An Approach to the Calibration of Modelica Models

Miguel A. Rubio, Alfonso Urquia, and Sebastian Dormido

Departamento de Informatica y Automatica, UNED
Juan del Rosal 16, 28040 Madrid, Spain
{marubio,aurquia,sdormido}@dia.uned.es

Abstract. An approach to the calibration of Modelica models using genetic algorithms (GA) is presented. The functions required to perform the model calibration have been programmed in the Modelica language and structured in a Modelica library, called *GAPI*Lib. This Modelica library is intended for parameter estimation in any Modelica model, supporting simple-objective optimization. Model calibration with *GAPI*Lib does not require to perform model modifications. During the algorithm run, the user can interactively change the value of the GA parameters. In addition, *GAPI*Lib supports parameter sensitivity analysis, and it is well suited for parallel computing. *GAPI*Lib is a free library (available on http://www.euclides.dia.uned.es/GAPILib) that can be easily used, modified and extended.
The design, implementation and use of *GAPI*Lib are discussed in this manuscript. Its use is illustrated by means of a case study: the estimation of electrochemical parameters in fuel cell models, which have been composed using *FuelCell*Lib Modelica library.

1 Introduction

Genetic algorithms (GA) are used for model parameter estimation from experimental data. The main advantage of this technique lies in its robustness and simplicity. GA can be successfully applied for finding solutions in high-dimensional search spaces. The search range of the parameters can be changed during the algorithm run [1]. In addition, parallel implementations of genetic algorithms, intended to reduce the computation time, have been developed.

The use of GA for parameter estimation in Modelica models has been previously proposed by [2]. However, those authors implemented and ran the GA using Matlab/Simulink. As a consequence, those author's approach requires the combined use of Modelica/Dymola and Matlab/Simulink.

The lack of a freely-available Modelica library implementing GA, suited for parameter estimation in Modelica models, has motivated the implementation of the *GAPI*Lib library [3]. Two key advantages of *GAPI*Lib are its simplicity of use and generality: it can be applied for parameter estimation in any Modelica model, without needing to modify the model. The *GAPI*Lib library is freely available and can be downloaded from *http://www.euclides.dia.uned.es/GAPILib*

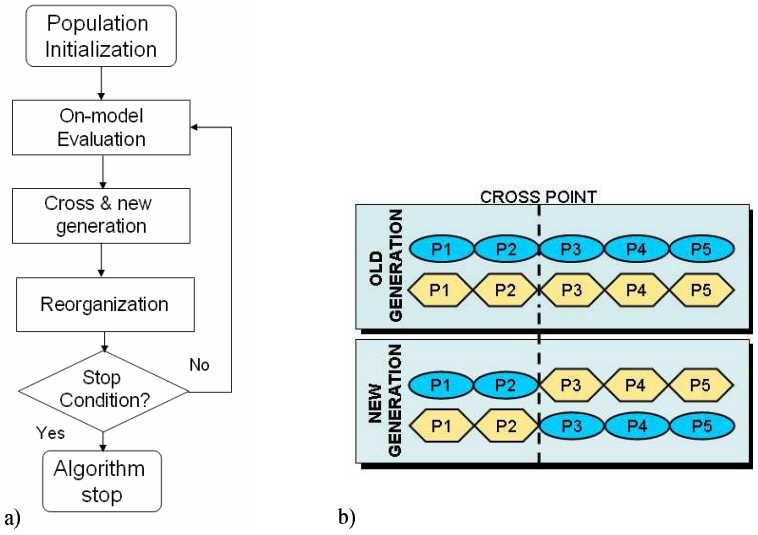

Fig. 1. a) Genetic algorithm supported by *GAPI*Lib; b) New generation obtained by crossover.

The fundamentals of the GA supported by the *GAPI*Lib library are briefly explained and the library structure is described. A new feature introduced since *GAPI*Lib version 1.0 [3] is discussed: the capability of changing the search range of the parameters during the GA execution. A procedure to compare the relative sensibility on the parameters is proposed. Also, a future development is discussed: support for parallel implementation of the GA. Finally, the use of *GAPI*Lib is illustrated by means of a case study: the estimation of electrochemical parameters in fuel cell models, which have been developed using *FuelCell*Lib Modelica library [4].

2 Model Calibration Using GA

The GA supported by the *GAPI*Lib library is schematically represented in Figure 1a [5–7]. The application of this algorithm will be illustrated by means of the simple model shown in Eq. (1).

$$y = a \cdot x^3 + b \cdot x^2 + c \cdot x + d \tag{1}$$

The GA is used to estimate the four parameters of the model (i.e., a, b, c and d) from the following set of experimental data:

$$\{x_i, y_i\} \qquad \text{for } i : 1, \cdots, N \tag{2}$$

The GA starts with an initial population, composed of $N_{POPULATION}$ individuals, which are randomly selected from the search space. Each individual of the population is formed by a group of chromosomes, which represents a solution to the problem. In case of the model shown in Eq. (1), each individual consists of a specific value of the parameters a, b, c and d. The j-th individual of the population is $I_j = \{a_j, b_j, c_j, d_j\}$. These initial values are randomly selected from the parameter search ranges.

Each individual of this initial population is evaluated by using a cost function. This function is used to calculate the validity of the population members. The cost function, evaluated for the j-th individual of the population, is the following:

$$f_j = \sum_{i:1}^{N} (y_i - \hat{y}_{i,j})^2 \tag{3}$$

where

$$\hat{y}_{i,j} = a_j x_i^3 + b_j x_i^2 + c_j x_i + d_j \tag{4}$$

The population members (i.e., $\{I_j\}$, with $j = 1, \ldots, N_{POPULATION}$) are sorted according to this criterion (i.e., the smaller f_j, the better). The sorted individuals can be represented as $I(1), I(2), \ldots, I(N_{POPULATION})$, where $I(1)$ is the best one (i.e., that with the smallest cost function).

- The $N_{ELITISM}$ best individuals (i.e., $I(1), I(2), \ldots, I(N_{ELITISM})$) pass unchanged to the following generation.
- The next $N_{PARENTS}$ individuals (i.e., $I(N_{ELITISM} + 1), I(N_{ELITISM} + 2), \ldots, I(N_{ELITISM} + N_{PARENTS})$) go through the *crossover* (see Figure 1b) and mutation processes.
- The remaining individuals of the population (i.e., $I(N_{ELITISM} + N_{PARENTS} + 1), \ldots, I(N_{POPULATION})$) are discarded.

The new generation is composed of the $N_{ELITISM}$ best individuals of the previous generation, the $N_{PARENTS}$ individuals obtained from the crossover and mutation processes, and $N_{POPULATION} - N_{ELITISM} - N_{PARENTS}$ new members, which are randomly selected from the search space. The individuals of this new generation are evaluated using the cost function, sorted, etc. The algorithm steps are repeated until the stop condition is reached (see Figure 1a).

The GA supported by *GAPI*Lib includes several processes intended to improve the algorithm performance, such as *elitism* and *mutation*.

- *Elitism* ensures that the most valid individuals pass on to the next generation without being altered by genetic operators. It guarantees that the best solution is never lost from one generation to the next.
- *Mutation* introduces random changes on the individuals, maintaining genetic diversity from one generation of the population of chromosomes to the next. The purpose of mutation is to allow the algorithm to avoid local minima.

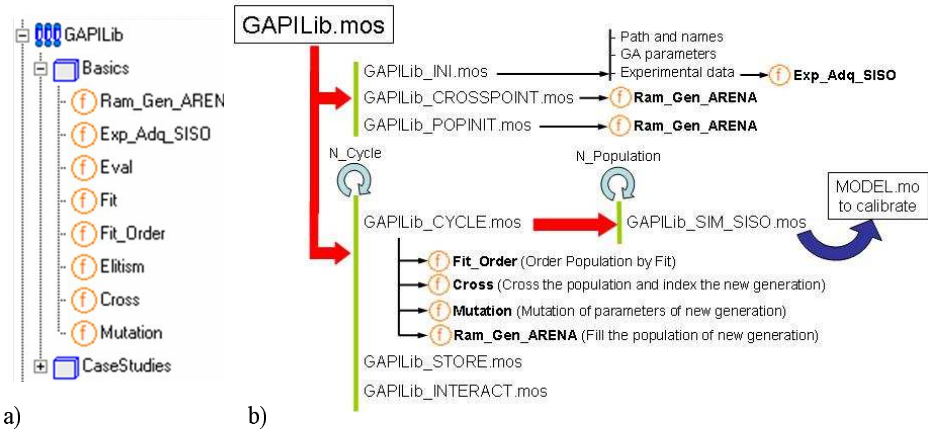

Fig. 2. *GAPI*Lib library: a) Functions; b) Script files with the function calls signaled.

3 GAPILIB Architecture

*GAPI*Lib has been implemented by combining the use of the scripting Modelica language and the use of functions written in the Modelica language. The functions, which are stored within the *GAPILib.Basics* package, are listed in Figure 2a. A detailed description of these functions can be found in [3].

*GAPI*Lib contains a set of script files, written in scripting Modelica language (*.mos* files in Figure 2b), that implement the GA. The GA execution is started by running the script file *GAPILib.mos* (see Figure 2b). This file contains the sentences required to execute the script files that set the GA initial conditions:

- *GAPILib_INI.mos* carries out the initialization of the GA parameters, including the number of individuals of the population ($N_{POPULATION}$), elitist individuals ($N_{ELITISM}$), parents ($N_{PARENTS}$) and cross points (see Figure 1b). Also, the mutation probability, the stop condition of the GA, the path of the file containing the experimental data, the Modelica model, the start and stop times for the Modelica model simulation, etc. are set in this script file.
- *GAPILib_POPINIT.mos* randomly generates the initial population.
- *GAPILib_CROSSPOINT.mos* randomly sets the initial value of the cross point, which is used in the crossover process (see Figure 1b).

The *Ram_Gen_ARENA* function, which is a pseudo-random number generator, is called by *POPINIT* and *CROSSPOINT* script files.

Next, *GAPILib.mos* executes a loop until the GA stop condition is satisfied. The stop condition shown in Figure 2b is of the type: "N_Cycle generations have been obtained". Other stop conditions are possible, e.g., "the calculated fitness

value is smaller than a given value". The loop statements launch the execution of the following script files (see Figure 2b):

- *GAPILib_CYCLE.mos* performs the operations required to obtain the next generation.
- *GAPILib_STORE.mos* logs the results to a file. The results, which are stored in the Matlab format, can be accessed by the user during the GA run.
- *GAPILib_INTERACT.mos* allows the user to change interactively (i.e., during the GA run) the GA parameters.

The script file *GAPILib_CYCLE.mos* contains the required function calls to perform the following tasks (see Figure 2b):

1. To execute the script file *GAPILib_SIM_SISO*, that performs the simulation of the Modelica model, with the parameter values corresponding to each individual of the population. The model is simulated as many times as individuals are in the population. The simulation results are stored and compared with the experimental data. The *Eval* and *Fit* functions are used. All the population individuals are evaluated.
2. To sort the population individuals according to the fitness values previously calculated. The *Fit_Order* function is used.
3. To pass on the elitist individuals to the next generation. These individuals are not altered by crossover and mutation.
4. To apply the crossover process to the parents, using the cross point calculated from *GAPILib_CROSSPOINT*. The *Cross* function is used. The algorithm implemented is shown in Figure 1b.
5. To apply the *Mutation* function. The mutation factor is the probability used to mutate any chromosome of an individual.
6. The new population is completed with random elements. *Ram_Gen_ARENA* function is called.

4 GAPILIB Use

Three practical aspects of *GAPI*Lib use are discussed in this section: (1) the set up of the model calibration; (2) the runtime monitoring of the algorithm convergence and the interactive change of the GA parameters; and (3) the analysis of the parameter sensitivity. Finally, a new capability that will soon be available is described: support for parallel computing of the GA.

4.1 Set up of the Model Calibration Study

The initial conditions of the model calibration study need to be provided by the user. They are defined by giving values to the parameters of the *GAPILib_INI* script file. This set up information includes:

- The name of the Modelica model and the parameters to fit.

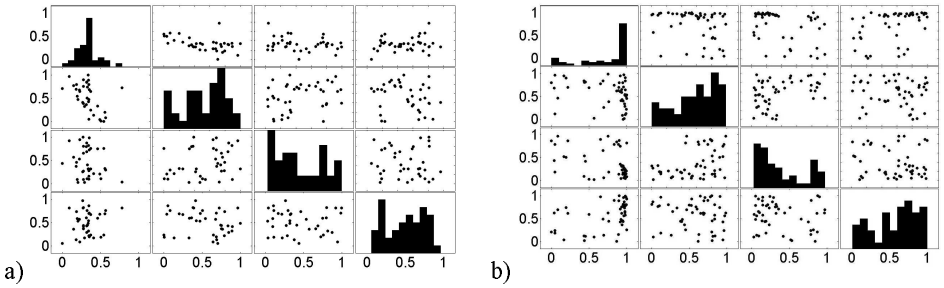

Fig. 3. Parameter sensitivity: a) Calibration of the model described by Eq. (1) (normalized a, b, c and d parameters of the 50-th generation); b) Calibration of a fuel cell model (normalized parameters of the 20-th generation).

- The name and path of the input data file (i.e., the file containing the experimental data) and the output file.
- The GA parameters, including the parent number (N_{PARENT}), the mutation factor, the number of elitist individuals ($N_{ELITISM}$), the number of cross points for the crossover process, the search space and the stop condition of the algorithm.

4.2 Runtime Monitoring of the Algorithm Convergence

*GAPI*Lib supports runtime monitoring of the algorithm. The data of the population chromosomes, the cost function of the individuals, etc. is saved to a file during the algorithm run. This information allows the user to monitor the algorithm convergence and to decide whether he has to interactively modify the GA parameters. The GA parameters can be modified during the algorithm run.

4.3 Parameter Sensitivity

*GAPI*Lib assists in the analysis of the parameter sensitivity. It provides a Matlab function (i.e., *GAPILib_SENSI.m*) that helps to estimate the relative sensitivity of the fitted parameters. This estimation is made considering the dispersion in the chromosome value of the population members with respect to the chromosome value of the best individual. The greater the dispersion, the smaller the parameter sensitivity.

An example of parameter sensitivity analysis is shown in Figure 3a. The 50-th generation of the model described by Eq. (1) is analyzed by plotting the normalized value of the a, b, c and d parameters. The diagonal plots show the relative frequency histograms of a, b, c and d parameters. As the a-parameter histogram exhibits the smaller dispersion, this is the most sensitive parameter.

Another example is the fitness of the fuel cell voltage in response to step changes in the load. The details of this model calibration will be discussed in

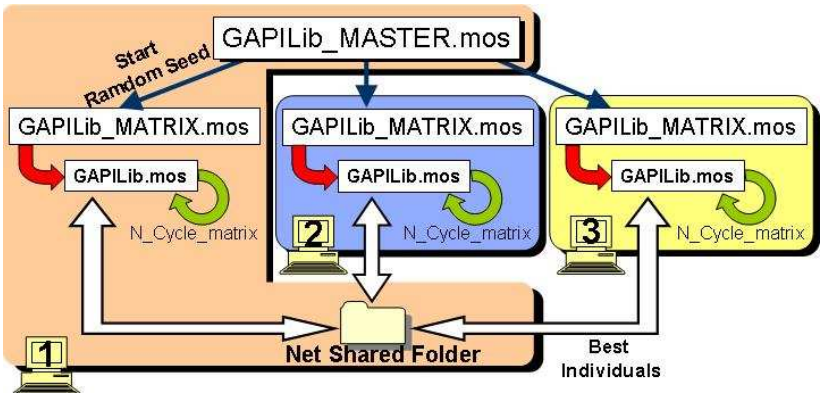

Fig. 4. Parallel computing of the GA using *GAPI*Lib.

Section 5. The goal is to fit the four parameters shown in Table 2. The relative frequency histograms of the 20-th generation chromosomes are plotted in Figure 3b. The first and third parameters (i.e., R_{inf} and C_{dl}) exhibit less dispersion than the other two parameters (i.e., R_{sup} and k_s). As a consequence, R_{inf} and C_{dl} are more sensitive than R_{sup} and k_s.

4.4 Parallel Computing of the GA

Parallel computing allows reducing the time required to complete the model calibration study. Next version of *GAPI*Lib will support the parallelization of the GA. The architecture is shown in Figure 4. *GAPI*Lib needs to be installed in all the computers, which have to be connected (e.g., using TCP/IP).

Initially, the user starts the *GAPILib_MASTER* script file in the master computer. This script file sends random seeds to the other computers, in order to guarantee that the sequences of pseudo random numbers used in the different computers are independent.

Then, the GA is run independently in each computer during certain number of generations (*N_Cycle_matrix*). Periodically, the chromosomes of the best individuals obtained in each computer are saved in a shared folder (see Figure 4) and they are included in the next generation of all the computers.

5 Case Study: Calibration of Fuel Cell Models using *GAPI*Lib

*GAPI*Lib has been successfully applied to the estimation of electrochemical parameters in fuel cell models composed by using *FuelCell*Lib Modelica library [4]. The obtained models can be used to simulate the steady-state and the dynamic

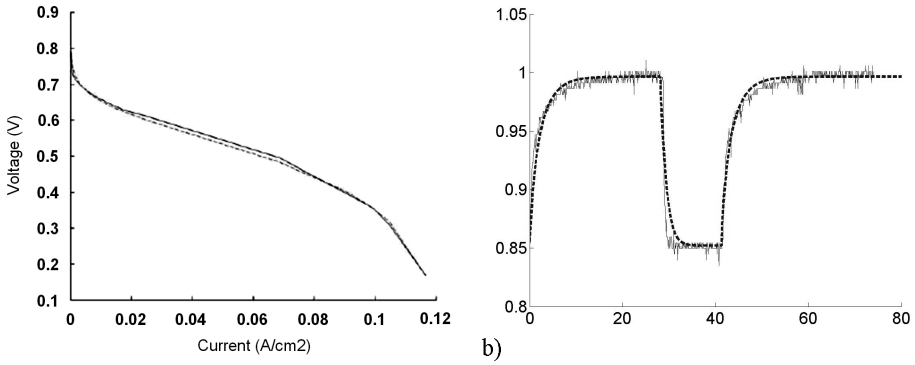

Fig. 5. Experimental (–), simulated using *FuelCell*Lib (- -): a) Fuel cell polarization curve; b) Fuel cell voltage in response to step changes in the load, Voltage [V] vs. Time [s].

behavior of the fuel cells along their complete range of operation [8–10]. Three model calibrations have been performed, in order to fit the model to:

1. The experimental polarization curve (I-V) of a fuel cell.
2. The experimental data of the fuel cell voltage obtained in response to step changes in the load.
3. The experimental data of the water long-term effect. The fitted model reproduces:
   (a) The slow voltage rise due to the membrane hydrate.
   (b) The voltage fall due to the water flooding of the cathode.

5.1 Fitness of the Polarization Curve

In order to obtain the polarization curve, the model has to be simulated, for each of the operation points composing the curve, until the steady-state is reached. The parameters estimated and the obtained values are shown in Table 1. One cross point was used. The fitness function was defined as the sum of the quadratic differences between the experimental and the simulated values of the variable.

The GA parameters were set to the following values: the stop condition is satisfied after 5000 generations; the population was composed of 100 individuals; the mutation factor was 0.25; $N_{PARENT} = 70$; and $N_{ELITISM} = 1$. The experimental data of the fuel cell polarization curve and the simulation results of the calibrated model are shown in Figure 5a.

Table 1. Model parameters and their fitted values.

Parameter		Value	Unit
A	Tafel Slope	0.0390	V
I_n	Internal current density	$1.4 \cdot 10^{-3}$	$A \cdot cm^{-2}$
I_0	Exchange current density	$1.5856 \cdot 10^{-6}$	$A \cdot cm^{-2}$
B	Mass Transfer slope	0.0918	V
R	Internal specific resistance	$7.2860 \cdot 10^{-4}$	$\Omega \cdot cm^{-2}$
I_{lim}	Limiting internal current density	0.2265	$A \cdot cm^{-2}$

Table 2. Model parameters and their fitted values.

Parameter		Value	Unit
R_{inf}	Low value of the load	0.03315	$\Omega \cdot m^{-2}$
R_{sup}	High value of the load	5.1	$\Omega \cdot m^{-2}$
C_{dl}	Double layer capacitance	10.12	$F \cdot m^{-2}$
k_s	Electrical conductivity of the solid	0.01	$S \cdot m^{-1}$

5.2 Fitness of the Fuel Cell Voltage in Response to Step Changes in the Load

*GAPI*Lib is used to estimate the values of the four parameters shown in Table 2. The GA parameters are set to the following values: the stop condition is satisfied after 200 generations; the population contains 150 individuals; a factor of mutation of 0.25 was applied; $N_{PARENT} = 100$; and $N_{ELITISM} = 1$. The experimental data of the fuel cell response to step changes in the load and the simulation results of the calibrated model are shown in Figure 5b.

5.3 Fitness of the Long Term Effect of Water on the Fuel Cell Voltage with Constant Resistance Load

The fuel cell model was modified in order to reproduce the variation of the membrane conductivity. The parameters used to fit the model are shown in Table 3. The GA parameters were set to the following values: the stop condition was satisfied after 700 generations; the population was composed of 70 individuals; a factor of mutation of 0.15 was applied; $N_{PARENT} = 50$; and $N_{ELITISM} = 1$. The experimental data of the cathode flooding process and the simulation results of the calibrated model are shown in Figure 6.

Table 3. Model parameters and their fitted values.

Parameter		Value	Unit
$d_{a(Act)}$	Width of active layer	$6 \cdot 10^{-8}$	m
ϵ_g	Volume fraction of pore	0.05	
D_{12}	Binary diffusion coefficient	$5 \cdot 10^{-9}$	$m^2 \cdot s^{-1}$
$d_{a(Mem)}$	Width of membrane layer	$1.6 \cdot 10^{-5}$	m
R_{mem}	Resistance of membrane layer	$1.42 \cdot 10^{-3}$	$\Omega \cdot m^{-2}$

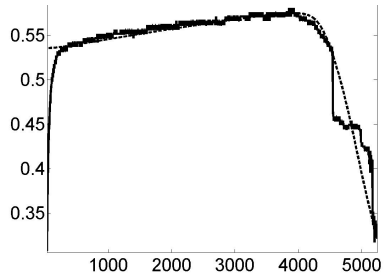

Fig. 6. Experimental (–), simulated using *FuelCell*Lib (- -): long-term effect of the water with a constant load applied, Voltage [V] vs. Time [s].

6 Conclusions

The design, implementation and use of *GAPI*Lib has been discussed. The *GAPI*Lib library is an effective tool for parameter identification in Modelica models using GA. It is completely written in the Modelica language, which facilitates its use, modification and extension. *GAPI*Lib can be used for parameter identification in any Modelica model and the estimation process does not require to perform model modifications. *GAPI*Lib has been successfully applied to the estimation of electrochemical parameters in fuel cell models, which have been composed by using *FuelCell*Lib library.

Acknowledgements

This work has been supported by the Spanish CICYT, under DPI2004-01804 grant, and by the IV PRICIT (Plan Regional de Ciencia y Tecnologia de la Comunidad de Madrid, 2005-2008), under S-0505/DPI/0391 grant.

The fuel cell experimental data used in this work has been obtained in the Laboratory of Renewable Energy of the IAI-CSIC in Madrid (Spain).

References

1. Stuckman, B., Evans, G., Mollaghasemi, M.: Comparison of Global Search Methods for Design Optimization Using Simulation. In: Proceedings of 1991 Winter Simulation Conference, 1991, pp. 937–944.
2. Hongesombut, K., Mitani, Y., Tsuji, K.: An Incorporated Use of Genetic Algorithm and a Modelica Library for Simultaneous Tuning of Power System Stabilizers. In: Proceedings of the 2^{nd} International Modelica Conference, 2002, pp. 89–98.
3. Rubio, M.A., Urquia, A., Gonzalez, L., Guinea, D., Dormido, S.: GAPILib - A Modelica Library for Model Parameter Identification Using Genetic Algorithms, In: Proceedings of 5th International Modelica Conference, 2006, pp. 335–342.
4. Rubio, M.A., Urquia, A., Gonzalez, L., Guinea, D., Dormido, S.: FuelCellLib - A Modelica Library for Modeling of Fuel Cells. In: Proceedings of the 4^{th} International Modelica Conference, 2005, pp. 75–82.
5. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning, Kluwer Academic Publishers, Boston, MA, 1989.
6. Holland, J.H.:Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, 1975.
7. Mitchell, M.: An Introduction to Genetic Algorithms, MIT Press, Cambridge, MA., 1996.
8. Larminie, J., Dicks, A.: Fuel Cell Systems Explained, Wiley, 2000.
9. Bevers, D., Wohr, M., Yasuda, K., Oguro, K.: Simulation of Polymer Electrolyte Fuel Cell Electrode. J. Appl. Electrochem, **27** (1997).
10. Broka, K., Ekdunge, P.: Modelling the PEM Fuel Cell Cathode, J. Appl. Electrochem. **27** (1997).

Dynamic Optimization of Modelica Models – Language Extensions and Tools

Johan Åkesson

Department of Automatic Control
Faculty of Engineering
Lund University
Sweden
jakesson@control.lth.se

Abstract. The Modelica language is currently gaining increased interest, both in industry and in academia. Modelica is an object-oriented, general purpose modeling language, targeted at modeling of complex physical systems. While the main usage of models developed in Modelica is simulation, several other usages emerge. Examples of such usages are dynamic optimization, model reduction, calibration, verification and code generation for embedded systems. This paper reports the current status of the JModelica project, in which an extensible, Java-based Modelica compiler is being developed. In addition, an extension of the Modelica language directed towards dynamic optimization, Optimica, is discussed.

1 Introduction

High-level modeling languages are receiving increased industrial and academic interest within several domains, such as chemical engineering, thermo-fluid systems and automotive systems. One such modeling language is Modelica, [8]. Modelica is an open language, specifically targeted at multi-domain modeling and model re-use. Key features of Modelica include object oriented modeling, declarative equation-based modeling, and a component model enabling acausal connections of submodels, as well as support for hybrid/discrete behaviour. These features have proven very applicable to large-scale modeling problems in various fields.

While there exist very efficient software tools for simulation of Modelica models, tool support for static and dynamic optimization is generally weak. Furthermore, specification of optimization problems is not supported by Modelica. Since Modelica models represent an increasingly important asset for many companies, it is of interest to investigate how Modelica models can be used also for optimization.

This contribution gives an overview of a project, entitled JModelica, targeted at *i)* defining an extension of Modelica, Optimica, which enables high-level formulation of optimization problems, *ii)* developing prototype tools for translating a Modelica model and a complementary Optimica description into a

representation suited for numerical algorithms, and *iii)* performing case studies demonstrating the potential of the concept.

The project integrates dynamic modeling and optimization with computer science and numerical algorithms. One of the main benefits of the suggested approach is that the high-level descriptions are automatically translated into an intermediate representation by the compiler front-end. This intermediate representation can then be further translated to interface with different numerical algorithms. The user is therefore relieved from the burden of managing the often cumbersome API:s of numerical algorithms. The flexibility of the architecture also enables the user to select the algorithm most suitable for the problem at hand.

2 Software Tools

In order to demonstrate the proposed concept, prototype software tools are being developed. In essence, the task of the software is to read the Modelica and Optimica source code and then translate, automatically, the model and optimization descriptions into a format which can be used by a numerical algorithm. The core of the software is a compiler front-end, referred to as the JModelica compiler, which translates a subset of Modelica into a flat model description. In addition, an extended front-end, based on the JModelica compiler, supporting a first prototype of the Optimica extension has been developed. The extended compiler is referred to as the Optimica compiler. In addition, a back-end for generation of efficient code for dynamic optimization has been developed.

2.1 Development Environment

The JModelica compiler is developed using the Java-based compiler construction tool JastAdd, [7]. JastAdd is a development environment targeted at implementation of the semantics of computer programming languages, and has also been explicitly designed with modular and extensible compiler construction in mind. The core concepts used in JastAdd are object orientation, static aspect orientation, and reference attributed grammars [6].

The JastAdd system is based on an object oriented specification of an abstract grammar (AG), from which standard Java classes are generated. Semantic behaviour is added in *aspects*, which are useful for organizing cross-cutting behaviour. It is natural to structure the implementation of different semantic functions, such as name analysis (the task of binding identifiers to declarations) and type analysis (e.g. computation of the types of expressions), into separate modules. However, since the implementation of, for example, name analysis, typically affects a large number of classes, the object-oriented paradigm does not inherently offer support for this kind of modularization. In JastAdd, this problem is overcome by allowing definition of behaviour, in the form of inter-type declarations, in separate aspects, which are then *woven* into the AG classes. The

resulting classes contain only Java code, and can be compiled by a standard Java compiler.

The choice of JastAdd is natural in this project, since its main focus is extensions of the Modelica language. In particular, the methodology adopted by JastAdd enables the implementations of the core language compiler and the extensions to be separated. It is then possible to build the core compiler alone, or with one or more extensions. As a notable example, a full Java 1.4 compiler, and a fully modular extension to also support Java 1.5 have been implemented in JastAdd, [3]. For an overview of the JModelica compiler implementation, including some performance benchmarks, see [1].

2.2 Code Generation to AMPL

Currently, the front-end of the JModelica/Optimica compiler supports a subset of Modelica and a basic version of Optimica. In addition, a code-generation back-end for AMPL, [4], has been developed. AMPL is a language intended for formulation of algebraic optimization problems. Accordingly, the compiler performs automatic transcription of the original continuous-time problem into an algebraic formulation which can be encoded in AMPL. In the transcription procedure, the problem is discretized by means of a simultaneous optimization approach based on collocation over finite elements, see for example., [2] for an overview. Finally, the automatically generated AMPL description may be executed and solved by a numerical NLP algorithm. For this purpose we have used IPOPT, [9].

2.3 Project Status

This paper describes the current status of the JModelica project, as of June 2007. Currently, the JModelica compiler supports a limited subset of Modelica, which includes classes, components, inheritance, value modifications, connect-clauses and partial support for arrays. The functionality of the Optimica compiler will be described in detail in the next section.

3 Optimica

A key issue is the definition of syntax and semantics of the Modelica extension, Optimica. Optimica should provide the user with language constructs that enable formulation of a wide range of optimization problems, such as parameter estimation, optimal control and state estimation based on Modelica models.

At the core of Optimica are the basic optimization elements such as cost functions and constraints. It is also possible to specify bounds on variables in the Modelica model as well as marking variables and parameters as optimization quantities, i.e., to express what to optimize over. While this type of information represents a canonical optimization formulation, the user is often required to supply additional information, related to the numerical method which is used to

solve the problem. In this category we have e.g., specification of transcription method, discretization of control variables and initial guesses. Optimica should also enable convenient specification of these quantities.

The current preliminary specification of the Optimica language admits formulation of dynamic optimization problems on the following form:

$$\begin{aligned} &\min_{u(t),p} \int_0^{t_f} L(x(t),u(t),p)dt + \phi(x(t_f)) \\ &\text{subject to} \\ &f(\dot{x},x,u,p) = 0 \\ &c_i(x(t),u(t),p) \leq 0, \quad c_e(x(t),u(t),p) = 0 \\ &c_{fe}(x(t_f),u(t_f),p) = 0, \quad c_{fi}(x(t_f),u(t_f),p) \leq 0 \\ &c_{0e}(x(0),u(0),p) = 0, \quad c_{0i}(x(0),u(0),p) \leq 0 \end{aligned} \tag{1}$$

The dynamic constraint $f(\dot{x},x,u,p) = 0$ is expressed using Modelica, and Optimica is used for everything else.

3.1 The Optimica Extension

The anatomy of an Optimica description of an optimization problem is similar to a simple Modelica model, and consists of three sections. In the first section, information relevant for formulation of the optimization problem may be superimposed on elements in the Modelica model. For example, variable bounds and initial guesses can be specified. In addition, it is possible to mark Modelica parameters and initial conditions of dynamic variables as free optimization variables. In the second section, referred to as `optimization`, the cost function and the optimization horizon can be specified. In the third section, referred to as `subject to` the constraints of the problem is given.

In the current version of Optimica, the content of a Modelica class is implicitly assumed to be present in the scope of an Optimica class. This is equivalent to the Optimica class extending from the corresponding Modelica class. In future versions of Optimica, this implicit assumption will be removed in favor of allowing explicit extends statements as well as component declarations in the Optimica description.

In essence, Optimica supports four constructs:

– **Superimpose information on Modelica variables.** Commonly, it is desirable to superimpose optimization-related information on variable declarations in the Modelica model. For this purpose, a new construct is introduced:

```
[oq] component_access [modification]
```

where the name `component_access` binds to a name in the corresponding Modelica model. In addition, the optional prefix `oq`, see below, and a modification construct can be specified. Notice that this is not a component declaration, but should be seen as a mechanism for adding information to

an existing declaration; modifications given in this construct are merged with those of the original declaration. In Modelica, this construct corresponds to a redeclare modification, which may change the prefix of a variable as well as add modifications. This new construct can therefore be viewed as a simplified and shorthand alias for a redeclare modification. The introduction of a new language construct is motivated by the need for a compact and efficient way to superimpose information on variables, without having to use the more involved component redeclaration mechanism. In addition, the current version of Optimica does not support component declarations, which makes the proposed construct convenient.
Bounds on variables, both inputs and states, and parameters can be expressed using the construct

```
[oq] varName(lowerBound=-1,upperBound=1);
```

where `varName` refers to a variable or parameter in the Modelica model. The optional prefix `oq` (Optimization Quantity) is used to let a Modelica parameter or variable be free in the optimization. The effect of using the `oq` prefix for a variable is that the binding expression, if any, of the corresponding declaration is removed.
It is also possible to specify an initial guess for a variable or parameter in Optimica:

```
varName(lowerBound=-1,upperBound=1,initialGuess=0);
```

The initial guess is a constant expression, which is used to initialize variables and optimization parameters. If an initial guess file is supplied upon compilation, the initial guess in the Optimica description has priority over the one in the file. Also notice that the initial guess has no effect for a Modelica parameter if the `oq` prefix is not specified.
It is also possible to specify bounds and initial guess for derivatives of variables:

```
der(varName)(lowerBound=-1,upperBound=1,initalGuess=0.3);
```

Dynamic variables by default have fixed initial conditions, specified by the start-attribute given in the corresponding Modelica variable declaration. The following construct enables free initial conditions:

```
varName(freeInitial(lowerBound=[-0.01;-0.001;-0.01;-0.001],
                            upperBound=[0.01;0.001;0.01;0.001],
                            initialGuess=[0.001;0;0;0])=true);
```

where there the variable `varName` in this case is an array variable. Notice that upper and lower bound as well as initial guess (optional) for the variable can be given in the same construct:

```
varName(lowerBound=-3,upperBound=3,initialGuess=1,
     freeInitial(lowerBound=-2,upperBound=2,initialGuess=0)=true);
```

– **Specification of grid.** The solution of the optimization problem is defined on a grid, consisting of a number of time points. The accuracy (and usually execution time) is increased if a grid with more points is used. Due to the nature of the transcription scheme used in the Optimica compiler, it is more natural to specify the number of *elements* of the grid. The number of points is then given by three times the number of elements, since a third order collocation method is used. A grid with fixed final time is specified by the construct

```
grid(finalTime=fixedFinalTime(finalTime=tf),nbrElements=n_el);
```

and a grid with free final time is specified by

```
grid(finalTime = openFinalTime(initialGuess=tf_ig,lowerBound=tf_lb,
                        upperBound=tf_ub),nbrElements=n_el);
```

By specifying a free final time, it is possible to formulate minimum time problems.
A static optimization problem is defined by using the construct:

```
grid(static=true);
```

In this case, all `der`-operators in the model are replaced by zero.
Notice that the `grid` construct must reside in an `optimization` section.

– **Definition of cost function.** The cost function is specified in the `optimization` section using the construct

```
minimize(lagrangeIntegrand=li_exp,terminalCost=tc_exp);
```

The argument `lagrangeIntegrand` corresponds to the integrand expression in the Lagrange cost function, L and `terminalCost` corresponds to ϕ.

– **Specification of constraints** In the `subject to` section, path, initial and terminal constraints can be specified. A terminal constraint is introduced using the prefix `terminal` and an initial constraint is introduced by the prefix `initial`. Examples of constraints are

```
y<=x^2; // Path constraint
initial cos(x)>=0.4 // Initial constraint
terminal y=4; // Terminal constraint
```

3.2 2D Double Integrator Example

Consider the following model of a two dimensional double integrator:

$$\begin{aligned} \ddot{x}(t) &= u_x(t) \\ \ddot{y}(t) &= u_y(t) \end{aligned} \tag{2}$$

We would like to find trajectories that transfer the state of the system from $(-1.5, 0)$ to $(1.5, 0)$ in shortest possible time. In addition, we would like to impose

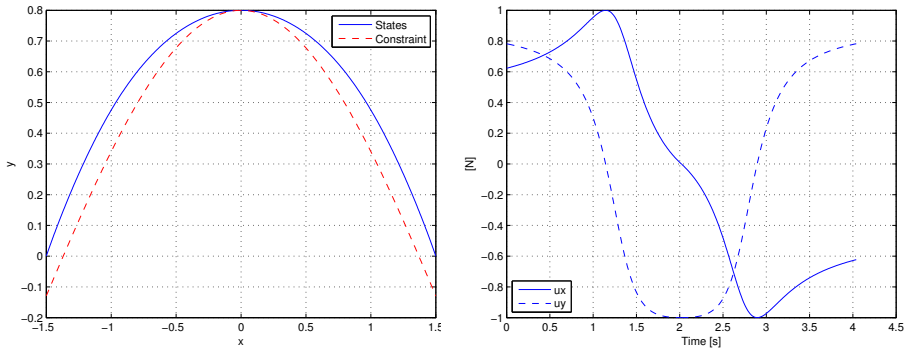

Fig. 1. Resulting optimization profiles for the minimum time case.

the path constraint $y \geq \cos x - 0.2$ and $u_x^2 + u_y^2 \leq 1$. The latter constraint ensures that the resulting force has a magnitude equal to or less than 1. This gives us the following optimal control formulation

$$
\begin{aligned}
&\min_{u} \int_0^{t_f} 1 dt \\
&\text{subject to} \\
&\quad \ddot{x}(t) = u_x(t) \\
&\quad \ddot{y}(t) = u_y(t) \\
&\quad x(0) = -1.5, \quad x(t_f) = 1.5, \quad y(0) = 0, \quad y(t_f) = 0 \\
&\quad \dot{x}(0) = 0, \quad \dot{x}(t_f) = 0, \quad \dot{y}(0) = 0, \quad \dot{y}(t_f) = 0 \\
&\quad y(t) \geq \cos x(t) - 0.2 \\
&\quad 1 \geq u_x(t)^2 + u_y(t)^2
\end{aligned}
\tag{3}
$$

The dynamics of the double integrator system is given by the following Modelica model:

```
model DoubleIntegrator2d
  input Real ux;
  input Real uy;
  Real x(start=-1.5), vx(start=0);
  Real y(start=0),    vy(start=0);
equation
  der(x)=vx; der(vx)=ux;
  der(y)=vy; der(vy)=uy;
end DoubleIntegrator2d;
```

and the Optimica description of the optimization problem is given by:

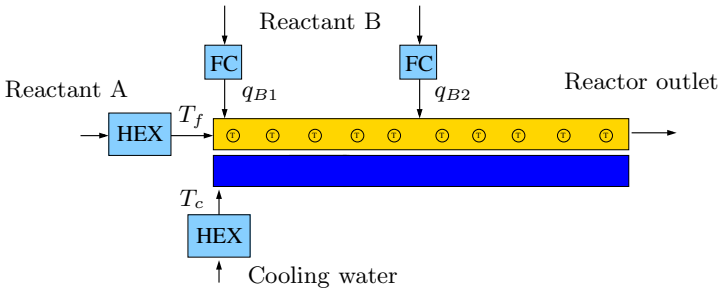

Fig. 2. The reactor shown as a schematic tubular reactor. There are four inflows to the process and there is one manipulated variable for each inflow; q_{B1}, q_{B2}, T_f and T_c. Each inflow has an actuator subsystem that provides flow control (FC) or temperature control through heat exchangers (HEX). The circles with T represents internal temperature sensors.

```
class optDI2d
optimization
  grid(finalTime = openFinalTime(initialGuess=4.5,lowerBound=3,
                                 upperBound=tf_ub),nbrElements=5);
  minimize(lagrangeIntegrand=1);
subject to
  terminal x=1.5; terminal vx=0;
  terminal y=0;   terminal vy=0;
  ux^2+uy^2<=1;   y>=cos(x)-0.2;
end optDI2d;
```

Notice that the initial conditions are expressed in the Modelica model using the `start` attribute, whereas the terminal constraints are given in the `subject to` clause in the Optimica model. The resulting time optimal trajectories are shown in Figure 1.

4 A Case Study

The Optimica compiler has been used to formulate and solve a start-up problem for a plate reactor system. The plate reactor is conceptually a tubular reactor located inside a heat exchanger, and offers excellent flexibility, since it is reconfigurable and allows multiple injection points for chemicals, separate cooling/heating zones and easy mounting of temperature sensors. In this case study, an exothermic reaction, $A + B \rightarrow C$, was assumed. The reactor was fed with a fluid with a specified concentration of the reactant A. The reactant B was injected at two points along the reactor. The control variables of the system were the temperatures of the inlet flow, the temperature of the cooling flow and the injection flow-rates of the reactant B, see Figure 2.

The primary objective of the start-up sequence was to transfer the state of the reactor from an operating point where no reaction takes place, to the desired

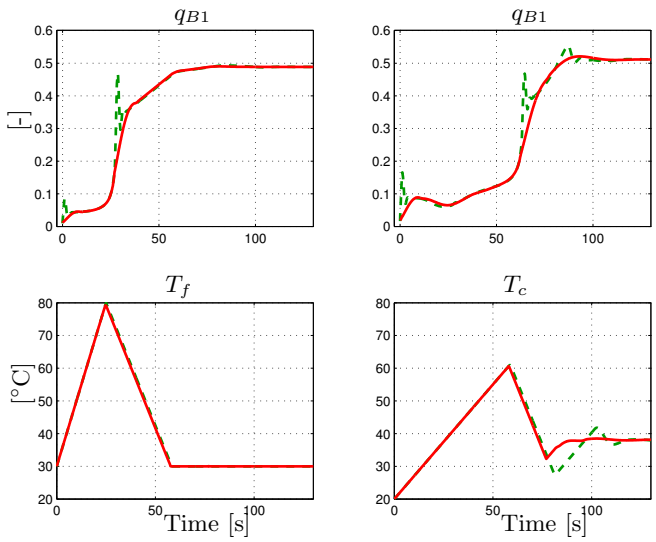

Fig. 3. Optimal control profiles. The dashed curves correspond to a case with a small high frequency penalty on the inputs, whereas the solid curves represents a case with a larger high frequency penalty, resulting in smoother control profiles.

point of operation. This problem is challenging, since the dynamics of the system is fast and unstable in some operating conditions. Also, the temperature in the reactor must be kept below a safety limit, in order not to damage the hardware.

A Modelica model, containing 131 states and 71 algebraic variables, was used to represent the dynamics of the system. Optimal control and state profiles were calculated off-line and then used as feedforward and feedback signals in a PID-based mid-ranging control system. The resulting optimization problem contained approximately 160,000 variables. The optimal control profiles, q_{B1}, q_{B2}, T_f and T_c are shown in Figure 3, and the corresponding output temperature and concentration profiles, T_1, T_2, $c_{B,1}$ and $c_{B,2}$ are shown in Figure 4.

The experiences from using the Optimica compiler in this project are promising, in that the tools enable the user to focus on *formulation* of the problem instead of, which is common, *encoding* of the problem. For more details on this case study, see [5].

5 Summary

This contribution gives an overview of the JModelica project, which is targeted at extending the Modelica language to also support optimization. The goals of the project include specification of the language extension Optimica, development of prototype software tools and case studies. A preliminary specification of Optimica, offering basic support for formulation of dynamic optimization problems based on Modelica models has been presented.

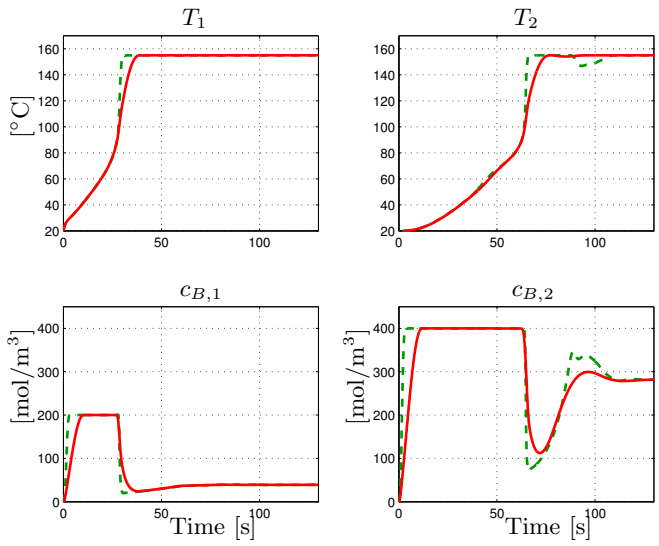

Fig. 4. Optimal profiles profiles for reactor temperature and concentration of substance B. The left plots correspond to the first injection point, whereas the right plots correspond to the second injection point.

References

1. Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Development of a Modelica compiler using JastAdd. In *Seventh Workshop on Language Descriptions, Tools and Applications*, Braga, Portugal, March 2007.
2. L.T. Biegler, A.M. Cervantes, and A Wchter. Advances in simultaneous strategies for dynamic optimization. *Chemical Engineering Science*, 57:575–593, 2002.
3. T. Ekman and G Hedin. The jastadd extensible java compiler. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007*, Montreal, Canada, October 2007. To appear.
4. R. Fourer, D. Gay, and B. Kernighan. *AMPL – A Modeling Language for Mathematical Programming*. Brooks/Cole — Thomson Learning, 2003.
5. Staffan Haugwitz, Johan Åkesson, and Per Hagander. Dynamic optimization of a plate reactor start-up supported by Modelica-based code generation software. In *Proceedings of 8th International Symposium on Dynamics and Control of Process Systems*, Cancun, Mexico, June 2007.
6. G. Hedin. Reference Attributed Grammars. In *Informatica (Slovenia)*, 24(3), pages 301–317, 2000.
7. G. Hedin and E. Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.
8. The Modelica Association, 2006. http://www.modelica.org.
9. Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–58, 2006.

Robust Initialization of Differential Algebraic Equations

Bernhard Bachmann, Peter Aronsson*, Peter Fritzson+

Dept. Mathematics and Engineering, University of Applied Sciences,
D-33609 Bielefeld, Germany
bernhard.bachmann@fh-bielefeld.de

* MathCore Engineering AB, Teknikringen 1F, SE-583 30 Linköping, Sweden
peter.aronsson@mathcore.com

+ PELAB Programming Environments Lab, Department of Computer Science
Linköping University, SE-581 83 Linköping, Sweden
petfr@ida.liu.se

(Previously published in Modelica'2006, Vienna, Sept 4-5, 2006. www.modelica.org)

Abstract. This paper describes a new solution method applied to the problem initializing DAEs using the Modelica lan-guage. Modelica is primarily an object-oriented equa-tion-based modeling language that allows specification of mathematical models of complex natural or man-made systems. Major features of Modelica are the mul-tidomain modeling capability and the reusability of model components corresponding to physical objects, which allow to build and simulate highly complex sys-tems. However, initializing such models has been quite cumbersome, since initial equations have to be pro-vided at the system level, where the user needs to know details on the underlying transformation and index-reduction algorithms, that in general are applied to simulate a Modelica model. .

1 Introduction

So far, using model initialization in Modelica has only been possible for higher-index problems if the user formulates the initial equations globally. This was also the case, e.g. when using the OpenModelica OpenModelica compiler which is an open source implementation developed at PELAB, Linköping University. In order to do such a global formulation successfully, the user needs to know about index reduction, at least the number of freedom left after applying the dummy derivative method is necessary. Therefore, only advanced users have been able to use this feature in the Modelica language, when higher index problems occur (which is very common). In order to provide a more complete simulation environment, we have started to add robust initialization techniques to the OpenModelica compiler.

2 Flattening of a Modelica Model to a Hybrid DAE

A Modelica model is typically translated to a basic mathematical representation in terms of a flat system of *differential and algebraic equations* (DAEs) before being able to simulate the model. This translation process elaborates on the internal model representation by performing analysis and type checking, inheritance and expansion of base classes, modifications and redeclarations, conversion of connect-equations to basic equations, etc. The result of this analysis and translation process is a flat set of equations, including conditional equations, as well as constants, variables, and function definitions. By the term *flat* is meant that the object-oriented structure has been broken down to a flat representation where no trace of the object hierarchy remains apart from dot notation (e.g. `Class.Subclass.variable`) within names.

3 Mathematical Formulation of Hybrid DAEs

3.1 Summary of notation

Below we summarize the notation used in the equations that follow, with time dependencies stated explicitly for all time-dependent variables by the arguments t or t_e:

- $p = \{p_1, p_2, ...\}$, 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.
- $x(t)$, 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.
- $\dot{x}(t)$, the differentiated vector of state variables of the model.
- $u(t)$, a vector of input variables, i.e., not dependent on other variables, of type `Real`. These also belong to the set of algebraic variables since they do not appear differentiated.
- $y(t)$, a vector of Modelica variables of type `Real` which do not fall into any other category. Output variables are included among these, which together with $u(t)$ are algebraic variables since they do not appear differentiated.
- $q(t_e)$, 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/statements and if-expressions from the original model as well as those generated during the conversion of when-equations and when-statements.

- $rel(v(t)) = rel(cat(1, x, \dot{x}, u, y, \{t\}, q(t_e), q_{pre}(t_e), p))$, a `Boolean` vector valued function containing the relevant elementary relational expressions from the model, excluding relations enclosed by `noEvent()`. The argument $v(t)$ = $\{v_1, v_2, ...\}$ is a vector containing all elements in the vectors $x, \dot{x}, u, y, \{t\}, q(t_e), q_{pre}(t_e), p$. This can be expressed using the Modelica concatenation function `cat` applied to these vectors; $rel(v(t))$ = $\{v_1 > v_2, v_3 >= 0, v_4<5, v_6<=v_7, v_{12}=133\}$ is one possible example.
- $f(...)$, the function that defines the differential equations $f(...) = 0$ in (1a) of the system of equations.
- $g(...)$, the function that defines the algebraic equations $g(...) = 0$ in (1b) of the system of equations.
- $f_q(...)$, the function that defines the difference equations for the discrete variables $q := f_q(...)$, i.e., (2) in the system of equations.
- $f_e(...)$, the function that defines the event conditions $c := f_e(...)$, i.e., (3) in the system of equations.
- $f_x(...)$, the function that defines the reinitialization values for the continuous variables $x(t_e) := f_x(...)$ at events.

In the context of hybrid DAE:s the *state* of a system is not only made up of the values of the set of variables that occur differentiated in the model. The overall *state* of a system may also include values of discrete variables. In this paper the word *state* is used in this sense, including the state of the discrete part of the system.

3.2 Continuous-Time Behavior

Now we want to formulate the continuous part of the *hybrid DAE* system of equations including discrete variables. This is done by adding a vector $q(t_e)$ of *discrete-time variables* and the corresponding predecessor variable vector $q_{pre}(t_e)$ denoted by `pre(q)` in Modelica. For discrete variables we use t_e instead of t to indicate that such variables may only change value at event time points denoted t_e, i.e., the variables $q(t_e)$ and $q_{pre}(t_e)$ behave as constants between events.

We also make the constant vector p of *parameters and constants* explicit in the equations, and make the time t explicit. The vector $c(t_e)$ of condition expressions, e.g. from the conditions of `if` constructs and `when` constructs, evaluated at the most recent event at time t_e is also included since such conditions are referenced in conditional equations. We obtain the following *continuous DAE* system of equations that describe the system behavior *between* events:

$$\begin{aligned} f(x(t), \dot{x}(t), u(t), y(t), t, q(t_e), q_{pre}(t_e), p, c(t_e)) &= 0 \quad (a) \\ g(x(t), u(t), y(t), t, q(t_e), q_{pre}(t_e), p, c(t_e)) &= 0 \quad (b) \end{aligned} \tag{1}$$

3.3 Discrete-Time Behavior

Discrete time behavior is closely related to the notion of an event. Events can occur asynchronously, and affect the system one at time, causing a sequence of state transitions.

An event occurs when any of conditions $c(t_e)$ (defined below) of conditional equations changes value from `false` to `true`. We say that an event becomes *enabled* at the time t_e, if and only if, for any sufficiently small value of ε, $c(t_e\text{-}\varepsilon)$ is `false` and $c(t_e+\varepsilon)$ is `true`. An enabled event is *fired*, i.e., some behavior associated with the event is executed, often causing a discontinuous state transition.

Firing of an event may cause other conditions to switch from `false` to `true`. In fact, events are fired until a stable situation is reached when all the condition expressions are `false`.

However, there are also state changes caused by equations defining the values of the *discrete* variables $q(t_e)$, which may change value *only* at events, with event times denoted t_e. Such discrete variables obtain their value at events, e.g. by solving equations in when-equations or evaluating assignments in when-statements. The instantaneous equations defining discrete variables in when-equations are restricted to particularly simple syntactic forms, e.g. *var = expr ;* . These restrictions are imposed by the Modelica language in order to easily determine which discrete variables are defined by solving the equations in a when-equation.

Such equations can be directly converted to equations in assignment form, i.e., assignment statements, with fixed causality from the right-hand side to the left-hand side. Regarding algorithmic when-statements that define discrete variables, such definitions are always done through assignments. Therefore we can in both cases express the equations defining discrete variables as *assignments* in the vector equation (1a), where the vector-valued *function* f_q specifies the right-hand side expressions of those *assignments to discrete variable*s.

$$q(t_e) := f_q(x(t_e), \dot{x}(t_e), u(t_e), y(t_e), t_e, q_{pre}(t_e), p, c(t_e)) \tag{2}$$

The last argument $c(t_e)$ is made explicit for convenience. It is strictly speaking not necessary since the expressions in $c(t_e)$ could have been incorporated directly into f_q. The vector $c(t_e)$ contains all `Boolean` condition expressions evaluated at the most recent *event* at time t_e. It is defined by the following vector assignment equation with the right-hand side given by the vector-valued function f_e. This function has as arguments the subset of the discrete variables having `Boolean` type, i.e., $q^B(t_e)$ and $q^B_{pre}(t_e)$, the subset of `Boolean` parameters or constants, p^B, and a vector $rel(v(t))$ evaluated at time t_e, containing the elementary relational expressions from the model. The vector of condition expressions $c(t_e)$ is defined by the following equation in assignment form:

$$c(t_e) := f_e(q^B(t_e), q^B_{pre}(t_e), p^B, rel(v(t_e))) \tag{3}$$

The argument $v(t) = \{v_1, v_2, ...\}$ is a vector containing all scalar elements of the argument vectors. This can be expressed using the Modelica concatenation function `cat` applied to the vectors, e.g. $v(t) = cat(1, x, \dot{x}, u, y, \{t\}, q(t_e), q_{pre}(t_e), p)$. For exam-

ple, if $rel(v(t))$ = {$v_1 > v_2$, $v_3 >= 0$, $v_4<5$, $v_6<=v_7$, $v_{12}=133$} where $v(t)$ = {v_1, v_2, v_3, v_4, v_6, v_7, v_{12}}, then it might be the case that $c(t)$ = {$v_1 > v_2$ `and` $v_3 >= 0$, v_{10}, not v_{11}, $v_4<5$ `or` $v_6<=v_7$, $v_{12}=133$}, where v_{10}, v_{11} are `Boolean` variables and v_1, v_2, v_3, v_4, v_6, v_7 might be `Real` variables, whereas v_{12} might be an `Integer` variable. $rel(v(t)) = rel(\texttt{cat}(1, x(t), \dot{x}(t), u(t), y(t), t, q(t_e), q_{pre}(t_e), p))$, is a `Boolean`-typed vector-valued function containing the relevant elementary *relational expressions* from the model, excluding relations enclosed by `noEvent()`.

Discontinuous changes of continuous dynamic variables $x(t)$ can be caused by so-called `reinit` equations in Modelica. As in the case of discrete variables, such discontinuous changes can only occur at events. The effect of a `reinit`-equation that is activated at t_e is an assignment to the continuous variable at time t_e of the form:

$$x(t_e) := f_x(x(t_e), \dot{x}(t_e), u(t_e), y(t_e), t_e, q_{pre}(t_e), p, c(t_e)) \qquad (4)$$

For all variables in $x(t_e)$ that are not affected by an `reinit`-equation $f_x(...)$ takes the value of $x(t_e)$, leaving the variable unchanged.

3.4 The Complete Hybrid DAE

The total equation system consisting of the combination of (1), (2), (3) and (4) is the desired *hybrid DAE* equation representation for Modelica models, consisting of *differential*, *algebraic*, and *discrete* equations.

This framework describes a system where the state evolves in two ways: continuously in time by changing the values of the state vector $x(t)$, and instantaneously during events triggered when some of the conditions $c(t_e)$ change value from `false` to `true`. The set of *state variables* from which other variables are computed is selected from the set of differentiated variables $x(t)$, algebraic variables $y(t)$, and discrete-time variables $q(t)$.

4 Simulation of Models Represented by Hybrid DAEs

4.1 Well-defined problem description

A Modelica *simulation problem* in the general case is a Modelica *model* that can be reduced to a hybrid DAE in the form of equations (1), (2), (3) and (4), together with additional constraints on variables and their derivatives called *initial conditions*.

The initial conditions prescribe initial start values of variables and/or their derivatives at simulation time=0 (e.g. expressed by the Modelica `start` attribute value of variables, with the attribute `fixed = true`), or default estimates of start values (the `start` attribute value with `fixed = false`).

The simulation problem is *well defined* provided that the following conditions hold:

- The total model system of equations is consistent and neither underdetermined nor overdetermined.
- The initial conditions are consistent and determine initial values for all variables.

- The model is specific enough to define a unique solution from the start simulation time t0 to some end simulation time t1.

The initial conditions of the simulation problem are often specified interactively by the user in the simulation tool, e.g. through menus and forms, or alternatively as default `start` attribute values in the simulation code. More complex initial conditions can be specified through `initial equation` sections in Modelica.

4.2 Simulation Techniques

There are three different kinds of equation systems resulting from the translation of a Modelica model to a flat set of equations, from the simplest to the most complicated and powerful:

- ODEs – Ordinary differential equations for continuous-time problems.
- DAEs – Differential algebraic equations for continuous-time problems
- Hybrid DAEs – Hybrid differential algebraic equations for mixed continuous-discrete problems.

In the following we present a short overview of methods to solve these kinds of equation systems. However, remember that these representations are strongly inter-related: an ODE is a special case of DAE without algebraic dependencies between states, whereas a DAE is a special case of hybrid DAEs without discrete or conditional equations. We should also point out that in certain cases a Modelica model results in one of the following two forms of purely algebraic equation systems, which can be viewed as DAEs without a differential equation part:

- Linear algebraic equation systems
- Nonlinear algebraic equation systems

However, rather than representing a whole Modelica model, such algebraic equation systems are usually subsystems of the total equation system.

4.3 The Notion of DAE Index

The DAE index is an important property of DAE systems. Consider once more a DAE system on the general form (neglecting the hybrid part, parameters and constants):

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

We assume that this system is solvable with a continuous solution, given an appropriate initial solution. There are several definitions of DAE *index* in the literature, of which the following, also called *differential index*, is informally defined as follows:

- The index of a DAE system (5) is the minimum number of times certain equations in the DAE must be differentiated in order to solve $\dot{x}(t)$ as a function of $x(t)$, $y(t)$, and $u(t)$, i.e. to transform the problem into ODE explicit state space form.

The index gives a classification of DAEs with respect to their numerical properties and can be seen as a measure of the distance between the DAE and the corresponding ODE

An ODE system on explicit state space form is of index 0 since it is already in the desired form:

$$\dot{x}(t) = f(t, x(t)) \tag{6}$$

The following *semi-explicit* form of DAE system is of index 1 under certain conditions:

$$\begin{aligned} \dot{x}(t) &= f(t, x(t), y(t)) && (a) \\ 0 &= g(t, x(t), y(t)) && (b) \end{aligned} \tag{7}$$

The condition is that the Jacobian of g with respect to y, $(\partial g / \partial y)$ – usually a matrix – is *non-singular* and therefore has a well-defined inverse. This means that in principle $y(t)$ can be solved as a function of $x(t)$ and substituted into (7a) to get state-space form. A DAE system in the general form (5) may have higher index than one. Mechanical models often lead to index 3 DAE systems. We conclude:

- There is no need for symbolic differentiation of equations in a DAE system if it is possible to determine the *highest order derivatives* as continuous functions of time and lower derivatives using stable numerical methods. In this case the index is at most 1.
- The index is zero for such a DAE system if there are no algebraic variables.

4.4 Mixed Symbolic and Numerical Solution of higher-index DAEs

A mixed symbolic and numerical approach to solution of DAEs avoids the problems of numeric differentiation. The DAE is transformed to a lower index problem by using index reduction. The standard mixed symbolic and numeric approach contains the following steps:

1. Use Pantelides algorithm to determine how many times each equation has to be differentiated to reduce the *index* to one or zero.
2. Perform *index reduction* of the DAE by analytic symbolic differentiation of certain equations and by applying the method of dummy derivatives.
3. Select the core state variables to be used for solving the reduced problem. These can either be selected statically during compilation, or in some cases selected dynamically during simulation.
4. Use a numeric ODE solver to solve the reduced problem.

In the following we will discuss the notions of index and index reduction in some more detail.

4.5 Higher Index Problems are Natural in Component-Based Models

The index of a DAE system is not a property of the modeled system but the *property* of a *particular model representation*, and therefore a function of the modeling methodology. A natural object-oriented component-based methodology with reuse and connections between physical objects leads to high index in the general case. The reason is the constraint equations resulting from setting variables equal across connections between separate objects.

Since the index is not a property of the modeled system it is possible to reduce the index by symbolic manipulations. High index indicates that the model has algebraic relations between differentiated state variables implied by algebraic relations between those state variables. By using knowledge about the particular modeling domain it is often possible to manually eliminate a number of differentiated variables, and thus reduce the index. However, this violates the object-oriented component-based modeling methodology for physical modeling that is intended to be supported by the Modelica language.

We conclude that high index models are natural, and that automatic index reduction is necessary to support a general object-oriented component-based modeling methodology with a high degree of reuse.

5 Finding Consistent Initial Values at Start or Restart

As we have stated briefly above, at the start of the simulation, or at restart after handling an event, it is required to find a consistent set of initial values or restart values of the variables of the hybrid DAE equation system before starting continuous DAE solution process.

At the *start* of the simulation these conditions are given by the initial conditions of the problems (including `start` attribute equations, equations in `initial equation` sections, etc., together with the system of equations defined by (1), (2), and (3). The user specifies the initial time of the simulation, t_0, and initial values or guesses of initial values of some of the continuous variables, derivatives, and discrete-time variables so that the algebraic part of the equation system can be solved at the initial time $t=t_0$ for all the remaining unknown initial values. In some application examples it is even necessary to calculate initial values of parameters (`fixed = false`), that afterwards be kept constant during simulation.

At *restart* after an event, the conditions are given by the *new values* of variables that have changed at the event, together with the current values of the remaining variables, and the system of equations (5), (6), and (7). The goal is the same as in the initial case, to solve for the new values of the remaining variables. In the initial case, however, the causality can be different since initial equations are included to calculate start values for the state variables, whereas at restart the state variables are always known.

6 Robust Initialization of Higher-Index DAEs

Initializing DAEs using the Modelica language has been quite cumbersome in the past, since initial equations have to be provided on the system level, where the user needs to know details on the underlying transformation and index-reduction algorithms, that are in general applied to simulate a Modelica model. Especially, when higher-index DAEs are involved the number of locally defined state variables no longer coincide with the number of state variables of the overall system. Although, one can influence the index-reduction algorithm by setting some attribute values (`stateSelect=always,prefer,…`), cases can be constructed which don't allow the straight forward prediction of the number of state variables left after transformation.

In order to make the initialization procedure more convenient a new concept is necessary, which allows to define the initial equations locally in each relevant component where the corresponding states appear, even if these states are eliminated during index-reduction. Naturally, this leads to an overdetermined system of equations, which has to be solved during the initialization process. In this context, we call a higher-index problem "well-posed" if enough equations of the system are redundant so that initial values can be determined which fulfill the whole set of initial equations. The main idea of the new approach is to reformulate the problem of finding roots of the set of non-linear equations to an equivalent optimization problem.

Considering the general mathematical description of the initialization problem:

$$\begin{aligned} f_1(z_1,...,z_n) &= 0 \\ &\vdots \\ f_m(z_1,...,z_n) &= 0 \end{aligned} \tag{8}$$

Cases where $m \geq n$ means that more equations (m) than variables (n) are given. Every solution to (8) minimizes the problem:

$$F(z_1,...,z_n) = \sum_{i=1}^{m} f_i(z_1,...,z_n)^2 \rightarrow \min \tag{9}$$

On the other hand, every global minimum of (9) is a solution to (8). In order to solve (9) a number of different algorithms have been developed during the past. The algorithm can be categorized depending on the order of derivatives needed during the solution process. In the OpenModelica environment the Simplex-method of Nelder and Mead as well as the Brent's method are currently implemented, only working with the minimization function F. The OpenModelica prototype already shows reliable results for the evaluated examples.

Further improvements can be achieved as soon as the Jacobian of F with regards to the unknown is available. In that case, more advanced algorithms like the method of Fletcher-Reeves, Quasi-Newton, and/or Levenberg-Marquardt methods can be applied which would provide a speed-up in convergence. We regard this as a quality of implementation, since the described approach is working in principle already.

7 Test and Evaluation with OpenModelica

Consider the following electrical 3-phase power system, where two generating units `VS1` and `VS2` are connected via a transmission line modeled by components LR1 and LR2.

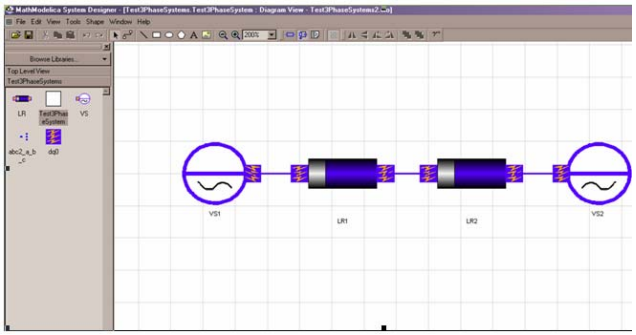

Fig. 1. An electrical power system where two generating units `vs1` and `vs2` are connected via a transmission line.

The connectors are written in `dq0`-coordinates implementing the potential variable `u_dq0` and the flow variable `i_dq0`. These quantities are constant in case of a nondistributed steady state, which is generally assumed during the initialization process. Introducing the Park-Transformation *P* the 3-phase rotating system (voltages `u_abc` and currents `i_abc`) can be calculated from the `dq0`-representation and vice versa.

The transmission line (`LR1` and `LR2`) is modeled by a purely inductive and resistive component, based on the Modelica Electrical Library. Since `LR1` and `LR2` are connected in series, giving a higher index system, index reduction has to be applied for simulation purposes.

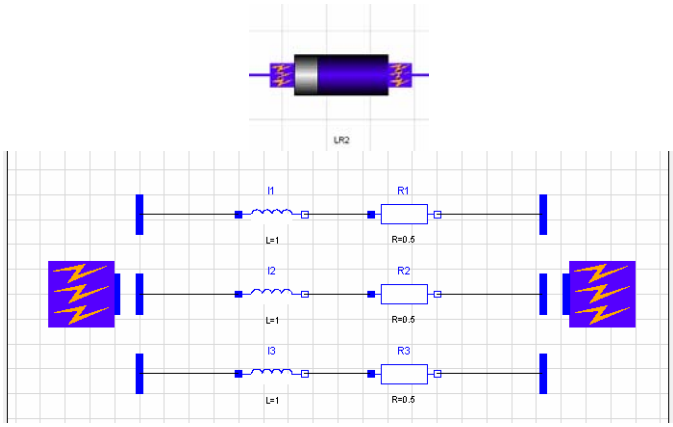

Fig. 2. LR2 component with dq0 connectors.

The voltage source is described similarly using the Modelica Standard Library combined with the `dq0`-connectors.

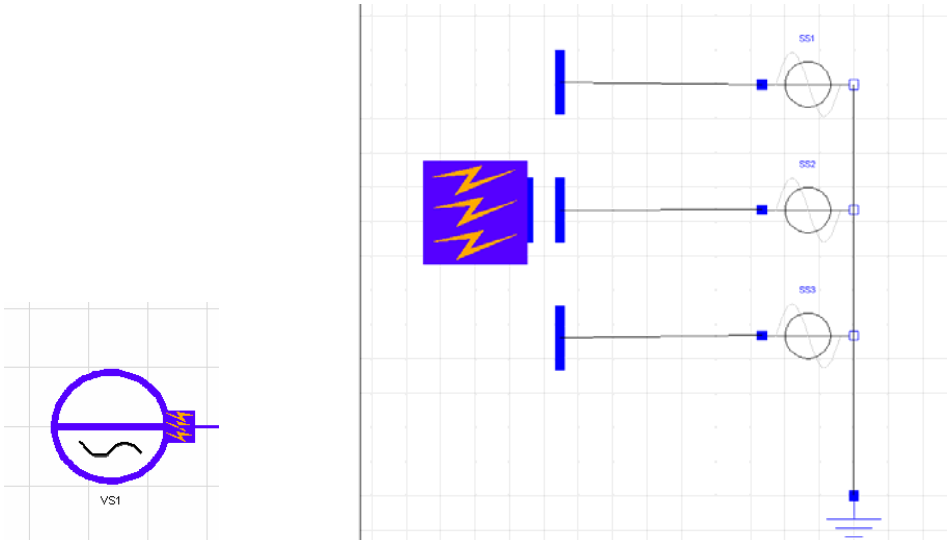

Fig. 3. Voltage source.

In order to initialize the model correctly to steady state the following initial equations have been added to the local components `LR1` and `LR2`.

```
model LR
  ...
equation
  ...
initial equation
  der(dq0_1.i_dq0)={0,0,0};
end LR;
```

Due to the higher-index of the overall system, index-reduction is applied. The system finally is determined by 3 state variables `LR1.I1.i`, `LR1.I2.i`, `LR1.I3.i`. The corresponding initial equation system has 3 equations more than number of unknowns, but these equations are redundant and could be eliminated. Due to the involvement of the Park-transformation, redundancy is not easy to detect. However, applying the concept described above correct initialization of the system is performed.

8 Conclusions and Future work

In this paper we have presented an overview of our implementation of initializing Modelica models in the OpenModelica compiler. A new concept has been developed to describe the initial equations locally in the relevant component where the corresponding states appear, that also works for arbitrary well-posed higher-index problems. Due to the necessary index reduction some of the states get changed to dummy states that means that they will be algebraic during the simulation of the model. The

corresponding initial equations are therefore redundant, but can be handled correctly by the new initialization process, if they are consistent. If not, an error/warning is issued to the user.

The described method has been implemented in the OpenModelica compiler. In the future we also wish to implement calculation of the Jacobian matrix of the equation system with regards to the state variables. This gives the possibility to implement more advanced and robust numerical algorithms in order to solve the corresponding optimization (minimization) problem during initialization of the DAE.

9 Acknowledgements

This work was supported by the University of Applied Sciences in Bielefeld, by MathCore Engineering AB, by the Swedish Research Council (VR), and by SSF in the VISIMOD project.

References

[1] Peter Fritzson, et al. The Open Source Modelica Project. In Proceedings of The 2nd International
Modelica Conference, 18-19 March, 2002. Munich, Germany See also:
http://www.ida.liu.se/projects/OpenModelica.

[2] Peter Fritzson. Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, 940 pp., ISBN 0-471-471631, Wiley-IEEE Press, 2004.

[3] The Modelica Association. The Modelica Language Specification Version 2.2, March 2005. http://www.modelica.org.

[4] The OpenModelica Users Guide, version 0.6, June 2005.
www.ida.liu.se/projects/OpenModelica

[5] The OpenModelica System Documentation, version 0.6, June 2006.
www.ida.liu.se/projects/OpenModelica

[6] K. E. Brenan, S. L. Campbell, and L. R. Petzold, Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations, Elsevier, New York, 1989.

[7] B. Bachmann et. al. (Modelica Association): Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification. 2002.

[8] P. Fritzson, P. Aronsson, P. Bunus, V. Engelson, L. Saldamli, H. Johansson, A. Karström: The Open Source Modelica Project. In: 2nd Modelica Conference 2002, Oberpfaffenhofen, 2002

[9] S.-E. Mattson, H. Olson, H. Elmqvist: Dynamic Selection of States in Dymola. In: 1st Modelica Workshop 2000, Lund, Sweden, 2000

[10] M. Otter: Objektorientierte Modellierung Physikalischer Systeme (Teil 4) – Transformationsalgorithmen. In: at Automatisierungstechnik, Oldenbourg Verlag München, 1999

[11] M. Otter, B. Bachmann: Objektorientierte Modellierung Physikalischer Systeme (Teil 5,6) – Singuläre Systeme. In: at Automatisierungstechnik, Oldenbourg Verlag München, 1999

[12] R. Fletcher: Practical Methods of Optimization John Wiley & Sons, 1995

[13] J. Stoer, R. Burlisch: Einführung in die numerische Mathematik. Springer Verlag, 1994

[14] S.E. Mattsson, G. Söderlind: Index reduction in differential-algebraic equations using dummy derivatives. SIAM Journal of Scientific and Statistical Computing, Vol. 14, 1993.

[15] K.E. Brenan., S.L. Campbell, L.R. Petzold: Numerical Solution of Initial Value Problems in Differential Algebraic Equations. North-Holland, Amsterdam, 1989

[16] C.C. Pantelides: The Consistent Initialization of Differential-Algebraic Systems, SIAM Journal of Scientific and Statistical Computing, 1988.

[17] L.R. Petzold: A description of DASSL: A differential / algebraic system solver. Sandia National Laboratories, Albuquerque, 1982

[18] H. Elmqvist: A Structured Model Language for Large Continuous Systems, PhD dissertation, Department of Automatic Control, Lund Institute of Technology, Lund, Schweden, 1978

[19] R.E. Tarjan: Depth First Search and Linear Graph Algorithms. SIAM Journal of Comp., Nr. 1, 1972