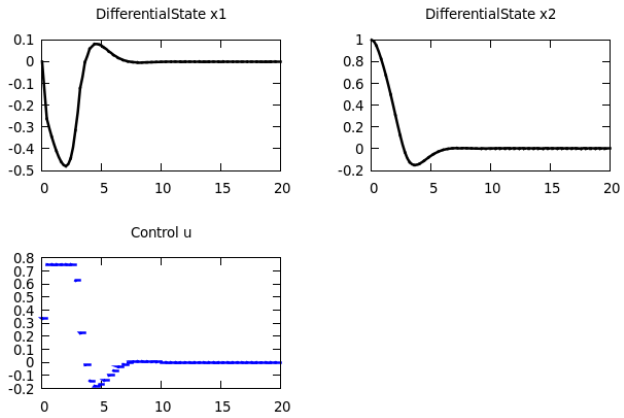

Figure 5. The optimal state and control trajectories for the Van der Pol oscillator example

Discretizing Time or States? A Comparative Study between DASSL and QSS - Work in Progress Paper -

Xenofon Floros[1] François E. Cellier [1] Ernesto Kofman[2]

[1]Department of Computer Science, ETH Zurich, Switzerland
{xenofon.floros,francois.cellier}@inf.ethz.ch
[2]Laboratorio de Sistemas Dinámicos, FCEIA, Universidad Nacional de Rosario, Argentina
kofman@fceia.unr.edu.ar

Abstract

In this study, a system is presented and analyzed that automatically translates a model described within the **Modelica** framework into the Discrete Event System Specification (**DEVS**) formalism.

More specifically, this work interfaces the open-source implementation of Modelica, **OpenModelica**, and one particular software tool for DEVS modeling and simulation, the **PowerDEVS** environment, which implements the Quantized State Systems (**QSS**) integration methods introduced by Kofman.

The interface enables the automatic simulation of large-scale models with both DASSL (using the OpenModelica run-time environment) and QSS (using PowerDEVS) and extracts features, such as accuracy and simulation time, that allow a quantitative comparison of these integration methods. In this way, meaningful insight can be obtained on their respective advantages and disadvantages when used for simulating real-world applications. Furthermore, the implemented interface allows any user without any knowledge of DEVS and/or QSS methods to simulate their systems in PowerDEVS by supplying a Modelica model as input only.

Keywords OpenModelica, DASSL, PowerDEVS, QSS, sparse system simulation

1. Introduction

Modelica [4, 5] is a an object-oriented, equation-based language that enables a standardized way to model complex physical systems containing, e.g., mechanical, electrical, electronic, hydraulic, thermal, control, electric power, or process-oriented subcomponents. The Modelica language allows the representation of continuous and hybrid models with a set of non-causal equations.

Modelica modeling environments, such as Dymola, Scicos, and the open-source OpenModelica software [3], after performing a series of preprocessing steps (model flattening, sorting and optimizing the equations, index reduction), convert the model to a set of explicit ODEs of the form:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t) \tag{1}$$

The built-in simulation environments provide routines (solvers) that invoke the right-hand side evaluation of Eq. 1 at discrete time steps t_k, in order to compute the next value of the state vector $\mathbf{x_{k+1}}$. At least in the case of Dymola and OpenModelica, efficient C++ code is generated in order to perform the simulation. Both software environments make use of **time slicing**, i.e., their underlying simulation algorithms are based on time discretization rather than state quantization.

Recently, a new class of algorithms for the numerical integration of ODEs based on **state quantization** and the DEVS formalism introduced by Zeigler [13] was proposed. A first-order non-stiff Quantized State System (QSS1) algorithm was introduced by Kofman in 2001 [6], followed by second and third-order accurate non-stiff solvers, called QSS2 [10] and QSS3 [9], respectively. The family of QSS methods presented are implemented in PowerDEVS, [11], a DEVS-based simulation software. In the mean time, also stiff QSS solvers as well as QSS solvers for dealing with marginally stable systems were introduced.

QSS methods have been theoretically analyzed to exhibit nice stability, convergence, and error bound properties, [2, 9, 10], and in general come with the following benefits over classical approaches:

- Most of the classical methods that use discretization of time, need to have their variables updated in a **synchronous** way. This means that the variables that show fast changes are driving the selection of the time steps. In a stiff system with widely-spread eigenvalues, i.e., with mixed slow and fast subsystems, the slowly changing state variables will have to be updated much more

3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. October, 2010, Oslo, Norway.

frequently than necessary, thus increasing substantially the computation time of the simulation. On the other hand, the QSS methods allow for **asynchronous** variable updates, allowing each state variable to be updated at its own pace, and specifically when an event triggers its evaluation. Furthermore as most systems are sparse, when a state variable x_i changes its value, it suffices to evaluate only those components of $\mathbf{f}$ in Eq. 1 that depend on x_i, allowing for a **significant reduction of the computational costs**.

- Dymola and OpenModelica handle **discontinuities** using **zero-crossing functions** that need to be evaluated at each step, and when they change their sign, the solver knows that a discontinuity occurred. Then an iterative process is initiated in order to detect the exact time of that event. In contrast, QSS methods provide dense output and do not need to iterate to detect discontinuities, but rather predict them. This feature, besides improving on the overall computational performance of these solvers, enables **real-time simulation**. Since in a real-time simulation the computational load per unit of real time must be controllable, Newton iterations are usually not admitted for use in real-time simulation.
- Another important advantage of DEVS methods arises in the context of **hybrid systems**, where continuous time, discrete time, and discrete event models can coexist as subcomponents of an overall system. DEVS methods [8] provide a unified simulation framework for hybrid models, because all of these model types can be represented as valid DEVS models.

Therefore, **state quantization** and the QSS methods appear promising in the context of simulating certain classes of real-world problems. However in order to simulate systems in PowerDEVS directly, the user will have to manually convert his or her model to an explicit ODE form. This is only feasible in the case of very small systems. PowerDEVS unfortunately is not object oriented. For this reason, it is much more convenient for a user to formulate models in the Modelica language than in PowerDEVS.

This works aims to bridge the gap between the powerful object-oriented modeling platform of Modelica on the one hand and the equally powerful simulation platform of PowerDEVS on the other. The interface between OpenModelica and PowerDEVS, introduced in this article, allows a modeler to formulate his or her model in the Modelica language, while simulating it in PowerDEVS. The necessary compilation of the Modelica model to PowerDEVS is fully automatic, and the user does not need to know anything about either DEVS or QSS in order to take advantage of it.

1.1 Relevance of Work

The run-time efficiency of the DASSL and QSS solvers, when used to simulate Modelica models, has so far not been compared in an automated, large-scale framework. In earlier publications describing QSS methods, [6, 7, 8, 10], there can be found examples that demonstrate the superiority of the run-time efficiency of QSS methods, when simulating sparse and discontinuous systems, but the comparison has invariably been restricted to small-scale models that could be easily modeled in PowerDEVS directly.

Furthermore, there have been other approaches, [1, 12], to implement Modelica libraries that allow for DEVS models inside a Modelica environment, but these approaches require from the users to understand the DEVS framework, as they would have to model their system in the DEVS formalism in order to make use of these libraries. In that context, the object orientation of continuous-time models is lost.

In contrast, our approach enables a Modelica user to simulate a Modelica model using QSS solvers without any explicit manual transformation. Furthermore, it allows for the automatic transformations of large-scale models to the DEVS formalism, which is a difficult if not unfeasible task even for experts in DEVS modeling.

The article is organized as follows: Section 2 provides a brief introduction of the QSS methods. Section 3 describes theoretically what is needed in order to simulate a Modelica model without discontinuities employing the QSS algorithms. In Section 4, the actual implementation of the interface between OpenModelica and PowerDEVS is presented. Section 5 describes the simulation results comparing the DASSL solver of the OpenModelica run-time environment with the QSS methods as implemented in PowerDEVS. Finally, Section 6 concludes this study, lists open problems, and offers directions for future work.

2. QSS Simulation

Let a time invariant ODE system:

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

where $\mathbf{x}(t) \in \mathbb{R}^n$ is the state vector. The QSS1 method approximates the ODE in Eq. 2 as:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t)) \tag{3}$$

where $\mathbf{q}(t)$ is a vector containing the quantized state variables. Each quantized state variable $q_i(t)$ follows a piecewise constant trajectory via the following quantization function with hysteresis:

$$q_i(t) = \begin{cases} x_i(t) & \text{if } |q_i(t^-) - x_i(t)| = \Delta Q_i, \\ q_i(t^-) & \text{otherwise.} \end{cases} \tag{4}$$

where ΔQ_i is called quantum. Thus, the quantized state $q_i(t)$ changes if and only if it differs more than the value of the quantum from the state variable $x_i(t)$. In QSS1, the quantized states $\mathbf{q}(t)$ are following piecewise constant trajectories, and since the time derivatives, $\dot{\mathbf{x}}(t)$, are functions of the quantized states, they are also piecewise constant, and consequently, the states, $\mathbf{x}(t)$, themselves are composed of piecewise linear trajectories.

Unfortunately, QSS1 is a first-order accurate method only, and therefore, in order to keep the simulation error small, a large number of short integration steps needs to be calculated.

To circumvent this problem, higher-order methods have been proposed. In QSS2 [10], the quantized state variables

evolve in a piecewise linear way with the state variables following piecewise parabolic trajectories. In the third-order accurate extension, QSS3 [9], the quantized states follow piecewise parabolic trajectories, while the states themselves exhibit piecewise cubic trajectories.

Eq. 3 reveals one of the key concepts and advantages of QSS methods. During simulation, steps are only performed when a quantized variable $q_i(t)$ changes substantially, i.e. when it deviates by more than a quantum from the corresponding state $x_i(t)$. Thus, QSS methods inherently focus on the part of the system that is active at a certain time point, which is particularly attractive for the simulation of sparse models of large real-world systems.

3. Modelica Models Simulation with QSS and DEVS

The Modelica language enables high-level modeling of complex systems. However, at the core of every Modelica model lies an Ordinary Differential Equation (ODE) system or, in general, a Differential Algebraic Equation (DAE) system that mathematically represents the system under consideration. We shall now show how a Modelica model can be simulated using QSS methods. For simplicity, we shall assume that the model is described by an ODE system.

Let us write again Eq. 3 expanded to its individual component equations:

$$\begin{aligned} \dot{x}_1 &= f_1(q_1, \ldots, q_n, t) \\ &\vdots \\ \dot{x}_n &= f_n(q_1, \ldots, q_n, t) \end{aligned} \tag{5}$$

If we consider a single component of Eq. 5, we can split it into two equations:

$$q_i = Q(x_i) = Q(\int \dot{x}_i \, dt) \tag{6}$$

$$\dot{x}_i = f_i(q_1, \ldots, q_n, t) \tag{7}$$

The **DEVS** formalism [13] allows to describe the above equations via a coupling of simpler DEVS models. More specifically:

- The first equation (Eq. 6) can be represented by an atomic DEVS model, called **Quantized Integrator**, with $\dot{x}_i$ as input and q_i as output.
- The second equation (Eq. 7) can also be represented as an atomic DEVS model, called **Static Function**, that receives the sequence of events, $q_1, \ldots, q_n$, and calculates the sequence of state derivative values, $\dot{x}_i$.

Thus in absence of discontinuities, we can simulate Eq. 5 using a coupled DEVS model consisting of the coupling of n Quantized Integrators and n Static Functions. A block diagram representing the final DEVS model is shown in Fig. 1.

The model depicted in Fig. 1 contains all possible connections between the state variables. However, real-world systems are sparse as each state normally depends on a small subset of other states only. Thus, the graphical DEVS structure is typically sparse for most practical applications.

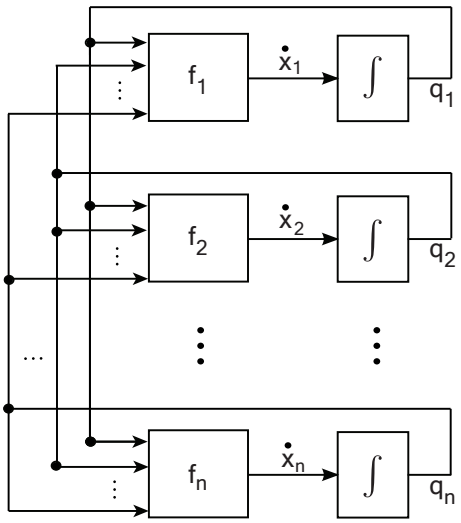

Figure 1. Coupled DEVS model for QSS simulation of Eq. 5

4. OpenModelica to PowerDEVS (OMPD) Interface

This section describes the work done to enable the simulation of Modelica models in PowerDEVS using QSS algorithms.

4.1 What is Needed by PowerDEVS

Let us first concentrate on what PowerDEVS requires in order to perform the simulation of a Modelica model. As depicted in Fig. 1, an essential component of a PowerDEVS simulation is the graphical structure. In PowerDEVS, the structure is provided in the form of a dedicated **.pds structure file** that contains information about the blocks (nodes) of the graph as well as the connections (edges) between those blocks. More specifically, we need to add in the structure:

- A **Quantized Integrator** block for each state variable with $\dot{x}_i$ as input and q_i as output.
- A **Static Function** block for each state variable that receives as input the sequence of events, $q_1, \ldots, q_n$, and calculates $\dot{x}_i$.
- A connection is added between two blocks if and only if there is a dependence between them.

Having correctly identified the DEVS structure, we need to specify what needs to be calculated inside each of the static function blocks. The different blocks need to have access to different pieces of information.

In the current implementation, a **.cpp code file** is generated that contains the code and parameters for all blocks in the structure. The generated code file contains the following information:

- For each **Quantized Integrator** block, the initial condition, error tolerance, and integration method (QSS1, QSS2, QSS3).
- For each **Static Function**, the equations/expressions needed in order to calculate the derivative of each state variable in the system. Furthermore, the desired error tolerance is provided together with a listing of all input and output variables of the specific block.

4.2 What is Provided by OpenModelica

In Section 4.1, we described what must be contained in the files needed to perform simulations in PowerDEVS. The PowerDEVS simulation files should be generated automatically exploiting the information contained in the Modelica model supplied as input. Luckily, existing software used to simulate Modelica models, such as Dymola or OpenModelica, produces simulation code that contains all information needed by PowerDEVS. Thus, we were able to make use of an existing simulation environment by modifying the existing code generation modules to produce the desired simulation files.

This work is based on modifying the OpenModelica Compiler (OMC), since it is open-source and has a constantly growing contributing community. OMC takes as input a Modelica source file and translates it first to a flat model. The flattening consists of parsing, type-checking, performing all object-oriented operations such as inheritance, modifications, etc. The flat model includes a set of equation declarations and functions, with all object-oriented structure removed. Then the equations are analyzed, sorted in Block Lower Triangular (BLT) form, and optimized. Finally, the code generator at the back end of OMC produces c++ code that is then compiled. The resulting executable is used for the simulation of the model.

The information needed to be extracted from the OMC compiler is contained mainly in the DLOW structure where the following pieces of information are defined:

- Equations: $E = \{e_1, e_2, \ldots, e_N\}$
- Variables: $V = \{v_1, v_2, \ldots, v_N\} = V_S \bigcup V_R$ where V_S is the set of state variables with $|V_S| = N_S \leq N$ and V_R the set of all other variables in the model.
- BLT blocks: subsets of equations $\{e_i\}$ needed to be solved together because they are part of an algebraic loop.
- Incidence matrix: An $N \times N$ adjacency matrix denoting, which variables are contained in each equation.

The OMPD interface utilizes the above information and implements the following steps:

1. **Equation splitting** : The interface extracts the indices of the equations needed in order to compute the derivative of each state variable. To achieve this, it builds a dependence graph, where the nodes are the equations, and the edges represent dependences between these equations. The interface traverses the graph backwards from each state derivative until it cannot no longer reach any additional nodes.
2. **Mapping split equations to BLT blocks** : Having extracted the indices of the equations that calculate each state derivative, the equations are mapped back to BLT blocks of equations. This is needed in order to pass this information to the part of the OpenModelica compiler that is responsible for solving linear/non-linear algebraic loops.
3. **Mapping split equations to DEVS blocks** : The split equations are also mapped onto the static blocks of the DEVS structure. Since the state variables and the equations needed to compute them have been identified, they are assigned sequentially to static blocks in the DEVS structure. Each static block corresponds then to a state variable, and the lookup of equations in static blocks can be performed efficiently if needed in the future.
4. **Generating DEVS structure** : In order to correctly generate the DEVS structure of the model, the dependences between the state variables that are computed in each static block have to be resolved. This is accomplished by employing the incidence matrix and the mapping of equations to DEVS blocks from step 3 to find the corresponding inputs for each block. In Fig. 2, an example of a DEVS structure automatically generated for model M_1 is depicted.
5. **Generating the .pds structure file**: Having correctly produced the DEVS structure for PowerDEVS, outputting the respective .pds structure file is straightforward.
6. **Generating static blocks code** : In this step, the functionality of each static block is defined via the simulation code provided in the code.cpp file. Each static block needs to know its inputs and outputs, identified by the DEVS structure, as well as the BLT blocks needed to compute the corresponding state derivatives, described by the mapped split equations. Then, the existing code generation module of OMC is employed to provide the actual simulation code for each static block, since it has already been optimized to solve linear and non-linear algebraic loops.
7. **Generating the .cpp code file**: The code for the static blocks is output in the .cpp code file along with other needed information.

4.3 Example Model M_1

Various tests have been performed in order to ensure that the implemented OMPD interface is performing the desired tasks correctly. Here are presented the results of processing the following second-order non-linear model through the implemented interface:

```
model M1
  Real x1;
  Real x2;
  Real u2;
equation
  der(x1)=x2-x1/10;
```

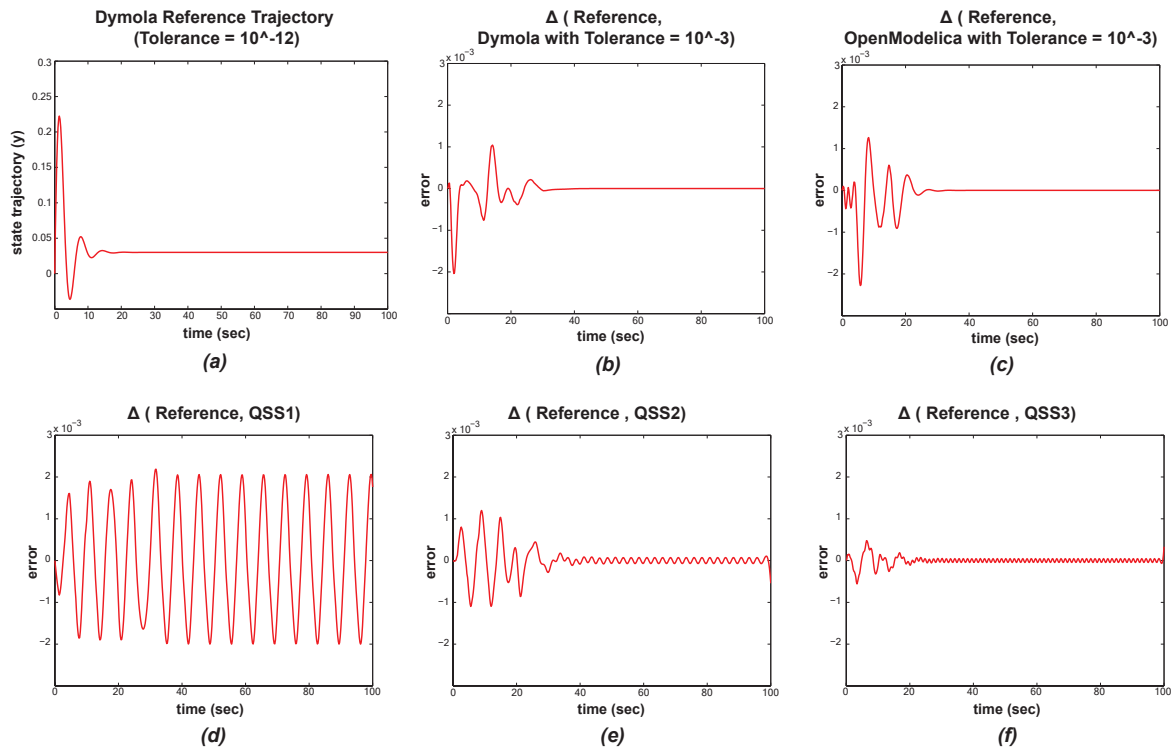

Figure 3. Simulation results for model M_1. In (a), the reference trajectory of state variable x_2 is depicted. It was computed using DASSL by setting the tolerance to 10^{-12}. In (b) and (c), the simulation errors of Dymola and OpenModelica relative to the reference solution are plotted with the tolerance value now set to 10^{-3}. The achieved accuracy is indeed in the order of 10^{-3} with the errors converging to zero in both simulations. In (d), (e), and (f), the simulation errors of QSS1, QSS2, and QSS3 are depicted. Again we observe that the desired accuracy of 10^{-3} is approximately attained. In QSS2 and QSS3, the errors decay also, but a small amplitude high-frequency oscillation around the steady-state remains. In QSS1, the errors don't converge to zero, but remain of the order of 10^{-3}, which is still within specs.

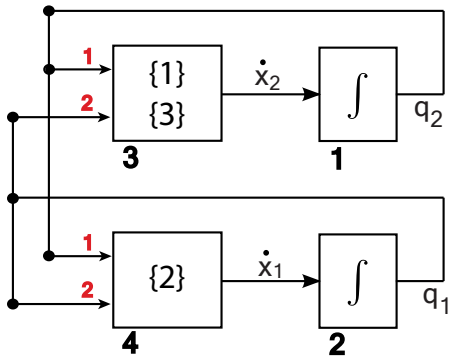

Figure 2. Automatically generated DEVS structure for M_1

```
  der(x2)=u2-x1;
  u2=(1-u2-x2)^3;
end M1;
```

The following information has been extracted from the OMC compiler:

```
Variables (3)
=========
1:  $u2:VARIABLE sys3, Real type: Real ...
2:  $x2:STATE sys3, Real type: Real  ...
3:  $x1:STATE sys3, Real type: Real ...

Equations (3)
=========
1 : $u2 = (1.0 - $u2 - $x2) ^ 3.0
2 : $DER$x1 = $x2 - $x1 / 10.0
3 : $DER$x2 = $u2 - $x1

BLT blocks
2,
1,
3,
```

In other words, OMC has identified that model M_1 has 2 state variables, x_1 and x_2, 1 algebraic variable, u_2, and 3 equations:

$$u_2 = (1 - u_2 - x_2)^3 \qquad \{1\}$$
$$\dot{x}_1 = x_2 - \frac{x_1}{10} \qquad \{2\}$$
$$\dot{x}_2 = u_2 - x_1 \qquad \{3\}$$

Each of the equations is placed in a BLT block. These are sorted according to the execution order as follows: {2}, {1}, {3}.

The automatically generated DEVS structure contained in the .pds file is illustrated in Fig. 2. Since both state variables are depending on each other, the obtained structure is full. The BLT blocks needed to calculate each state derivative are shown inside the static function blocks.

5. Simulation Results

In this section, the simulation results obtained by use of the OPMD interfaced are presented and discussed. First in order to provide a simple working example of the implemented interface, model M_1 was simulated using all three currently implemented QSS methods of PowerDEVS as well as the standard DASSL solvers of OpenModelica and Dymola. The stand-alone version of PowerDEVS contains additional QSS-based solvers for simulating stiff and marginally stable systems, but in the OPMD, only QSS1, QSS2, and QSS3 were included until now.

The article also presents preliminary results regarding the performance of QSS algorithms in comparison with DASSL when simulating sparse systems.

5.1 Simulation of a Simple Model (M_1)

In Fig. 3, the simulation results obtained for model M_1 are depicted. In Fig. 3(a), the reference trajectory of state variable x_2 is plotted. To obtain the reference trajectory, Dymola was employed using the default DASSL solver while setting the tolerance value to 10^{-12}. In the remaining panels, the simulation errors relative to the reference solution are shown that were obtained by setting the desired tolerance to 10^{-3}. Fig. 3(b) shows the simulation error obtained by Dymola using DASSL. Fig. 3(c) depicts the simulation error obtained by OpenModelica using DASSL. Figs. 3(d-f) graph the simulation error obtained by OpenModelica with PowerDEVS using QSS1, QSS2, and QSS3, respectively.

We observe that all solvers accomplish the desired task of keeping the global simulation error approximately within 10^{-3}. Using DASSL, this is not obvious, because DASSL only controls the local simulation error of a single integration step. Errors can in principle accumulate over time, but this didn't happen in this simple example. QSS-based algorithms all control the global simulation error, and consequently, are expected to perform as desired. In this simple example, the simulation errors decrease over time in all solvers except for QSS1. Yet, whereas the errors approach zero in the case of DASSL, a small-amplitude, high-frequency oscillation remains in the cases of QSS2 and QSS3.

These steady-state oscillations are bad news, because they will foce small step sizes upon us in steady state, if we decide for whatever reason to simulate the system over a longer time period. Such steady-state oscillations are observed frequently in the non-stiff QSS solvers. They will disappear when the OPMD is extended to the stiff LIQSS1, LIQSS2, and LIQSS3 solvers.

In QSS1, the errors do not decay over time in this example. After all, this is only a simple first-order non-stiff ODE solver. However, the errors remain approximately within 10^{-3}, i.e., the simulation still performs within the desired specifications.

Based on these results and a number of additional example models studied, we conclude that the OMPD interface is able to simulate arbitrary linear or non-linear Modelica models without discontinuities.

In the following experiments, only the QSS3 algorithm is compared to DASSL, since it is a third-order accurate method and is expected to be more efficient than the other implemented QSS methods.

5.2 Benchmark Framework

One of the goals of this study was to compare the performance of the standard DASSL solver of OpenModelica with the most efficient among the hitherto implemented methods of the QSS family, namely QSS3. Since the current interface implementation allows only for non-stiff models without discontinuities, we only focus on studying the effect of the sparsity of a model on the CPU performance of both algorithms. To achieve our goal, we need to be able to automatically generate models of arbitrary size and sparsity.

For the benchmark, we chose to generate linear models of the form:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} \tag{8}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the vector of state variables. Matrix $\mathbf{A}$ controls the dynamics of the generated system. Since we needed to control the eigenvalues of the system and avoid producing stiff systems, we constructed $\mathbf{A}$ as follows:

1. Generate real-valued random eigenvalues drawn from a Gaussian distribution : $eig \sim -\mathcal{N}(5, 2)$.
2. Create a diagonal matrix $\mathbf{D} = diag(eig)$.
3. Create a random orthogonal matrix $\mathbf{M}$.
4. Then matrix $\mathbf{A} = \mathbf{M} \cdot \mathbf{D} \cdot \mathbf{M}^T$ has the desired eigenvalues eig.

The constructed matrix $\mathbf{A}$ is a full matrix. **Sparsity** s is defined as the number of connections to every state variable. To achieve a certain sparsity level s, we set the $n - s$ absolute smallest elements of each row of $\mathbf{A}$ to zero. The absolute smallest elements were eliminated in order to minimize the impact on the eigenvalue locations of the resulting matrix $\tilde{\mathbf{A}}$. Having constructed a matrix $\tilde{\mathbf{A}}$ of given size n and sparsity s, it is straightforward to generate an equivalent Modelica model.

For the comparison simulations, the OpenModelica 1.5.0 environment was used with DASSL as the standard solver. The tolerance was set to 10^{-3}, and the simulation end time was set to 3 sec for both OpenModelica and PowerDEVS. Furthermore, the output file generation was disabled for both environments in order to measure the pure simulation time.

For each parameter configuration (n, s), 100 Modelica models were randomly generated and given as input to the

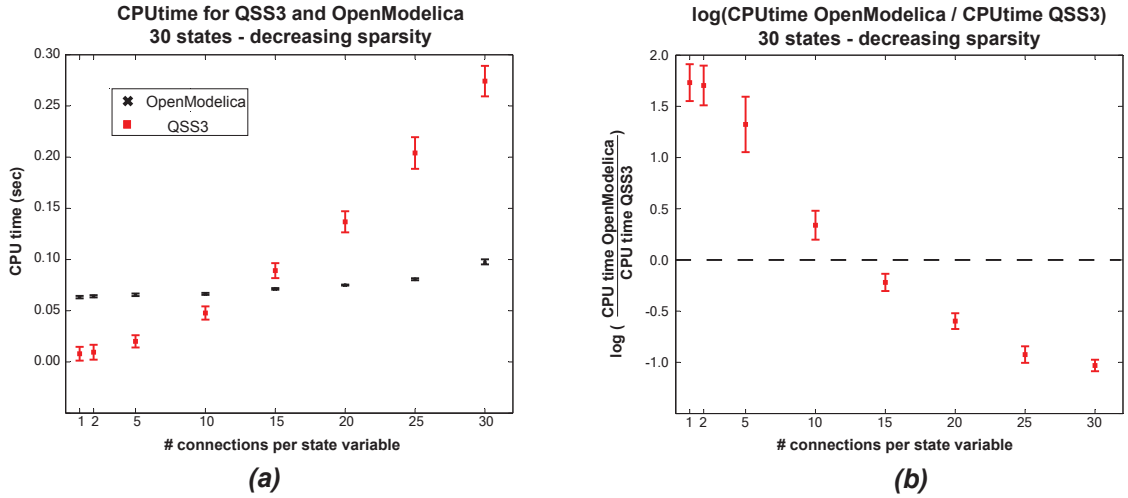

Figure 4. Simulation results for automatically generated models with $n = 30$ state variables and varying sparsity. In (a), the mean CPU time needed for simulating these systems using QSS3 and OpenModelica is plotted (red squares and black crosses, respectively).In (b), the logarithm of the ratio between the CPU time of OpenModelica and QSS3 is shown. Observing both plots, we conclude that QSS3 is more efficient than DASSL when simulating sparse systems ($s < 13$) Most real-world large-scale systems belong to that category.

standard OpenModelica compiler and via the OMPD interface to PowerDEVS. The CPU time needed for the simulation was measured for each generated executable. In order to obtain more reliable results, each simulation was repeated 10 times, and the median over all 10 repetitions was considered as the CPU time needed for each simulation. For each parameter setting (n, s) the mean of the CPU time measurements is reported along with $\pm$ 1 standard deviation.

Two types of experiments were conducted. First, the number of states n was kept constant with the sparsity s being varied, and then the reverse procedure was performed by fixing the sparsity s while varying the number of states. For each experiment, the CPU time was measured for simulations performed with DASSL and QSS3, respectively.

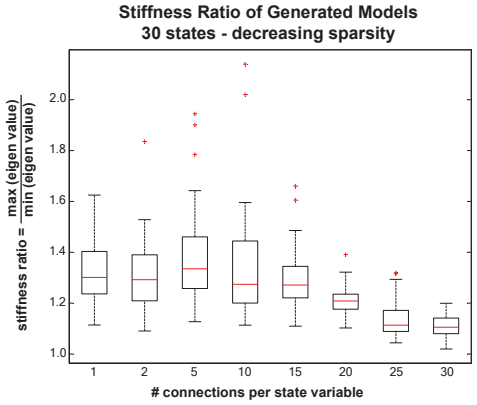

Figure 5. The stiffness ratio of the generated models in Sec. 5.3 is depicted. We observe that the resulting models exhibit approximately the same stiffness for different sparsity values.

5.3 Fixed Number of States - Varying Sparsity

In the first experiment, the number of states n in the generated models was fixed to $n = 30$ states. Then starting from a sparsity of $s = 1$, meaning that only one non-zero connection was preserved for the computation of each state derivative, the number of connections was gradually increased until we reached the full model with $s = 30$ connections. In Fig. 4(a), the obtained average CPU time is plotted against s for QSS3 (red squares) and for DASSL (black crosses). We observe that QSS3 is considerably faster than DASSL for sparse models (low values of s), whereas DASSL gets more efficient than QSS3 for approximately $s \geq 13$ connections.

This result is not overly surprising. In fact, it was expected. Whereas QSS3 benefits from sparsity as it only updates states and state derivatives asynchronously if and when there is a need for it, DASSL benefits in a fully connected model from the fact that it calls the right-hand equations only once per iteration, i.e., maybe thrice per step, whereas QSS3 updates each state derivative separately. In a non-sparse model, there is nothing that QSS3 can exploit.

However, real-world large-scale systems are always sparse. It simply doesn't happen in real life that a state derivative in a 30-th order model depends on all 30 state variable. Usually each state equation depends on two or three state variables only. Hence the case with $s = 3$ is probably the most relevant for all practical purposes.

The areas where each algorithm is superior in terms of CPU performance are more clearly visible in Fig. 4(b), where the logarithm of the ratio between the CPU times of DASSL and QSS3 is plotted against s. A positive log-ratio value means that QSS3 is more efficient than DASSL, whereas a negative value indicates the opposite.

In Fig. 5, the stiffness ratio of the generated models is plotted. The stiffness ratio is defined as the ratio of the largest to the smallest eigenvalue of matrix $\tilde{\mathbf{A}}$ and is used as an indicator of a model being stiff or not. The resulting stiffness ratio of the models used is kept smaller than 10 and does not vary much with s. Therefore, we can conclude that the differences in the measured execution times are not a result of a significant difference in the stiffness ratios of the models used.

5.4 Fixed Sparsity - Varying Number of States

Next, we studied how the simulation performance is affected when the number of connections per state variable is kept constant, whereas the number of states is modified. In Fig. 6, the sparsity s is set to 2 inputs per state, and the number of states is increased from 10 to 150. We observe that up to $n = 130$ states, QSS3 is more efficient in terms of needed CPU time. On the other hand, if we increase the sparsity s to 5 connections per state variable, the range where QSS3 is superior to DASSL is reduced to about $n = 70$ states, as depicted in Fig. 7.

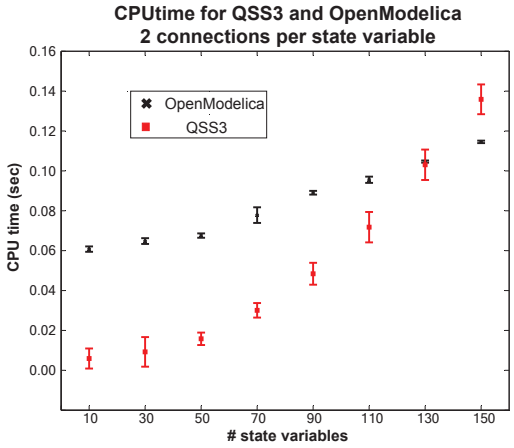

Figure 6. Sparsity s is set to 2 connections per state, and the number of states is increased from 10 to 150. We observe that up to 130 states, QSS3 is more efficient in terms of required CPU time. However, the computational cost of QSS3 increases with the model size much faster for QSS3 than for DASSL due to inefficiency in the current PowerDEVS implementation.

In both cases, the CPU time needed by QSS3 is increasing much faster than the CPU time that DASSL needs as the complexity of the model increases. This means bad news for PowerDEVS. PowerDEVS evidently does not handle large systems very well.

On the other hand, it can been shown that the computational load of the QSS3 algorithm grows only linearly with the number of states. Thus, the measurement results are in contradiction with theory.

The reason is that we measure not only the time needed by the QSS3 algorithm for integration, but also the time needed by PowerDEVS for administrating the simulation.

PowerDEVS was not designed to simulate large DEVS models. After all, the PowerDEVS users were expected to construct their models manually, which clearly limits the size of the models that users may want to simulate in PowerDEVS. For this reason, the designers of PowerDEVS did not spend much thought on an efficient implementation of the underlying simulation engine. In particular, block are implemented as a linearly linked list, and PowerDEVS traverses that list in a linear fashion when looking for the next model to be executed. This kills the overall efficiency of the implementation, and what we measured in Fig. 6 for larger numbers of states is primarily not simulation time, but rather the quadratic growth pattern of a linear search algorithm. Re-implementing the blocks as an equilibrated binary tree, a trivial programming exercise, will reduce the growth pattern of the search time to $n \cdot \log n$ and will make PowerDEVS perform considerably better for larger models.

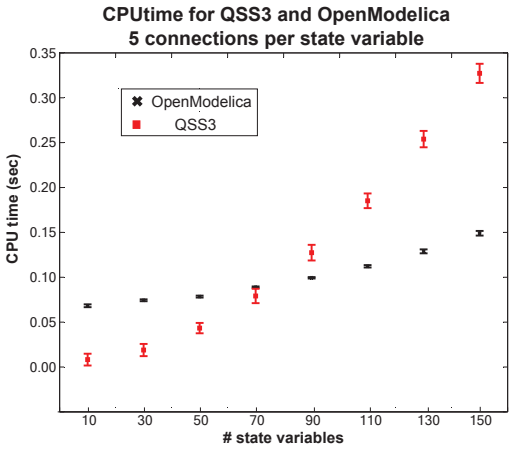

Figure 7. Sparsity s is set to 5 connections per state, and the number of states is increased from 10 to 150. Now, QSS3 is more efficient for up to 70 states.

6. Discussion

6.1 Conclusions

In this article, an interface between the OpenModelica environment and PowerDEVS is presented and analyzed. The OPMD interface implemented until now does not handle discontinuities yet, but it represents the first effort to automatically bridge the gap from the powerful Modelica modeling language standard to the also very powerful state-quantization based (QSS) simulation methods. The currently available interface allows a Modelica user to simulate arbitrarily complex non-stiff Modelica models without discontinuities using the PowerDEVS simulation software. Future extensions of the interface shall handle stiff and discontinuous models as well.

Preliminary results exhibit the superiority of QSS3 over the standard DASSL solver when simulating sparse models. More rigorous experiments will need to be performed in order to reach concrete conclusions about the performance of each algorithm. An inefficiency in the current implementation of PowerDEVS when simulating large systems was observed and will be addressed in the next distribution of PowerDEVS.

6.2 Future Work

We have shown that the implemented OMPD interface successfully allows a user to simulate an arbitrary Modelica model without discontinuities using PowerDEVS and the QSS methods. However there are open problems that need to be addressed in the future.

The OMPD interface should be extended to cover models with discontinuities. QSS methods are intrinsically well suited for simulating discontinuous models. Therefore, it is of great importance to add this functionality to the OMPD interface. Most importantly, we shall then be able to perform large-scale comparisons between DASSL and QSS algorithms for a variety of real-world models that are inherently discontinuous.

On the other hand, as discussed in Section 5.4, PowerDEVS needs to be implemented more efficiently in order to take advantage of all theoretical properties of the QSS methods. In particular, the current simulation engine is quite inefficient in the way it searches through the blocks to find the one that produces the next event in the simulation. This issue can be addressed by employing a more efficient search strategy, e.g. by organizing the atomic models in an equilibrated binary tree structure. These modifications are already being implemented, and we are looking forward to incorporating these modifications in the next PowerDEVS distribution.

7. Acknowledgements

We would like to acknowledge the help and support from the PELAB group at Linköping University and in particular Per Östlund, Adrian Pop, Martin Sjölund, and Prof. Peter Fritzson.

References

[1] Tamara Beltrame and François E. Cellier. Quantised state system simulation in dymola/modelica using the devs formalism. In *Modelica*, 2006.

[2] François E. Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer-Verlag, New York, 2006.

[3] Peter Fritzson, Peter Aronsson, Hakan Lundvall, Kaj Nystrom, Adrian Pop, Levon Saldamli, and David Broman. The openmodelica modeling, simulation, and development environment. *Proceedings of the 46th Conference on Simulation and Modeling (SIMS'05)*, pages 83–90, 2005.

[4] Peter Fritzson and Peter Bunus. Modelica-a general object-oriented language for continuous and discrete-event system modeling and simulation. In *Annual Simulation Symposium*, pages 365–380, 2002.

[5] Peter Fritzson and Vadim Engelson. Modelica - a unified object-oriented language for system modelling and simulation. In *ECOOP*, pages 67–90, 1998.

[6] Ernesto Kofman. A second-order approximation for devs simulation of continuous systems. *Simulation*, 78(2):76–89, 2002.

[7] Ernesto Kofman. Quantization-based simulation of differential algebraic equation systems. In *Simulation, Transactions of the Society for Modeling and Simulation International*, volume 79, pages 363–376, 2003.

[8] Ernesto Kofman. Discrete event simulation of hybrid systems. *SIAM JOURNAL ON SCIENTIFIC COMPUTING*, 25:1771–1797, 2004.

[9] Ernesto Kofman. A third order discrete event simulation method for continuous system simulation. *Latin America Applied Research*, 36(2):101–108, 2006.

[10] Ernesto Kofman and Sergio Junco. Quantized-state systems: a devs approach for continuous system simulation. *Trans. Soc. Comput. Simul. Int.*, 18(3):123–132, 2001.

[11] Ernesto Kofman, Marcelo Lapadula, and Esteban Pagliero. Powerdevs: A devs-based environment for hybrid system modeling and simulation. Technical Report LSD0306, Universidad Nacional de Rosario, Argentina, 2003.

[12] Victor Sanz, Alfonso Urquía, François E. Cellier, and Sebastián Dormido. System modeling using the parallel devs formalism and the modelica language. *Simulation Modeling Practice and Theory*, 18(7):998–1018, 2010.

[13] Bernard P. Zeigler and J. S. Lee. Theory of quantized systems: formal basis for devs/hla distributed simulation environment. *Enabling Technology for Simulation Science II*, 3369(1):49–58, 1998.

Model verification and debugging of EOO models aided by model reduction techniques
* Work in Progress Paper **

Anton Sodja Borut Zupančič

Fakulteta za elektrotehniko, Univerza v Ljubljani, Slovenia,
{anton.sodja,borut.zupancic}@fe.uni-lj.si

Abstract

Equation-based object-oriented modeling approach significantly reduced effort needed for model implementation by releasing modeler of performing many error-prone tasks. An increasingly more complex models can be built, preferably from components of different model libraries. However, complexity of the models complicate the process of verification – assuring that the model was implemented correctly and behaves as expected – and possible subsequent debugging. A cause of error in a model with over 1000 different equations can be often hard to find by the desk-checking method. This requires the development of new modeling environment tools for model understanding and automated discovery of the fault causes.

The difficulty of designing such tools in EOO modeling environments is linked to the difficulty of mapping simulation form to the model sources. Furthermore, debugging of complex models consisting of over thousand equations by traversing each equation may be very ineffective, especially when the fault has multiple and not very evident causes.

A model reduction methods is proposed and discussed as a method of verification. With model reduction methods it is possible to identify the most important parts of the model which have contributed to the specific model behavior. Because model reduction can be performed on original model representation, the difficulty of mapping simulation form back to model source is avoided.

Keywords verification, debugging of EOO models, verification by model reduction

1. Introduction

Important step in process of modeling and simulation is verification of the model. With verification we assure matching of our conceptual model with the implementation of the model.

Traditionally models were implemented in imperative programming languages and verification of the models could rely heavily to the established software debugging practices and techniques . However, declarative equation-based object-oriented modeling languages, like Modelica, introduced a new abstraction layer in implementation of the model to improve modeling process. Differential algebraic equations by which model is described can be entered directly and further such models can be combined into more complex models according to the rules defined by their interfaces. Such implementations of the models preserve topology of the modeled system and models are thus more evident and clear, but for simulation purposes such model must be preprocessed – translated into the simulation form. Translation is performed automatically.

Since the modeler is no longer acquainted with the simulation form of the model, the paradigm of the model verification and debugging has changed and we can no longer effectively use software debugging tools developed for imperative programming languages. However, verification and debugging of declarative modeling languages is still not solved adequately and it remains a challenging research topic.

2. Overview of verification techniques

A number of verification techniques have been developed, spanning from very informal approaches to formal mathematical proofs of correctness. Whitner and Balci [13] categorized verification methods into six distinct perspectives based on increasing level of mathematical formality: informal, static, dynamic, symbolic, constraint and formal analysis. Effectiveness of verification usually increases as method becomes more formal, but on behalf of increased complexity.

2.1 Informal analysis

Informal analysis techniques, such as desk checking, are most commonly used verification strategies where model is evaluated using the human mind. The evaluations can be made by mentally exercising the model, reviewing the logic behind the equations, algorithms and decisions, and

3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. October, 2010, Oslo, Norway.

examining the effects that the various implementations will have on the overall outcome of the model.

Declarative modeling languages specifically improve efficiency of the informal analysis of the model, since implementation of the model is close to the form of conceptual model with preserved topology of the modeled system.

However, informal analysis is very time consuming and its success depends on the level of knowledge and expertise of the individual, which comes into concern when very complex models are built with an aid of pre-prepared model libraries with implementation of the components that may not be completely known to the user.

2.2 Static analysis

Static analysis is a verification method where only the source code of the model is analyzed without actually performing simulation.

It is the best supported verification method by EOO modeling tools. The most important aspect of it is structural analysis – checking that number of variables match the number of equations in the model (resulting system of equations derived from model is well-constrained) and in case if over- or under-constrained system of equations was detected, a tool (debugger) must be able to report the location of the error consistently with the user's perception of the simulation model and possibly suggest the right error-fixing solution [1]. In Modelica language standard 3.0, restrictions have been introduced into the language in order to force matching of a number of unknowns and equations on every hierarchical level (in each submodel) [8]. This has improved efficiency of the debugging over- and under-constrained models since it is possible to easily determine a submodel with too many or too few equations.

Object-oriented modeling where variables are also objects with many additional properties besides value enables also other kind of static analysis, for example, unit checking [6].

The advantage of static analysis in EOO modeling is that costly translation and simulation of the model is avoided, but it can not fully verify that intentions of the modeler are being met or determine the model's behavior.

2.3 Dynamic analysis

Verification by dynamic analysis is accomplished by evaluating model's simulation results. In general this a complex task since it is not easy to determine what to test and how to test it. Furthermore, it can be complicated to interpret the analysis' results.

The EOO modeling introduce additional difficulties into verification by dynamic analysis due to optimization performed during translation of the model when a lot of information about the original model's structure is lost. Mapping the simulation form back to original model is thus a very challenging task.

Currently, no modeling tool supporting Modelica provides even the simplest run-time debugging capabilities that could be used for inspecting of failed tests. However, there were some prototypes of automated debuggers for EOO languages developed [1, 9]. They are based on the principles of debugging of the programming languages. The debugging strategy is interactive and based on the user's choice of the variables which trajectories are wrong and according to these variables the simulation form of the model is sliced so that only parts (equations and algorithms) having influence on selected-variables' calculation are shown to the user. A dependency graph is built where nodes represent equations that contributed to the result connected by directed edges labeled by variables or parameters which are inputs or outputs from or to the equation represented by the node. Then the user is able to classify a variable as variable with wrong value or classify an equation as correct (which results in rebuilding the dependency graph) and modify some of equations and variables' values interactively. Each equation in dependency graph is also mapped to model source.

Although debugging methods [1, 9] provide significant advantage over no debugging tools at all, they have some serious deficiencies. The user is basically still operating on the level of model's simulation form, even if mapping to model code positions is provided. Also scalability, when dealing with large models with many equations, is problematic while considering each equation one by one is very impractical.

A special topic of dynamic analysis is resolving numerical problems that might arise during simulation. However, this is very advanced topic, while user must posses a good knowledge of the simulation form to which model was translated and properties of the numerical solver.

2.4 Symbolic analysis

Verification by symbolic analysis addresses some drawbacks of the dynamic analysis, namely the inability to verify all possible test cases. Verification by symbolic analysis seeks to determine the behavior of the model during simulation by providing symbolic inputs to the model.

2.5 Constraint analysis

Constraint analysis verifies on the basis of comparison between model assumptions and actual conditions arising during simulation of the model. The model assumptions are made already during the development of the conceptual model. In Modelica they can be checked by means of assert statement provided by language standard [7] and are extensively used in Modelica Standard Library, for example, in *Modelica.Media* it is in that way assured that the medium equations are never used outside the valid temperature range.

The disadvantage of constraint analysis is reliance on formal model specifications which is needed to effectively state and place the assertions. Creating a formal specification is a difficult task.

2.6 Formal analysis

Formal analysis is based on formal mathematical proof of correctness. It usually can not be applied even on the most simple models and basically it has no practical value so far.

3. Problematics of verifying EOO models

Modeling is a process of extraction, organization and representation of the knowledge about physical system [2].

The contribution that equation-based object-oriented modeling approach brought to the process of modeling is implementation of the model close to the conceptual model, i.e., model is stated in acausal form and topology of the system is preserved. In contrast to traditional procedures where model was implemented in imperative languages, automatic translation of the model relives the user of tedious and error-prone task of manual manipulation of the model's equations.

EOO modeling approach also enabled truly reusable models and consequently many model libraries have been developed. Rich selection of already prepared components allows building relatively complicated and "error-free" models with a little effort. However, although once common errors due to implementation in imperative languages are no longer an issue, there can be identified three types of faults emerging in EOO models which can be exposed in simulation results [1]:

- when parameter values for the model simulation are incorrect
- when the equations that specify the model behavior are incorrect (violate physical laws)
- when submodels are used inappropriately (assumptions about the system are violated)

These faults can be found by either failed assert statement or by inspection of the simulation results (failed test).

When a fault is found in a model, the cause or several of them (if the fault have multiple causes) have to be found and corrected in the model implementation. For example, if certain quantity in the model increased unbounded, although stable system with bounded input signal have been simulated, it is a plausible assumption that some equation is wrong or some parameter has been assigned an unphysical value. A strategy of traversing the equations related to this quantity will certainly lead to the solution of the problem, although in a large-size model this procedure is somehow laborious. A debugger based on interactive dependency graph proposed by [9] may be very useful in such case.

However, a case when, for example, model's response exposes unexpected initial undershoot is much more complicated to resolve. Initial undershoot may be a property of the system (a nonminimal phase), only property of the model (introduced by some modeling assumptions and simplifications) or an error in the implementation. A simple traversal of the equations related to the trajectory exposing initial undershoot may not provide a proper insight, especially if a large-size model built from complicated components is under consideration (e.g., component *DynamicPipe* from the library *Modelica.Fluid* can consist of over 100 equations distributed to 10 models from which original component is extended).

Verification of complex EOO models should be supported by a tool that directs the modeler towards the important parts of the model regarding a certain behavior of the model (exposed by trajectories obtained by simulation).

3.1 Model reduction techniques

For some modeling purposes, most notably structural and control design, large size models that include detailed dynamics are undesired, since determining the major design parameters and their relationship to the system performance is difficult [5].

Model reduction techniques represent also an important aspect of the systems analysis, when a low-dimensional model can provide qualitative understanding of the phenomena under consideration [4]. An important property of the model reduction methods when used in system analysis or for structural design is that it generates a *proper model*, i.e. reduced model with the minimum complexity required to meet the performance specifications and possessing physically meaningful parameters and states [5].

In attempt to reduce the complexity of the model (and number of its parameters), a number of mixed numerical-symbolic model reduction techniques have been developed and successfully applied [11, 12, 4, 5].

Model order reduction techniques consists of running a series of simulations, ranking the individual coordinates or elements by the appropriate metric and removing those that fall below a certain threshold [3].

The most straightforward metrics for reduction of the proper models is related to energy or power. Method of Rosenberg and Zhou [10] removes bonds with low power from a bond graph model, Luca [5] introduces activity – the time integral of the absolute value – and Ye and Youcef-Youmi [14] reduces bond graph models by eliminating bonds with smallest associated energy in comparison to its neighbors. Chang *et al.* [3] eliminate system's states with low associated energy in the model comprised of Lagrangian subsystems with force interconnections based on Lyapunov stability. Metrics related to energy or power requires modeling formalism with clearly defined components' energy and power respectively. Method of Sommer *at al.* [12] consists of term substitution and deletion (as well as on some other simplification) in the equations of the DAE system derived from the model. The term are ranked according to their influence on the output error which is defined as a difference of original and reduced model's output. The resulting simplified system of equations can be interpreted again in the form of component equations and can be mapped to a reduced model scheme.

For the simulations that are performed to obtain the error estimates, excitation of the model must be selected in such way that a valid model is obtained in a desired frequency range.

3.2 Verification aided by model reduction techniques

Verification by dynamic analysis as well as subsequent debugging in equation-based object-oriented approach could be improved substantially by using model reduction techniques.

When a behavior of the model obtained by simulation is not as expected or even erroneous, an explanation is sought. It is sensible to first look at that components or (terms of) equations and parameters of the model that have the most influence on the dominant system dynamics and trajectory of the model's variable of interest respectively.

In a large-size models made up of components from various model libraries which implementation is not precisely known to the user, it is not apparent which components or equations of the model have the greatest impact on the response of the model, i.e. on selected variable's trajectory (or part of it). Determination of the most influential components can be automatized by using ranking algorithm known from the model-reduction methods. The user could focus only on the few components contributing most significantly to the simulation results and potentially extend his/her search to lower ranked components. Because the ranking is affected by selected time-window, components can be separated also according to the impact they have during different time of simulation. For example, in the steady-state, dynamic terms (those that include time derivatives) are totally irrelevant, while at the beginning of the transient, terms of equations describing the fast dynamics of the system are those which are worth of the most attention.

Furthermore, on behalf of the user, proper reduced model could be generated and the user could further experiment on the reduced model which can be much more easily understand. Because all the parameters and states of the reduced model are physically meaningful, the changes of the reduced model could be easily merged with original model.

An important advantage of the model reduction of EOO models is also that user never needs to consider the translated form of the model, while model reduction is performed on the original representation of the model.

However, in general modeling language such as Modelica, models can be implemented in various ways, for example, entirely in textual form as a set of equations or by connecting basic components from the Standard Library in the graphical interface. That implies using different model reduction strategies and possibly also different metrics.

4. Conclusions

Use of model reduction techniques for verification and debugging have been proposed. Methods originated from model reduction techniques can be used to rank the components (equation terms) of the model with greatest impact on the model behavior (selected trajectory) and proper reduced models can be generated on behalf of the user. Reduced models are easily to comprehend by the user and while proper reduced models have physical meaningful parameters, changes done to the reduced model can be easily merged to the original model.

The advantage of debugging aided by model reduction methods over traditional software debugging methods is that since the user does not need to consider the simulation form of the model anymore, the difficult mapping of translated equations to model source is not needed.

However, most model reduction methods requires specific modeling formalism (e.g., bond graphs) and can be restricted to specific physical domains. The usefulness of verification and debugging tools based on model reduction techniques is therefore limited in a general EOO modeling languages such as Modelica.

References

[1] Peter Bunus. *Debugging Techniques for Equation-Based Languages*. PhD thesis, Linköping University, 2004.

[2] F. E. Cellier and E. Kofman. *Continous System Simulation*. Springer Science+Business Media, New York, 2006.

[3] Samuel Y. Chang, Christopher R. Carlson Carlson, and J. Christian Gerdes. A lyapunov function approach to energy based model reduction. In *Proceedings of the ASME Dynamic Systems and Control Division – 2001 IMECE*, pages 363–370, New York, USA, 2001.

[4] Sanjay Lall, Petr Krysl, et al. Structure-preserving model reduction for mechanical systems. *Physica D*, 284:304–318, 2003.

[5] Loucas Sotiri Louca. *An Energy-based Model Reduction Methodology for Automated Modeling*. PhD thesis, University of Michigan, 1998.

[6] S. E. Mattsson and H. Elmqvist. Unit checking and quantity conservation. In *Proceedings of the 6th Modelica Conference*, pages 13–20, Bielefeld, Germany, 2008.

[7] Modelica Association. *Modelica Specification, version 3.1*, 2009. `http://www.modelica.org/documents/ModelicaSpec31.pdf`.

[8] H. Olsson et al. Balanced models in modelica 3.0 for increased model quality. In *Proceedings of the 6th Modelica Conference*, pages 21–33, Bielefeld, Germany, 2008.

[9] A. Pop and P. Fritzson. A portable debugger for algorithmic modelica code. In *Proceedings of the 4th Internationl Modelica Conference*, pages 435–443, Hamburg, Germany, 2005.

[10] R. Rosenberg and T. Zhou. Power-based model insight. In *Proceedings of the ASME WAM Symposium on Automated Modeling for Design*, pages 1–67, New York, USA, 1988.

[11] P. Schwarz et al. A tool-box approach to computer-aided generation of reduced-order models. In *Proceedings EUROSIM 2007*, Ljubljana, Slovenia, 2007.

[12] Ralf Sommer, Thomas Halfmann, and Jochen Broz. Automated behavioral modeling and analytical model-order reduction by application of symbolic circuit analysis for multiphysical systems. *Simulation Modelling Practice and Theory*, 16:1024–1039, 2008.

[13] R. B. Whitner and O. Balci. Guidelines for selecting and using simulation model verification techniques. Technical report, Department of Computer Science, Virgina Polytechnic Institute and State University, Blacksburg, Virginia, 1989. Technical Report TR-89-17.

[14] Y. Ye and K. Youcef-Youmi. Model reduction in the physical domain. In *Proceedings of the American Control Conference*, pages 4486–4490, San Diego, CA, USA, 1999.