# Towards a Modular and Accessible Modelica Compiler Backend

Jens Frenkel[+], Günter Kunze[+], Peter Fritzson[*], Martin Sjölund[*], Adrian Pop[*], Willi Braun[#]
[+]Dresden University of Technology, Institute of Mobile Machinery and Processing Machines
[*]PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, SE-581 83 Linköping, Sweden
[#]FH Bielefeld, University of Applied Sciences
{jens.frenkel, guenter.kunze}@tu-dresden.de,{peter.fritzson,martin.sjolund,adrian.pop}@liu.se,
willi.braun@fh-bielefeld.de

## Abstract

Modelica is well suited for modelling complex physical systems due to the acausal description it is using. The causalisation of the model is carried out prior to each simulation. A significant part of the causalisation process is the symbolic manipulation and optimisation of the model. Despite the growing interest in Modelica, the capabilities of symbolic manipulation and optimisation are not fully utilized. This paper presents an approach to increase the customisability, access, and reuse of symbolic optimisation by a more modular and flexible design concept. An overview of the common symbolic manipulation and optimisation algorithms of a typical Modelica compiler is presented as well as a general modular design concept for a Modelica compiler backend. The modularisation concept will be implemented in a future version of the OpenModelica compiler.

*Keywords: Compiler Backend; Optimisation; Interfaces.*

## 1 Introduction

Modelica is a multi-domain object-oriented equation based modelling language. It is mostly used for modelling and simulation of complex physical systems but other tasks like symbolic analysis of the equation system behind the model are possible as well. To accomplish those tasks a Modelica Compiler or Interpreter is needed. All Modelica compilers known to the authors are divided into a frontend and a backend. The frontend is used to extract the system of equations behind the object-oriented Modelica Model. This task is typically called flattening and forms a challenge on its own [1].

The Backend is the part of the compiler which performs symbolic manipulation of the equation systems and generates code suitable for (efficient) execution or interpretation. These symbolic manipulations are dependent on the specific analysis task and the type of the equation system. To prepare a model for numerically stable simulation of a higher index system, for example, the backend could perform an index reduction in combination with a matching algorithm to assign all the equations and perform some symbolic optimisation in order to improve the runtime performance.

Because of the complexity of Modelica the typically implemented symbolic manipulation algorithms are generally applicable for all kinds of equation systems. On the one hand this is an advantage because the user does not need to deal with the tasks of symbolic manipulation to get the desired results. On the other hand it is a disadvantage for the advanced user or library developer because there is normally only restricted access to the symbolic manipulation algorithms.

To overcome these limitations the authors designed a concept for a modular Modelica compiler backend which to a large extent has been realised in the new reorganised implementation of the Open Source OpenModelica Compiler from OpenModelica version 1.6, excluding external APIs and phase reordering flexibility. The concept will include an interface for external tools, for example for symbolic optimisation, and a safe and task-dependent method to implement manipulation algorithms for equation systems.

The next chapter introduces the reader into the common symbolic manipulation process for Modelica Compilers. Chapter three presents an overview of advanced symbolic manipulation algorithms which may speed up the simulation. The options for the user to access and customisation of symbolic manipulation algorithms are discussed within chapter four.

After concluding the previous chapters, chapter five presents the concept of the new modular compiler backend. Chapter six continues with the basic concept for its implementation and chapter seven presents interfaces of the modular compiler backend in more detail.

## 2 Common Symbolic Manipulation in Modelica Compiler Backends

Simulation itself is the most common usage of Modelica Models. The following section gives a general overview on the tasks of the backend of the Modelica compiler and applies to almost all implementations, see also Figure 1.

The output of the frontend is usually a flat representation of the Modelica model containing all variables with their properties, equations and algorithms. The first step of the backend is to analyse the flat model representation and extract additional information. Extracted information could for example be the candidates for states, implicit discrete variables and the variables with time-dependency. Candidates for states are differentiated variables and variables with the property `stateSelect` equal to `StateSelect.prefer` or `StateSelect.always`.

With the change of Modelica 3.2 the derivative operator "der" may have an expression as its argument. With the ambition of an efficient code generation all such expressions have to be symbolically differentiated with respect to the time dependent variables. Only if it is not possible to perform the symbolic differentiation an additional variable should be introduced.

The next steps of the symbolic optimisation are to remove simple equations and equations with no time dependency. Simple equations are of the form "a=b", "a=-b" also called "alias equations". Alias equations appear quite often because of the connect-equations concept. Equations with no time dependency are composed of constant or parameter variables and variables calculated based on constants or parameters.

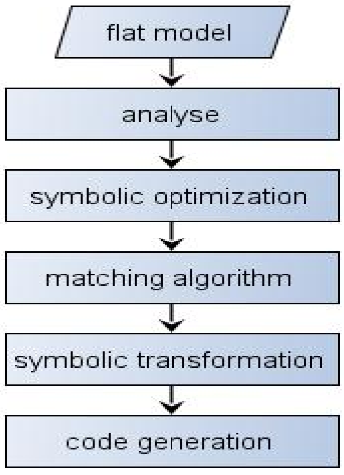

**Figure 1:** Common symbolic manipulation process for Modelica models.

Subsequently follows the matching algorithm which is one of the main phases of the backend. This algorithm has to find an equation for each variable the equation is solved for. If the system should be solved as an explicit ordinary differential equation (ODE) system, but is a higher index system, the matching algorithm works together with an index reduction algorithm.

To get the block lower triangular form (BLT-Form) Tarjan's algorithm [10] is performed. This algorithm sorts the equations in the order they have to be solved.

The next step is the symbolic transformation and the collection of additional information, for example the occurring of zero crossings. Within the symbolic transformation phase the information from the matching algorithm is used to solve certain equations symbolically. This has to be done for single equations as well as for linear or nonlinear equation systems. To solve certain equations symbolically means to solve an equation or a system of equations for the desired variable or variables or provide all information to use an applicable numerical solver.

The last part of the backend after the whole symbolic manipulation process is concerned with code generation.

## 3 Advanced User Supported Symbolic Manipulation

In addition to the methods mentioned in Section 2 there are specialised symbolic manipulation algorithms which may speed up the simulation. These include algorithms like:

- dummy derivative with dynamic state selection
- tearing
- inline integration
- function inlining.

All to the author's known Modelica compilers do not use these algorithms automatically, apart from tearing. Normally the users have to set special compiler flags or adjustments or use specific keywords, for example annotations for code generation.

The Dummy Derivative method [3] is a commonly used index reduction algorithm for higher index differential algebraic equations (DAE) systems. Using the Dummy Derivatives method may for some systems result in a singular Jacobian. A well-known example is the description of a planar pendulum described using the Cartesian coordinates.

To overcome these mathematical difficulties dynamic state selection could be used. The basic idea of dynamic state selection is to change the section of state variables at run-time for higher performance and numerically stable simulation, see [4] and [5].

Large algebraic equation systems are typical of Modelica models. The tearing algorithm is a method to reduce the size of large algebraic equation systems by dividing strongly connected components in the BLT graph into smaller sub-systems of equations which could be solved more efficiently.

The basic concept of tearing is to make an assumption about one or several variables to be known. With this assumption the matching algorithm is continued. The torn equation systems are divided into several reduced algebraic equation systems and a set of explicit assignments. In case of a linear algebraic equation system the relaxation algorithm or a solver for nonlinear systems, for example the Newton Iteration method, can be used to solve the algebraic system. Detailed information on the tearing algorithm can be found in Cellier and Kofmann [6].

For simulations with strong requirements on execution time like hardware in the loop simulation, it may be beneficial to merge the integration method and the equation system. This method is called inline integration and yields an equation system where the numerical solver is eliminated as an explicit software component. Inline integration is for minimising call overhead and enabling additional symbolic manipulations for example removing the division of the time step on both sides of an equation. For further information about inline integration [6] and [7] are recommended.

The Modelica Specification 3.2 provides within chapter 17.3 annotations for code generation. The annotations:

- `Evaluate`,
- `Inline` and
- `LateInline`

are useful for symbolic optimisation. For example the Modelica MultiBody Library uses these annotations.
Performing the specialised symbolic manipulation algorithms the common symbolic manipulation process from Figure 1 is expanded to the advanced symbolic manipulation process shown in Figure 2. For some optimisation methods, for example inline integration, it is useful to run the matching and sorting algorithm repeatedly. Hence the graph presented in Figure 2 has in this case a loop within the second optimisation and matching algorithm stage.

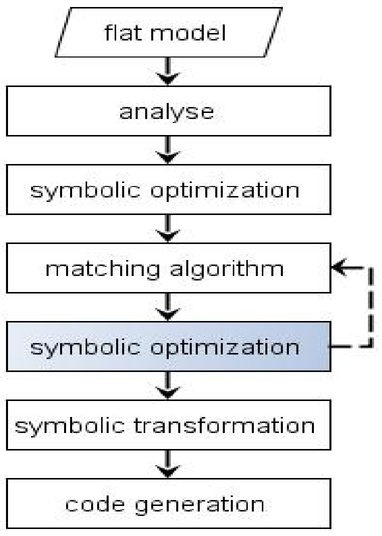

**Figure 2:** Advanced symbolic manipulation process for Modelica models.

## 4 Access to and Customisation of Symbolic Manipulation

Letting the user or library developers provide the compiler with specific information on how the symbolic manipulation algorithms should optimise the equation system can be achieved through several options:

1. Usage of specific keywords introduced into the language
2. Usage of optional compiler flags/settings
3. Exporting the systems of equations in a symbolic standardized and easy to use format and perform the optimisation with another tool
4. Development of a new Modelica compiler

However, none of the above mentioned options are practical in case a special optimisation is needed. The reasons are:

- Option 1 requires a long term standardisation process for the new methods.
- Option 2 is tool-specific, and therewith restricted.
- For Option 3 there is no standardized and easy to use format available yet nor does the common compiler support an export or import of the system of equations. Even if there is a support for export available the user would have to provide a code writer for the target format.
- Option 4 is not possible for most of the users.

In case an advanced user requires a symbolic manipulation algorithm, the compiler has to fulfill the following requirements:

- The export and import of the system of equations in a symbolic standardised and easy to use format. (A promising development is shown in [2].)
- An increasing number of symbolic manipulation algorithms.
- The possibility to define and use a symbolic manipulation algorithm in a safe and application-related way.

## 5 Modular Compiler Backend

To achieve the requirements of a more open and customisable access to symbolic manipulation capabilities the backend of the compiler needs a more general design concept. Keeping the tasks presented in Section 2 in mind the main tasks of the backend can be summarized into five phases:

- input phase
- pre-optimisation phase
- transformation phase
- post-optimisation phase
- output phase

Based on these phases and the concept of transferring the system of equations represented by a data object from one phase to the next, an equation system pipeline is formed as shown in Figure 3.

The transformation phase achieves a matching using index reduction methods and sorting the equations. Different modules for index reduction should be available as shown in Figure 3 within the left upper inner box "DAE-Handler for Index Reduction".

For some optimisation methods, for example inline integration, it is useful to run the matching and sorting algorithm repeatedly. Hence the pipeline presented in Figure 3 has a loop within the post-optimisation and transformation phase.

The output phase transforms the system of equations into the desired target format. This phase is the combination from the symbolic transformation and the code generation from Figure 2. Supposable target formats are a simulation executable or a functional mockup unit or if not a simulation is desired a XML, Matlab or Python file export.

With a modularisation concept in mind each phase should be presented by one (input phase, transformation phase and output phase) or several (pre- and post-optimisation phases) modules. For a specific equation system the pipeline might be built from specific modules in a specific order. For another equation system the pipeline might be built from other modules in a different order. More precisely, it will be possible to use application-oriented pipelines for different equation system. As mentioned above a Modelica model is represented in the backend as an equation system. This equation system could be composed of equations, algorithms and sub-systems of equations. In case of a multi-domain model it may be beneficial to use different optimisation modules for the different sub-systems present in the model. The equation system pipeline frame work will allow the optimisation of different sub-systems using different optimisation modules. The challenge is to define proper rules for the differentiation of systems of equations.

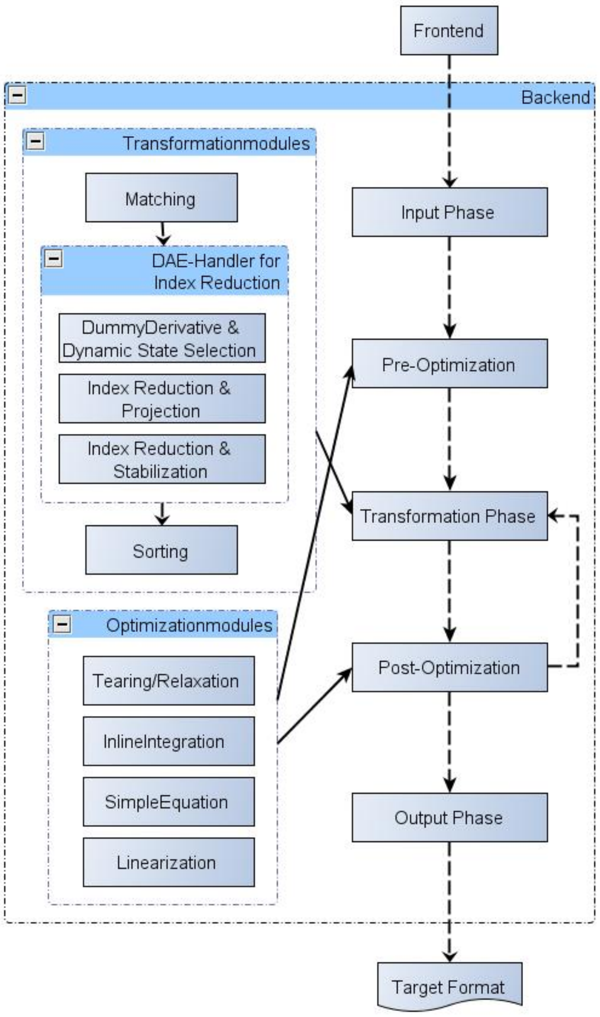

**Figure 3:** Data Flow in the Equation System Pipeline.

The system of equations has to be represented by a data object. This data object should basically include:

- the variables,
- the equations and
- the algorithms.

In case of an efficient implementation and to use the same interface for the pre- and the post-optimisation modules as desired, it is beneficial to store additional information within the data object and to classify variables, equations and algorithms into different groups. The additional information should be:

- an incidence matrix for the equations/algorithms and variables,
- the matching of variables and equations/algorithms solved for and
- the order of the equations/algorithms they have to be calculated in.

## 6 Concept for Implementation

To reduce the possibilities of introducing errors in developing the backend and support the developers in the most suitable way, three levels of complexity will be introduced by a class concept represented in Figure 4.

The basic level is a library for manipulating symbols and symbolic expressions, as shown in the inner-most box called “Symbolic Math Library” of Figure 4. The Symbolic Math Library comprises the four modules:

- Expression
- Symbol
- Simplify
- Solve

An expression consists of symbols, numbers and operators. In case of a non-scalar symbol an expression is used to point to the scalar elements of the symbol. Hence, the expression module and the symbol module use one another as shown in Figure 4. The two modules comprise functions to:

- generate expressions/symbols,
- transform them to other types,
- manipulate ,
- get something,
- traverse,
- compare and
- replace sub-expressions.

The module “Simplify” performs symbolic simplification of expressions and the module “Solve” implements functions to solve symbolical equations with the form “0=exp” for sub-expression.

Based on this symbolic math library an equation system library presents the second level, which supports all operations to access and manipulate equations, algorithms and equation systems. The “Equation System Library” comprises the four modules:

- Variable
- Equation
- Algorithm
- EquationSystem

This library provides functions to manipulate the system of equations, traverse them or replace variables with other variables or expressions for example. Both libraries have to preserve the consistency of the system of equations.

By using the library for symbols and expressions as well as the equation system library specific modules can be developed for each phase of the equation system pipeline within the third level in a safe and task-oriented way. The upper box “Modules for Optimisation” in Figure 4 for example represents optimisation algorithms such as:

- Tearing/Relaxation
- InlineIntegration
- Linearisation
- Remove Simple Equation

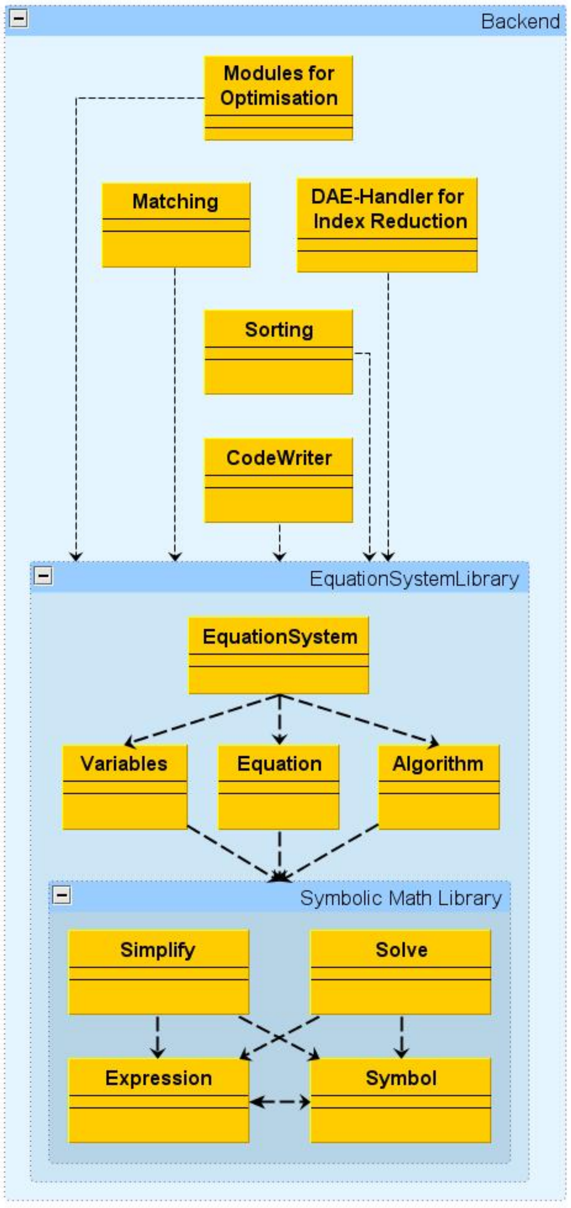

**Figure 4.** Concept for Implementation.

The developer can use the library functions of the equation system library and deal with the tasks of the optimisation algorithm.

The same principle holds for the box "DAE-Handler for Index Reduction" shown in Figure 4. For example to implement a classical handler for index reduction the developer can use the function to get the derivatives of the constrained equations and the function to replace equations. The classical handler for index reduction performs index reduction by taking the derivatives of the constrained equations.

The two modules to perform the sorting of the equations in BLT-Form and the module for code writing use the functions of the Symbolic Math Library and the equation system library as well as all other modules within the third level of the backend.

Furthermore, through the use of the library functions the implementation of debugging functionality for the system of equations is simplified and less error-prone because the information about the performed transformations would be added automatically by the library functions.

## 7 Interfaces and User Modules

With the possibility to export the system of equations during each phase the usability of the compiler for various analysis tasks can be significantly increased.
The next level of usability can be reached with an import of equation systems. This opens up the possibilities to:

- use the backend without the frontend,
- use other optimisation tools via an intermediate file format or an interface and
- store equation systems in different optimisation stages and run the optimisation algorithm again with improved optimisation modules.

As shown in Figure 5 the export is feasible by one or more modules in the pre- or past-optimisation phase. In Figure 5 an export to an XML-file format is shown as one example. The import has to be realized in the input phase to enable a consistency check of the system of equations.

Furthermore the presented design concept does not only include external interfaces as presented above. Additionally internal interfaces expand the usability of the compiler. With the advantages of the modularisation:

- replacement of modules,
- reusability of functions,
- distribution of the implementation complexity,
- limitation of complexity and
- a comfortable way to implement symbolic manipulation algorithms

the barriers to design, implement and test new algorithms for symbolic manipulation are reduced. Clearly, the symbolic simplification of a Modelica model is a difficult topic and it seems that only the Modelica compiler writers are able to provide such features. But, with a simplified access to the Modelica compiler development the Modelica compiler writer community can increase.

A comfortable way to implement those algorithms can be supported with the presented class concept from Figure 4, a development environment and the possibility to test the implemented algorithms. With the use of MetaModelica [8] for the Modelica compiler development it would be possible for an advanced Modelica user to implement his/her own modules. MetaModelica is an extension of the Modelica Language for Meta-programming facilities. Because of the similarity of MetaModelica and Modelica the additional requirements are marginal. Using the OpenModelica Development Environment (OMDev) the requirements mentioned above would also be fulfilled.

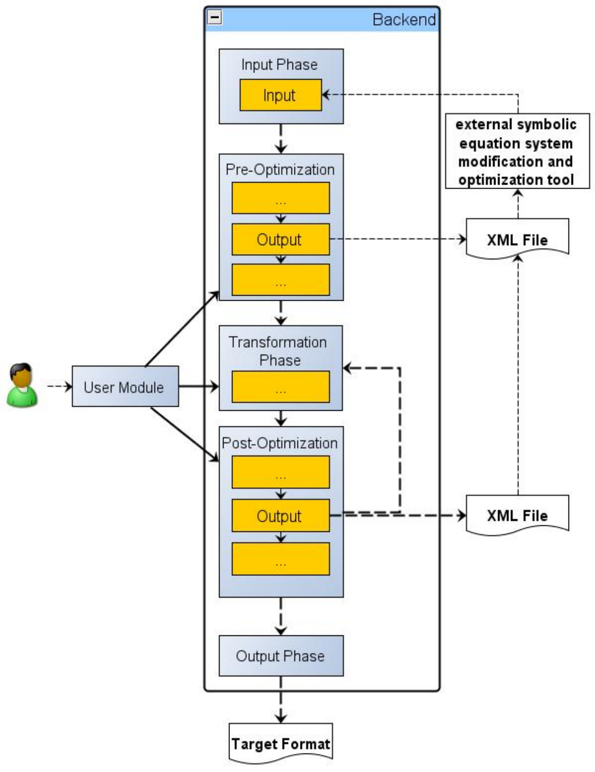

**Figure 5:** Interfaces for a modular Modelica compiler backend.

By offering the possibility to load the modules for symbolic manipulation dynamically during compile time the user could combine the compiler backend with his/her own modules. With a standardized data object format it would furthermore be possible to implement new modules in a different language, e.g. C/C++.

## 8 Conclusion

With the possibility for the user to define and use his/her own symbolic manipulation algorithms the possibilities of using Modelica for many different kinds of applications beside simulation can be significantly increased.

Users may on their own develop, analyse and get a deeper understanding of symbolic manipulation algorithms and get better results for specific applications.

Furthermore, the user could e.g. get access to higher index equation system solvers without the challenge to implement his/her own compiler.

By giving the user a safe and easy option to extend and adapt the functionality of the compiler to their own needs, it would be possible to enhance the development of the compiler to a new quality and wider practicability. This paper presents a contribution to an approach towards achieving this goal.

The described concept is partly implemented in OpenModelica 1.6. The modularization has been achieved, but the external API with the XML interface is still mostly future work, and an internal API for flexible phase ordering is also future work.

## References

[1] Peter Fritzson: Principles of Object-Oriented Modelling and Simulation with Modelica 2.1, Page 57ff, Wiley IEEE Press, 2004.

[2] Roberto Parrotto, Johan Åkesson, and Francesco Casella: An XML representation of DAE systems obtained from continuous-time Modelica models, EOOLT 2010.

[3] S. E. Mattsson and G. Söderlind: Index Reduction in Differential Algebraic Equations Using Dummy Derivatives, SIAM Journal on Scientific Computing, Vol. 14. No. 3, pp. 677-692, 1993

[4] S.E. Mattsson, H. Olsson and H. Elmqvist: Dynamic Selection of States in Dymola. Modelica Workshop 2000 Proceedings, pp. 61-67,

[5] P. Kunkel and V. Mehrmann: Index reduction for differential-algebraic equations by minimal extension, ZAMM, vol. 84, pp. 579–597, 2004.

[6] F. E. Cellier, E. Kofman: Continuous System Simulation, Springer, 2006.

[7] Bonvini, Donida, Leva: Modelica as a design tool for hardware-in-the-loop simulation, Proceedings 7th Modelica Conference, Como, Italy, Sep. 20-22, 2009

[8] A. Pop and P. Fritzson: MetaModelica: A Unified Equation-Based Semantical and Mathematical Modelling Language. In Proceedings of Joint Modular Languages Conference 2006 (JMLC2006) LNCS Springer Verlag. Jesus College, Oxford, England, Sept 13-15, 2006.

[9] P. Fritzson, A. Pop, D. Broman, P. Aronsson: Formal Semantics Based Translator Generation and Tool Development in Practice. In *Proceedings of 20th Australian Software Engineering Conference (ASWEC 2009)*, Gold Coast, Queensland, Australia, April 14 – 17, 2009.

[10] R.E. Tarjan: Depth First Search and Linear Graph Algorithms, SIAM Journal of Computting, 1, pp. 146-160, 1972.

[11] H. Lundvall and P. Fritzson: Event Handling in the OpenModelica Compiler and Run-time System. In Proceedings of the 46th Conference on Simulation and Modelling of the Scandinavian Simulation Society (SIMS2005), Trondheim, Norway, October 13-14, 2005. An extended version in Linköping University Press, www.ep.liu.se.