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

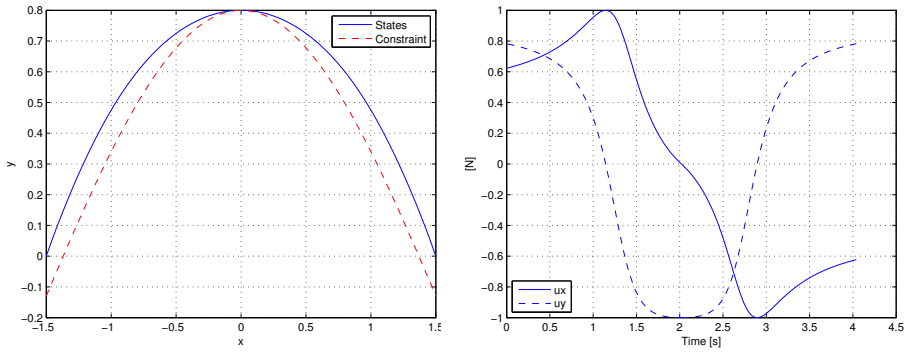

Fig. 1. Resulting optimization profiles for the minimum time case.

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:

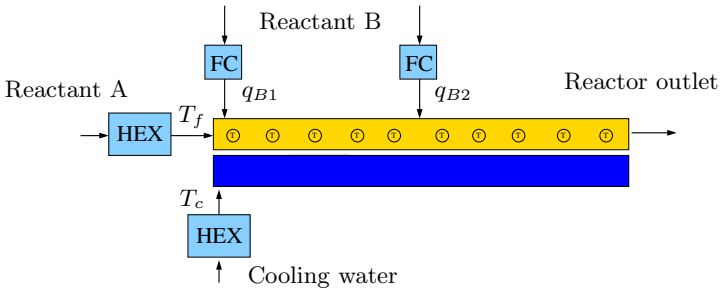

Fig. 2. The reactor shown as a schematic tubular reactor. There are four inflows to the process and there is one manipulated variable for each inflow; q_{B1}, q_{B2}, T_f and T_c. Each inflow has an actuator subsystem that provides flow control (FC) or temperature control through heat exchangers (HEX).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

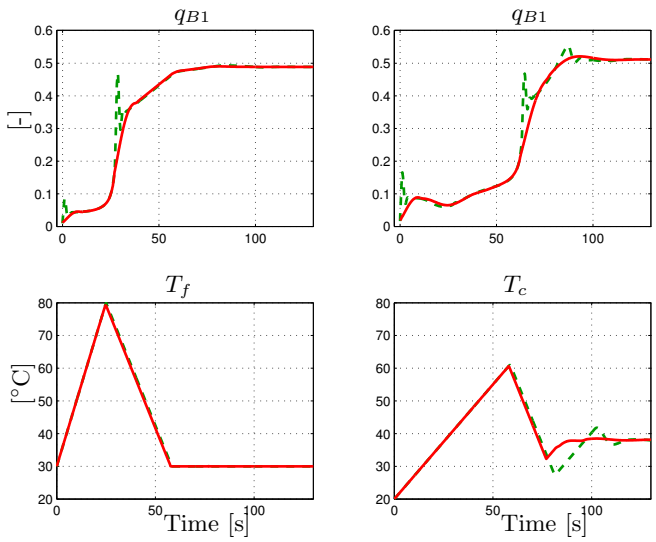

Fig. 3. Optimal control profiles. The dashed curves correspond to a case with a small high frequency penalty on the inputs, whereas the solid curves represents a case with a larger high frequency penalty, resulting in smoother control profiles.

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.

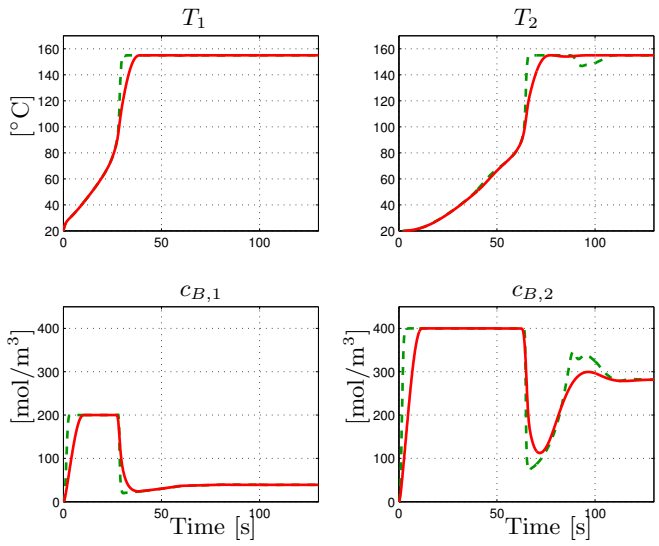

Fig. 4. Optimal profiles profiles for reactor temperature and concentration of substance B. The left plots correspond to the first injection point, whereas the right plots correspond to the second injection point.

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.