Modeling and optimization with Optimica and JModelica.org—Languages and tools for solving large-scale dynamic optimization problems

J. Åkesson\textsuperscript{a,b,*}, K.-E. Årzén\textsuperscript{a}, M. Gäfvert\textsuperscript{b}, T. Bergdahl\textsuperscript{b}, H. Tummescheit\textsuperscript{b}

\textsuperscript{a} Department of Automatic Control, Lund University, P.O. Box 118, SE-221 00 Lund, Sweden
\textsuperscript{b} Modelon AB, Ideon Science Park, SE-223 70 Lund, Sweden

\textbf{ARTICLE INFO}

\textbf{Article history:}
Received 1 May 2009
Received in revised form 20 October 2009
Accepted 8 November 2009
Available online 29 December 2009

\textbf{Keywords:}
Modelica
Optimica
Dynamic optimization
Model predictive control

\textbf{ABSTRACT}

The Modelica language, targeted at modeling of complex physical systems, has gained increased attention during the last decade. Modelica is about to establish itself as a de facto standard in the modeling community with strong support both within academia and industry. While there are several tools, both commercial and free, supporting simulation of Modelica models, few efforts have been made in the area of dynamic optimization of Modelica models. In this paper, an extension to the Modelica language, entitled Optimica, is reported. Optimica enables compact and intuitive formulations of optimization problems, which are static and dynamic, based on Modelica models. The paper also reports a novel Modelica-based open source project, JModelica.org, specifically targeted at dynamic optimization. JModelica.org supports the Optimica extension and offers an open platform based on established technologies, including Python, C, Java and XML. Examples are provided to demonstrate the capabilities of Optimica and JModelica.org.

© 2009 Elsevier Ltd. All rights reserved.

1. Introduction

The complexity of products and processes is rapidly increasing. This complexity stems both from the increasing size of systems, and increasing heterogeneity of systems. In order to manage such complexity and to encode system knowledge in a structured way, modeling languages, primarily intended for simulation of dynamic models, have been traditionally used. An example of such a language is Modelica (The Modelica Association, 2007).

While modeling and simulation remain important tools for engineers in many disciplines, the landscape is shifting towards a more flexible and diverse use of model-based design methodologies. This trend raises new demands on associated tools. In particular, model-based process and product development commonly includes activities such as simulation, parameter sensitivity analysis, design optimization and control system development and deployment. Commonly, these activities are performed in a highly iterative manner. Fig. 1 illustrates a typical work cycle. Accordingly, flexibility and interoperability are key success factors for algorithms and tools in order to meet future challenges.

This paper focuses on dynamic optimization in the context of high-level modeling languages, in particular Modelica. Optimization is used extensively in many industrial branches, in particular in the process industry. Applications include design optimization to develop optimal processes, set-point optimization to minimize raw material and energy consumption, and on-line optimal control strategies such as model predictive control. While there are several tools supporting dynamic optimization, they are typically restricted to a particular modeling domain and a particular numerical algorithm. In this paper, a different approach is taken. In particular, the problem of optimizing heterogeneous physical systems while enabling use of a wide range of numerical algorithms is considered.

Sophisticated numerical optimization algorithms often have cumbersome APIs, which do not always match the engineering need for high-level description formats. For example, it is not uncommon for such numerical packages to be written in C, or in Fortran, and that they require the dynamic system to be modeled as an ODE/DAE, which is also encoded in C or Fortran. In addition, it may be required to also encode first and second order derivatives. Although there are efficient tools for automatic differentiation encoding of dynamic optimization problems in low-level languages\textsuperscript{1} like C or FORTRAN is often cumbersome and error-prone. An important goal of developing high-level languages for dynamic optimization is therefore to bridge the gap between the engineering need for high-level descriptions and the APIs of numerical algorithms.

\textsuperscript{*} Corresponding author. Tel.: +46 46 2228797.
E-mail addresses: johan.akesson@control.lth.se (J. Åkesson), karl-erik.arzen@control.lth.se (K.-E. Årzén), magnus.gaevernt@modelon.se (M. Gäfvert), tove.bergdahl@modelon.se (T. Bergdahl), hubertus.tummescheit@modelon.se (H. Tummescheit).

0098-1354/S – see front matter © 2009 Elsevier Ltd. All rights reserved.
This paper presents an extension of the Modelica language (The Modelica Association, 2007), entitled Optimica, dedicated to high-level formulation of dynamic optimization problems based on Modelica models. Optimica consists of a few but powerful new language constructs that enable the user to express optimization problems using high-level descriptions on par with Modelica. Using the Optimica extension, optimization interval, cost function, constraints and information related to the numerical algorithm used to solve the problem can be specified. A key feature of Optimica is that the formulation of the optimization problem is done independently of the numerical algorithm used to solve the problem. Rather, the algorithm and associated control parameters are specified using light weight language constructs that make it easy to evaluate the applicability of several different algorithms.

In order to demonstrate the feasibility and effectiveness of the proposed Optimica extension, a Modelica-based open source platform called JModelica.org (Modelon AB, 2009) is under development. The objective of the JModelica.org platform is to create a flexible and extensible Modelica environment focused on optimization. The software consists of several parts; compiler front-ends to transform Modelica and Optimica code into a canonical representation, compiler back-ends to generate efficient model code (currently C and XML), implementations of dynamic optimization algorithms. In addition, the compilers and algorithms have been interfaced with Python in order to offer a convenient scripting and prototyping environment. The main objective of the project is to create an industrially viable Modelica environment for optimization, yet offering rich opportunities for research. In particular, JModelica.org forms a vehicle for communicating industrial relevant applications into the academic community and for propagating state of the art numerical algorithms developed in the academic community into industrial use. JModelica.org can also be used for experimental language design where Modelica is extended with new functionality, or as a comprehensive environment for teaching. For more information, see the JModelica.org home page (Modelon AB, 2009).

The paper is outlined as follows. In Section 2, backgrounds on Modelica, dynamic optimization, and optimization tools is given. In Section 3, the Optimica extension is presented. Section 4 gives an overview of the JModelica.org open source project, and examples are given in Section 5. The paper ends with a summary in Section 6.

2. Background

2.1. Modelica

Modelica is the result of an effort to create a unified modeling language for complex heterogeneous physical systems. The effort started in the mid nineties with the objective of gathering practitioners from several application fields, including thermo-fluid systems, robotics, electronics, mechanics and avionics. In addition, researchers specializing in computer science were part of the group that published the first version of the Modelica language specification in 1997. The language is continuously maintained and developed by the non-profit organization The Modelica Association (2007)—since the start in 1997 there have been about 60 design meetings where the language has been discussed and improved. The latest version of the specification, Modelica 3.0, was published in September 2007.

The Modelica language builds on principles of earlier languages, notably Omola (Mattsson & Andersson, 1992). Also, the bond graph formalism (Karnopp & Rosenberg, 1968), influenced the design. One of the main targets of Modelica is to enable modeling of different physical domains in one unified language. This approach differs from some other modeling frameworks such as SPICE (Nagel & Pederson, 1973) and VHDL-AMS (IEEE, 1997) which are specialized on electrical circuits, gPROMS (Process Systems Enterprise, 2007) specialized in chemical processes, and ADAMS (MSC Software, 2007) specialized in simulation of mechanical systems. While this heterogeneous modeling strategy enables a high degree of flexibility, it should also be noted that Modelica lacks some specialized features, for example intrinsic PDE support.

Modelica shares several properties with other modern modeling languages, including gPROMS and VHDL-AMS. Modelica is an object-oriented language, supporting fundamental concepts such as packages, classes, inheritance, and components. These abstractions enable structuring and reuse of complex models. Modelica is based on acasual equations, rather than assignment statements. This property has far-reaching consequences in terms of ease of use, since it enables the user to input equations, both differential and algebraic, in a natural text-book style. In particular, this features eliminates the need to solve for the derivatives. In order to enable component-based modeling, Modelica offers a mechanism to explicitly model acasual physical interfaces: connectors. A connector models the physical quantities involved in the interface by means of through and across variables, e.g., flow and pressure. When a component with connectors of the same type is connected, the signal directions need not be determined by the user, but are rather given by the topology of the component network. This methodology is fundamentally different from that of block-based modeling used in, e.g., Simulink, see Åström, Elmqvist, and Mattsson (1998) for a discussion. Modelica also supports joint textual and graphical modeling, mixed continuous discrete (hybrid behavior), user defined functions, and interfacing with external C or Fortran code. For a comprehensive overview of Modelica, see Fritzson (2004).

Typically, Modelica code is organized into reusable model libraries. Commonly, domain expert knowledge is encoded in the form of Modelica libraries, which are then used by application engineers to construct model instances corresponding to a particular process or plant. The users of Modelica can then be roughly categorized into two groups: library developers and application developers. Library developers typically work with encoding of the laws of nature in the form of differential algebraic equations and structuring of the code. Application engineers on the other hand, typically use predefined components and connect them together in a graphical model editor. There are numerous Modelica libraries available, both commercially and freely available, ranging from thermodynamics and power plants to automotive and space applications. There is also a freely available standard library for Modelica from The Modelica Association (2007).

While the Modelica language provides a convenient modeling environment, Modelica code is typically not immediately suitable for integration with numerical algorithms. A Modelica tool therefore needs to perform several transformation steps in order to produce model code which in turn can be interfaced with an algorithm for simulation or optimization. The procedure is illustrated
in Fig. 2. In a first step called flattening, the hierarchical Modelica code is transformed into a flat representation consisting essentially of variables and equations. This representation corresponds to a hybrid DAE, but it is still not suitable for use with numerical algorithms, since it may be of high index, and it may also contain several alias variables which can be eliminated. In a second step, structural and symbolic algorithms are applied in order transform the DAE into a structured form. The equations are then sorted according to Tarjan’s algorithm (Tarjan, 1972) and index reduction is performed if needed (Mattsson & Söderlind, 1993). The result is a structured hybrid DAE of index one. In a final step, efficient code (typically C code) is generated. This code, in turn, is suited for integration with numerical algorithms.

There are several tools supporting Modelica, both commercial and free. Commercial tools include Dymola (Dynasin, 2009), Simulation X (ITI, 2009) and MapleSim (Maplesoft, 2009). The open source project OpenModelica (PELAB, 2009) offers a free Modelica environment which is mainly targeted at simulation.

2.2. Dynamic optimization

During the last five decades, the theory of dynamic optimization has received much attention. In 1957, Bellman formulated the celebrated principle of optimality, and showed that dynamic programming was applicable to a broad range of applications (Bellman, 1957). Following this work, dynamic programming has been applied to various fields, notably inventory control, economics, statistics, and engineering. For a modern description, see Bertsekas (2000) and Bertsekas (2000). Using dynamic programming, an optimal control law can be derived from the solution of a partial differential equation, the Hamilton–Jacobi–Bellman equation.

Another important contribution to the theory of optimal control is the maximum principle, which was presented by Pontryagin, Boltyanski, Gamkrelidze, and Mishchenko (1962). Whereas dynamic programming provides a closed loop control law (the control law as a function of the system states), the maximum principle provides the necessary conditions for an open loop control law (the control law as a function of time) to be optimal.²

Both dynamic programming and the maximum principle have practical drawbacks that make them hard to apply to large scale systems. For example, in the presence of inequality constraints, the activation sequence must be known a priori. Also, it may be difficult to find initial guesses for adjoint variables. In the last two decades, a new family of methods has emerged to overcome these difficulties. These methods are referred to as direct methods. Direct methods attempt to solve dynamic optimization problems by transcribing the original infinite dimensional dynamic problem into a finite dimensional static optimization problem. There are two main branches within the family of direct methods, referred to as sequential and simultaneous methods, see Binder et al. (2001) for an overview. The sequential methods rely on state of the art numerical integrators, typically also capable of computing state sensitivities, and standard nonlinear programming (NLP) codes. The controls

² There are some exceptions, where the maximum principle yields a closed loop control law, for example in the case of linear systems and quadratic cost.
In the third category we have numerical packages for dynamic optimization, often developed as part of research programs. Examples are ACADO (OPTEC K.U. Leuven, 2009), Muscod II (University of Heidelberg, 2009), and DynoPC (Lang & Biegler, 2007), which is based on Ipopt (Wächter & Biegler, 2006). Such packages are typically focused on efficient implementation of an optimization algorithm for a particular class of dynamic systems. Also, detailed information about the model to optimize is generally required in order for such algorithms to work, including accurate derivatives and in some cases also sparsity patterns. Some of the packages in this category are also targeting optimal control and estimation problems in real-time, e.g., non-linear model predictive control, which require fast convergence. While these packages offer state of the art algorithms, they typically come with simple or no user interface. Their usage is therefore limited due to the effort required to code the model and optimization descriptions.

The JModelica.org platform is positioned to fill the gap left between simulation tools offering optimization capabilities and state of the art numerical algorithms. In comparison with gPROMS/gOPT, the main advantages of JModelica.org are that it is based on an open standard, Modelica. It comes with open interfaces suitable for algorithm integration, and that it is freely available as open source software. Now, gPROMS/gOPT are highly sophisticated products which have been applied in a large number of industrial applications. JModelica.org, on the other hand, is currently more geared towards experimental algorithm development, research and education.

Primarily, target algorithms of JModelica.org are gradient-based methods offering fast convergence. Never the less, JModelica.org is well suited for use also with heuristic direct search methods; the requirements with respect to execution interface is typically a subset of the requirements for gradient based methods. The problems addressed by model integration tools is currently beyond the scope of JModelica.org, even though its integration with Python offers extensive possibilities to develop custom applications based on the solution of simulation and optimization problems.

3. Optimica

3.1. Information structure

In order to formulate a dynamic optimization problem, to be solved by a numerical algorithm, the user must supply different kinds of information. It is natural to categorize this information into three levels, corresponding to increasing levels of detail.

- **Level I.** At the mathematical level, a canonical formulation of a dynamic optimization problem is given. This include variables and parameters to optimize, cost function to minimize, constraints, and the Modelica model constituting the dynamic constraint. The optimization problem formulated at this level is in general infinite dimensional, and is thereby only partial in the respect that it cannot be directly used by a numerical algorithm without additional information, for example, concerning transcription of continuous variables.

- **Level II.** At the transcription level, a method for translating the problem from an infinite dimensional problem to a finite dimensional problem needs to be provided. This might include discretization meshes as well as initial guesses for optimization parameters and variables. It should be noticed that the information required at this level is dependent on the numerical algorithm that is used to solve the problem.

- **Level III.** At the algorithm level, information such as tolerances and algorithm control parameters may be given. Such parameters are often critical in order to achieve acceptable performance in terms of convergence, numerical reliability, and speed.

An important issue to address is whether information associated with all levels should be given in the language extension. In Modelica, only information corresponding to Level I is expressed in the actual model description. Existing Modelica tools then typically use automatic algorithms for critical tasks such as state selection and calculation of consistent initial conditions, although the algorithms can be influenced by the user via the Modelica code, by means of annotations, or attributes, such as StateSelect. Yet other information, such as choice of solver, tolerances and simulation horizon is provided directly to the tool, either by means of a graphical user interface, a script language, or alternatively, in annotations.

For dynamic optimization, the situation is similar, but the need for user input at the algorithm level is more emphasized. Automatic algorithms, for example for mesh selection, exist, but may not be suitable for all kinds of problems. It is therefore desirable to include, in the language, means for the user to specify most aspects of the problem in order to maintain flexibility, while allowing for automatic algorithms to be used when possible and suitable.

Relating to the three levels described above, the approach taken in the design of Optimica is to extend the Modelica language with a few new language constructs corresponding to the elements of the mathematical description of the optimization problem (Level I). The information included in Levels II and III, however, may rather be specified by means of annotations.

3.2. Dynamic system model

The scope of Optimica can be separated into two parts. The first part is concerned with the class of models that can be described in Modelica. Arguably, this class is large, since very complex, non-linear and hybrid behavior can be encoded in Modelica. From a dynamic optimization perspective, the inherent complexity of Modelica models is a major challenge. Typically, different algorithms for dynamic optimization support different model structures. In fact, the key to developing efficient algorithms lies in exploiting the structure of the model being optimized. Consequently, there are different algorithms for different model structures, such as linear systems, non-linear ODEs, general DAEs, and hybrid systems. In general, an algorithm can be expected to have better performance, in terms of convergence properties and shorter execution times, if the model structure can be exploited. For example, if the model is linear, and the cost function is quadratic, the problem can be obtained very efficiently by solving a Riccati equation. On the other hand, optimization of general non-linear and hybrid DAEs is still an area of active research, see for example Barton & Lee, 2002. As a result, the structure of the model highly affects the applicability of different algorithms. The Optimica compiler presented in this paper relies on a direct collocation algorithm in order to demonstrate the proposed concept. Accordingly, the restrictions imposed on model structure by this algorithm apply when formulating the Modelica model, upon which the optimization problem is based. For example, this excludes the use of hybrid constructs, since the right hand side of the dynamics is assumed to be twice continuously differentiable. Obviously, this restriction excludes optimization of many realistic Modelica models. On the other hand, in some cases, reformulation of discontinuities to smooth approximations may be possible in order to enable efficient optimization. This is particularly important in on-line applications. The Optimica extension, as presented in this paper, could also be...
extended to support other algorithms, which are indeed applicable to a larger class of models.

3.3. The dynamic optimization problem

The second part of the scope of Optimica is concerned with the remaining elements of the optimization problem. This includes cost functions, constraints and variable bounds. Consider the following formulation of a dynamic optimization problem:

\[
\min_{u(t), p} \psi(\bar{x}, p) \quad (1)
\]

subject to the dynamic system:

\[
F(\bar{x}(t), x(t), y(t), u(t), p, t) = 0, \quad t \in [t_0, t_f] \quad (2)
\]

and the constraints:

\[
eq_{\text{ineq}}(x(t), y(t), u(t), p) \leq 0, \quad t \in [t_0, t_f] \quad (3)
\]

\[
eq_{\text{eq}}(\bar{x}, p) = 0 \quad (4)
\]

\[
e^p_{\text{ineq}}(\bar{x}, p) \leq 0 \quad (5)
\]

\[
e^p_{\text{eq}}(\bar{x}, p) = 0 \quad (6)
\]

where \( x(t) \in \mathbb{R}^{n_x} \) are the dynamic variables, \( y(t) \in \mathbb{R}^{n_y} \) are the algebraic variables, \( u(t) \in \mathbb{R}^{n_u} \) are the control inputs, and \( p \in \mathbb{R}^{n_p} \) are parameters which are free in the optimization. In addition, the optimization is performed on the interval \( t \in [t_0, t_f] \), where \( t_0 \) and \( t_f \) can be fixed or free, respectively. The initial values of the dynamic and algebraic variables may be fixed or free in the optimization. The vector \( \bar{z} \) is composed from discrete time points of the states, controls and algebraic variables:

\[
\bar{z} = \{x(t_1), \ldots, x(t_{N_p}), y(t_1), \ldots, y(t_{N_p}), u(t_1), \ldots, u(t_{N_p})\}^T, \quad t_i \in [t_0, t_f], \quad N_p \text{ denotes the number of time points included in the optimization problem.}
\]

The constraints include inequality and equality path constraints, (3) and (4). In addition, inequality and equality point constraints, (5) and (6), are supported. Point constraints are typically used to express initial or terminal constraints, but can also be used to specify constraints for time points in the interior of the interval.

The cost function (1) is a generalization of a terminal cost function, \( \phi(t_f) \), in that it admits inclusion of variable values at other time instants. This form includes some of the most commonly used cost function formulations. A Lagrange cost function can be obtained by introducing an additional state variable, \( \chi_i(t) \), with the associated differential equation \( \dot{x}_i(t) = L(x(t), u(t)) \), and the cost function \( \psi(t_f) = \chi_i(t_f) \). The need to include variable values at discrete points in the interior of the optimization interval in the cost function arises for example in parameter estimation problems. In such cases, a sequence of measurements, \( y_d(t_i) \), obtained at the sampling instants \( t_i \), \( i = 1 \ldots N_p \) is typically available. A cost function candidate is then:

\[
\sum_{i=1}^{N_p} (y(t_i) - y_d(t_i))^T W (y(t_i) - y_d(t_i)) \quad (7)
\]

where \( y(t_i) \) is the model response at time \( t_i \) and \( W \) is a weighting matrix.

Another important class of problems is static optimization problems on the form:

\[
\min_{u, p} \psi(x, y, u, p) \quad (1)
\]

subject to

\[
F(0, x, y, u, p, t_i) = 0 \quad (8)
\]

\[
eq_{\text{ineq}}(x, u, p) \leq 0
\]

In this case, a static optimization problem is derived from a, potentially, dynamic Modelica model by setting all derivatives to zero. Since the problem is static, all variables are algebraic and accordingly, no transcription procedure is necessary. The variable \( t_i \) denotes the time instant at which the static optimization problem is defined.

In this paper a direct collocation method (see for example Biegler et al., 2002) will be used to illustrate how the transcription step can also be encoded in the Optimica extension. The information that needs to be provided by the user is then a mesh specification, the collocation points, and the coefficients of the interpolation polynomials.

3.4. The Optimica extension

In this section, the Optimica extension will be presented and informally defined. The presentation will be made using the following dynamic optimization problem, based on a double integrator system, as an example:

\[
\min_{u(t)} \int_0^{t_f} 1 \, dt \quad (9)
\]

subject to the dynamic constraint:

\[
\dot{x}(t) = u(t), \quad x(0) = 0 \quad (10)
\]

\[
\dot{v}(t) = u(t), \quad v(0) = 0 \quad (10)
\]

and

\[
x(t_f) = 1, \quad v(t_f) = 0 \quad (11)
\]

In this problem, the final time, \( t_f \), is free, and the objective is thus to minimize the time it takes to transfer the state of the double integrator from the point \( (0, 0) \) to \( (1, 0) \), while respecting bounds on the velocity \( v(t) \) and the input \( u(t) \).

model DoubleIntegrator

Real x(start=0);
Real v(start=0);
input Real u;

equation

der(x)=v;
der(v)=u;
end DoubleIntegrator;

Listing 1: A Modelica model of a double integrator system.

In summary, the Optimica extension consists of the following elements:

- A new specialized class: optimization.
- New attributes for the built-in type Real: free and initialGuess.
- A new function for accessing the value of a variable at a specified time instant.
- Class attributes for the specialized class optimization: objective, startTime, endTime and static.
- A new section: constraint.
- Inequality constraints.
- An annotation for providing transcription information.

3.4.1. A new specialized class

It is convenient to introduce a new specialized class, called optimization, in which the proposed Optimica-specific constructs are valid. This approach is consistent with the Modelica language, since there are already several other specialized classes,e.g., record, function and model. By introducing a new specialized
class, it also becomes straightforward to check the validity of a program, since the Optimica-specific constructs are only valid inside an optimization class. The optimization class corresponds to an optimization problem, static or dynamic, as specified in Section 3.3. Apart from the Optimica-specific constructs, an optimization class can also contain component and variable declarations, local classes, and equations.

It is not possible to declare components from optimization classes in the current version of Optimica. Rather, the underlying assumption is that an optimization class defines an optimization problem, that is solved off-line. An interesting extension would, however, be to allow for optimization classes to be instantiated. With this extension, it would be possible to solve optimization problems, on-line, during simulation. A particularly interesting application of this feature is model predictive control, which is a control strategy that involves on-line solution of optimization problems during execution.

As a starting-point for the formulation of the optimization problems (9)-(11), consider the optimization class:

```modelica
optimization DimInTime
  DoubleIntegrator di;
end DimInTime;
```

This class contains only one component representing the dynamic system model, but will be extended in the following to incorporate also the other elements of the optimization problem.

3.4.2. Attributes for the built-in type Real

In order to superimpose information on variable declarations, two new attributes are introduced for the built-in type Real.

Firstly, it should be possible to specify that a variable, or parameter, is free in the optimization. Modelica parameters are normally considered to be fixed after the initialization step, but in the case of optimization, some parameters may rather be considered to be free. In optimal control formulations, the control inputs should be marked as free, to indicate that they are indeed optimization variables. For these reasons, a new attribute for the built-in type Real, free, of Boolean type is introduced. By default, this attribute is set to false.

Secondly, an attribute, initialGuess, is introduced to enable the user to provide an initial guess for variables and parameters. In the case of free optimization parameters, the initialGuess attribute provides an initial guess to the optimization algorithm for the corresponding parameter. In the case of variables, the initialGuess attribute is used to provide the numerical solver with an initial guess for the entire optimization interval. This feature is particularly important if a simultaneous or multiple-shooting algorithm is used, since these algorithms introduce optimization variables corresponding to the values of variables at discrete points over the interval. Notice that such initial guesses may be needed both for control and state variables. For such variables, however, the proposed strategy for providing initial guesses may sometimes be inadequate. In some cases, a better solution is to use simulation data to initialize the optimization problem. This approach is also supported by the Optimica compiler.

In the double integrator example, the control variable $u$ is a free optimization variable, and accordingly, the free attribute is set to true. Also, the initialGuess attribute is set to 0.0.

```modelica
optimization DimInTime
  DoubleIntegrator di(u(free=true,
                        initialGuess=0.0));
end DimInTime;
```

4 The same attributes may be introduced for the built-in type Integer, in order to support also variables of type Integer in the optimization formulation.

3.4.3. A function for accessing instant values of a variable

An important component of some dynamic optimization problems, in particular parameter estimation problems where measurement data is available, is variable access at discrete time instants. For example, if a measurement data value, $y_i$, has been obtained at time $t_i$, it may be desirable to penalize the deviation between $y_i$ and a corresponding variable in the model, evaluated at the time instant $t_i$. In Modelica, it is not possible to access the value of a variable at a particular time instant in a natural way, and a new construct therefore has to be introduced.

All variables in Modelica are functions of time. The variability of variables may be different—some are continuously changing, whereas others can change value only at discrete time instants, and yet others are constant. Nevertheless, the value of a Modelica variable is defined for all time instants within the simulation, or optimization, interval. The time arguments of variables are not written explicitly in Modelica, however. One option for enabling access to variable values at specified time instants is therefore to associate an implicitly defined function with a variable declaration. This function can then be invoked by the standard Modelica syntax for function calls, $y_{(t,i)}$. The name of the function is identical to the name of the variable, and it has one argument; the time instant at which the variable is evaluated. This syntax is also very natural since it corresponds precisely to the mathematical notation of a function. Notice that the proposed syntax $y_{(t,i)}$ makes the interpretation of such an expression context dependent. In order for this construct to be valid in standard Modelica, $y$ must refer to a function declaration. With the proposed extension, $y$ may refer either to a function declaration or a variable declaration. A compiler therefore needs to classify an expression $y_{(t,i)}$ based on the context, i.e., what function and variable declarations are visible. An alternative syntax would have been to introduce a new built-in function, that returns the value of a variable at a specified time instant. While this alternative would have been straightforward to implement, the proposed syntax has the advantages of being easier to read and that it more closely resembles the corresponding mathematical notation. This feature of Optimica is used in the constraint section of the double integrator example, and is described below.

3.4.4. Class attributes

In the optimization formulations (1)–(6) and (8), there are elements that occur only once, i.e., the cost function and the optimization interval in (1)–(6), and in the static case (8), only the cost function. These elements are intrinsic properties of the respective optimization formulations, and should be specified, once, by the user. In this respect the cost function and optimization interval differ from, for example, constraints, since the user may specify zero, one or more of the latter.

One option for providing this kind of information is to introduce a built-in class, call it Optimization, and require that all optimization classes inherit from Optimization. Information about the cost function and optimization interval may then be given as modifications of components in this built-in class:

```modelica
extend Optimization

  objective=cost(finalTime),
  startTime=0,
  finalTime(free=true,initialGuess=1));

  Real cost;
  DoubleIntegrator di(u(free=true,
                        initialGuess=0.0));

  equation
der(cost) = 1;
end DimInTime;
```

Here, objective, startTime and finalTime are assumed to be components located in Optimization, whereas cost is a variable.
which is looked up in the scope of the optimization class itself. Notice also how the cost function, cost, has been introduced, and that the finalTime attribute is specified to be free in the optimization. This approach of inheriting from a built-in class has been used previously, in the tool Mosilab (Nytsch-Geusen, 2007), where the Modelica language is extended to support statecharts. In the statechart extension, a new specialized class, state, is introduced, and properties of a state class (for example whether the state is an initial state) can be specified by inheriting from the built-in class State and applying suitable modifications.

The main drawback of the above approach is its lack of clarity. In particular, it is not immediately clear that Optimization is a built-in class, and that its contained elements represent intrinsic properties of the optimization class, rather than regular elements, as in the case of inheritance from user or library classes.

To remedy this deficiency, the notion of class attributes is proposed. This idea is not new, but has been discussed previously within the Modelica community. A class attribute is an intrinsic element of a specialized class, and may be modified in a class declaration without the need to explicitly extend from a built-in class. In the Optimica extension, four class attributes are introduced for the specialized class optimization. These are objective, which defines the cost function, startTime, which defines the start of the optimization interval, finalTime, which defines the end of the optimization interval, and static, which indicates whether the class defines a static or dynamic optimization problem. The proposed syntax for class attributes is shown in the following optimization class:

```
optimization DMinTime (  
    objective=cost(finalTime),  
    startTime=0,  
    finalTime(true, initialGuess=1))
```

Real cost;
```
DoubleIntegrator du(free=true,  
    initialGuess=0.0));
```

```
equation  
der(cost) = 1;
```

```
end DMinTime;
```

3.4.5. Constraints

Constraints are similar to equations, and in fact, a path equality constraint is equivalent to a Modelica equation. But in addition, inequality constraints, as well as point equality and inequality constraints should be supported. It is therefore natural to have a separation between equations and constraints. In Modelica, initial equations, equations, and algorithms are specified in separate sections, within a class body. A reasonable alternative for specifying constraints is therefore to introduce a new kind of section, constraint. Constraint sections are only allowed inside an optimization class, and may contain equality, inequality as well as point constraints. In the double integrator example, there are several constraints. Apart from the constraints specifying bounds on the control input \( u \) and the velocity \( v \), there are also terminal constraints. The latter are conveniently expressed using the mechanism for accessing the value of a variable at a particular time instant; \( \text{di.x}(\text{finalTime})=1 \) and \( \text{di.v}(\text{finalTime})=0 \). In addition, bounds may have to be specified for the finalTime class attribute. The resulting optimization formulation may now be written:

```
optimization DMinTime (  
    objective=cost(finalTime),  
    startTime=0,  
    finalTime(true, initialGuess=1))
```

Real cost;
```
DoubleIntegrator du(free=true,  
    initialGuess=0.0));
```

```
equation  
der(cost) = 1;
```

```
constraint  
finalTime=0.5;  
finalTime=10;  
di.x(finalTime)=1;  
di.v(finalTime)=0;  
di.<=0.5;  
di.u=1;  
di.u<=1;
```

```
end DMinTime;
```

3.4.6. Annotations for specification of the transcription scheme

The transcription scheme used to transform the infinite-dimensional dynamic optimization problem into a finite-dimensional approximate problem usually influences the properties of the numerical solution. Nevertheless, transcription information can be considered to be complimentary information, that is not part of the mathematical definition of the optimization problem itself. Also, transcription information is closely related to particular numerical algorithms. It is therefore reasonable not to introduce new language constructs, but rather new annotations for specification of transcription schemes. This solution is also more flexible, which is important in order easily accommodate transcription schemes corresponding to algorithms other than the direct collocation method currently supported.

4. The JModelica.org platform

In order to demonstrate the feasibility and effectiveness of the proposed Optimica extension, a prototype compiler was developed (Åkesson, 2007). Currently, the initial prototype compiler is being developed with the objective of creating a Modelica-based open source platform focused on dynamic optimization.

The architecture of the JModelica.org platform is illustrated in Fig. 3. The platform consists essentially of two main parts: the compiler and the JModelica.org Model Interface (JMI) run-time library. The compiler translates Modelica and Optimica source code into a flat model description, then applies symbolic algorithms to transform the model into a hybrid DAE, and finally generates C and XML code. The generated C code contains the actual model equations in a format suitable for efficient evaluation, whereas the XML code contains model meta data, such as variable names and parameter values. The JMI run-time library provides a C interface which in turn can be interfaced with numerical algorithms. There is also an Eclipse plug-in and a Python integration module under development. In this section, the key parts of the JModelica.org platform will be described.

4.1. Compiler development—JastAdd

Compiler construction has traditionally been associated with intricate programming techniques within the area of computer science. Recent research effort has, however, resulted in new compiler construction frameworks that are easier to use and where it is feasible to develop compilers with a comparatively reasonable effort. One such framework is JastAdd (Ekman, Hedin, & Magnusson, 2006; Hedin & Magnusson, 2003). JastAdd is a Java-based compiler construction framework based on concepts such as object-orientation,
aspect-orientation and reference attributed grammars (Ekman & Hedin, 2004). At the core of JastAdd is an abstract syntax specification, which defines the structure of a computer program. Based on an abstract syntax tree (AST), the compiler performs tasks such as name analysis, i.e., finding declarations corresponding to identifiers, type analysis, i.e., verifying the type correctness of a program, and code generation.

The JastAdd way of building compilers involves specification of attributes and equations based on the abstract syntax specification. This feature is very similar to ordinary Knuth-style attribute grammars (Knuth, 1968) but enhanced with reference attributes. Accordingly, attributes may be used to specify, declaratively, links between different nodes in the AST. For example, identifier nodes can be bound to their declaration nodes. In Fig. 4, an example of a small Modelica program and its corresponding AST is shown. Notice how the reference attribute \texttt{myDecl} links an identifier (\texttt{IdUse}) to its declaration (\texttt{CompDecl}).

JastAdd attributes and equations are organized into separate aspects, which form a convenient abstraction for encoding of cross cutting behavior. Typically, implementation of a semantic function, for example name look-up, involves adding code to large number of classes in the AST specification. Using aspects, much like in AspectJ (Kiczales et al., 2001), cross cutting behavior can be modularized in a natural way. In addition, this approach is the basis for one of the distinguishing features of JastAdd: it enables development of modularly extensible compilers. This means that it is feasible to develop, with a comparatively moderate effort, modular extensions of an existing JastAdd compiler without changing the core compiler. This feature has been used in the implementation of the JModelica.org Modelica and Optimica compilers, where the Optimica compiler is a fully modular extension of the core Modelica compiler.

The JastAdd compiler transforms the JastAdd specification into pure Java code, where the definition of the abstract grammar translates into Java classes corresponding to Modelica classes, components, functions, and equations. The JastAdd attributes are woven into the Java classes as methods. In addition, methods for traversing an AST and query properties of a particular AST class, e.g., obtain a list of variables contained in a class declaration, are automatically generated. As a result of this approach, compilers produced by JastAdd are in the form of standard Java packages, which in turn can be integrated in other applications. It is therefore not necessary to know the particular details of how to write JastAdd specifications in order to use the JModelica.org compilers, knowledge of Java is generally sufficient.

4.2. The Modelica and Optimica compilers

At the core of the JModelica.org platform is a Modelica compiler that is capable of transforming Modelica code into a flat representation and of generating C code. In the Modelica compiler, several design strategies, for example name look-up, developed for a Java compiler developed using JastAdd (Ekman & Hedin, 2007), were reused. For additional details on the implementation of the compiler, see Åkesson, Ekman, and Hedin (2010). In order to support also the Optimica extension, an extended compiler capable of translating standard Modelica enhanced with the new Optimica syntax presented in Section 3 has also been developed.

The JModelica.org Modelica compiler currently supports a subset of Modelica version 3.0. The Modelica Standard Library version 3.0.1 can be parsed and the corresponding source AST can be constructed. Flattening support is more limited, but is being continuously improved.

4.3. Code generation

The JModelica.org offers a code generation framework implemented in Java as part of the compilers. The framework facilitates development of custom code generation modules and is based on templates and tags. A template is used to specify the structure of the generated code and tags are used to define elements of the template which is to be replaced by generated code. In order to develop a custom code generation module, the compiler developer needs to define a template and a set of tags, and then implement the
actual code generation behavior corresponding to each tag. In order to perform the latter, the AST for the flattened Modelica model is typically used, where objects corresponding to declarations, equations and functions are queried for information used to generate the target code.

The JModelica.org platform contains two code generation modules, one for C and one for XML. The generated C code contains the model equations and is intended to be compiled and linked with the JModelica.org Model Interface (see below) in order to offer efficient evaluation of the model equations. The XML export is described below in Section 4.5.

4.4. C API

The JModelica.org platform offers a C API, entitled the JModelica.org Model Interface (JMI), suitable for integration with numerical algorithms. The interface provides functions for accessing and setting parameter and state values, for evaluation of the DAE residual function and for evaluation of cost functions and constraints specified in an Optimica model. In addition, Jacobians and sparsity patterns can be obtained for all functions in the interface. To meet this end, a package for automatic differentiation, CppAD (Bell, 2008), has been integrated into JMI. Computation of derivatives may be done in several ways. One option is to use symbolic or automatic differentiation based on the ASTs in the compiler and then generate C code for the differentiated expressions. Another option is to export an XML model representation and then define an automatic differentiation scheme as an XSLT transformation (Bischof, Bucker, Marquardt, Petera, & Wyes, 2006; Elsheikh, 2008). The choice of CppAD was done in order to provide derivatives and sparsity patterns in a simple way in the context of the execution interface. The JMI code is intended to be compiled with the C code that is generated by the compiler into an executable, or into a shared object file.

The JMI interface consists of four parts: the ODE interface, the DAE interface, the DAE initialization interface, and the Optimization interface. These interfaces provide access to functions relevant for different parts of the optimization specification. The ODE and DAE interfaces provide evaluation functions for the right hand side of the ODE and the DAE residual function respectively. The DAE initialization problem provides functions for solving the DAE initialization problem, whereas the Optimization interface provides functions for evaluation of the cost functions and the constraints.

4.5. XML export

The XML output consists of model meta data such as specifications of variables, including their names, attributes and type. Also, the XML output includes a separate file for parameter values. The XML output is similar to what is discussed within the FMI initiative (DLR Dynasim, ITI, & QTronic, 2009), and the intention is for the JModelica.org XML output to be compliant with FMI once finalized. In addition, there is on-going work aimed to develop an XML specification for flattened Modelica models, including variable declarations, functions, and equations (Casella, Filippo, & Åkesson, 2009). This XML format is similar to CapeML (von Wedel, 2002), although geared towards representation of flat Modelica models. Providing XML export as an complement to the C model execution interface is motivated by the large body of general purpose tools for transforming XML into various formats. As noted above, an automatic differentiation scheme may be encoded as an XSLT transformation. Also, XSLT transformations may be used to encode code generation schemes.

5 Notice that this acronym is unrelated to Java Metadata Interface.

4.6. Interactive environment—Python

Solution of engineering problems typically involves atomization of tasks in the form of user scripts. Common examples are batch simulations, parameter sweeps, post-processing of simulation results and plotting. Given that JModelica.org is directed towards scientific computing, Python, see P. S. Foundation, is an attractive option. Python is a free open-source highly efficient and mature scripting language with strong support in the scientific community. Packages such as NumPy (Oliphant, 2009) and SciPy (Enthought, 2009), and bindings to state-of-the-art numerical codes implemented in C and Fortran make Python a convenient glue between JModelica.org and numerical algorithms. In addition, the IPython (Enthought, 2009) with the visualization package matplotlib (Hunter, Dale, & Droettboom, 2009 and the PyLab mode offer an interactive numerical environment similar to the ones offered by Matlab and Scilab.

The JModelica.org Python package includes sub packages for running the compilers, for managing file input/output of simulation/optimization results and for accessing the function provided by the JMI interface. The compilers are run in a Java Virtual Machine (JVM) which is connected to the Python environment by the package JPyve (Menard, 2009). One of JPyve’s main features is to enable direct access to Java objects from a Python shell or script. This feature is useful to communicate with the compilers, but can also be used to retrieve the ASTs generated by the compilers. The latter feature enables the user to traverse and query the ASTs interactively, to perform analyses and transformations of models.

The integration of the JMI is based on the ctypes package (P. S. Foundation, 2009). Using ctypes, a dynamically linked library (DLL) can be loaded into Python. All the contained functions of the DLL are then exposed and can be called directly from the Python shell. In order to enable use of NumPy arrays and matrices as arguments to the JMI functions, the argument types have been explicitly encoded using standard features of ctypes. In order to provide a more convenient interface to the JMI functions, a Python class, Model has been created. This class encapsulates loading of a DLL and typing of the JMI functions, and also provides wrapper functions supporting Python exceptions. In addition, upon creation of a Model class, the generated XML meta data files are loaded and parameter values and start attributes are set in the loaded model instance. Model objects can then be manipulated, e.g., by setting new parameter values, or passed as an argument to a simulation or optimization algorithm.

4.7. Optimization algorithms

The JModelica.org platform offers two different algorithms for solving dynamic optimization problems. The first is a simultaneous optimization method based on orthogonal collocation on finite elements (Biegler et al., 2002). Using this method, state and input profiles are parametrized by Lagrange polynomials which are based on Radau points. This method corresponds to a fully implicit Runge–Kutta method, and accordingly it possesses well known and strong stability properties. By parameterizing the variable profiles by polynomials, the dynamic optimization problem is translated into a non-linear programming (NLP) problem which may be solved by a numerical NLP solver. This NLP is, however, very large. In order to efficiently find a solution to the NLP, derivative information as well as the sparsity patterns of the constraint Jacobians need to be provided to the solver. The simultaneous optimization algorithm has been interfaced with the large-scale NLP solver Ipopt (Wächter & Biegler, 2006), which has been developed particularly to solve NLP problems arising in simultaneous dynamic optimization methods. The algorithm is implemented in C as an extension of JMI, and provides an example of how to implement algorithms based on the
5.1. Model predictive control of a CSTR system

We consider the Hicks–Ray Continuously Stirred Tank Reactor (CSTR) containing an exothermic reaction (Hicks & Ray, 1971). The states of the system are the reactor temperature \( T \) and the reactant concentration \( c \). The reactant inflow rate, \( F_0 \), concentration, \( c_0 \), and temperature, \( T_0 \), are assumed to be constant. The input of the system is the cooling flow temperature \( T_c \). The dynamics of the system is then given by:

\[
\dot{c}(t) = \frac{F_0(c_0 - c(t))}{V} - k_0 e^{-EdivR/T(t)} c(t)
\]

\[
\dot{T}(t) = \frac{F_0(T_0 - T(t))}{V} - \frac{dH}{\rho C_p} k_0 e^{-EdivR/T(t)} c(t) + \frac{2U}{\rho C_p} (T_c(t) - T(t))
\]

(12)

where \( r, k_0, EdivR, U, \rho, C_p, dH, V \) are physical parameters.

Based on the CSTR model, the following dynamic optimization problem is formulated:

\[
\min \int_{t_0}^{t_f} \left( (c^{ref} - c(t))^2 + (T^{ref} - T(t))^2 + (T_c^{ref} - T_c(t))^2 \right) dt
\]

subject to the dynamics (12). The cost function corresponds to a load change of the system and penalizes deviations from a desired operating point given by target values \( c^{ref}, T^{ref} \) and \( T_c^{ref} \) for \( c, T \) and \( T_c \), respectively. Stationary operating conditions were computed based on constant cooling temperatures \( T_c = 250 \) (initial conditions) and \( T_c = 280 \) (reference point). The stationary points were computed using an initialization algorithm available in JModelica.org.

The load change defined by (13) corresponds to ignition of the reactor, i.e., the initial conditions correspond to a cold reactor where little reaction takes place and the target set-point corresponds to production conditions. Due to the exothermic character of the reaction and the highly temperature dependent reaction kinetics, ignition of the reactor while avoiding temperature overshoot is a challenging control problem. An additional bound on the temperature was therefore added:

\[
T(t) \leq 350
\]

(14)

The Modelica and Optimica descriptions for the CSTR model and the optimal control problem, respectively, were translated into C code by the Optimica compiler and the resulting NLP was solved using IPOPT, see Section 4 for details on the collocation method implemented in JModelica.org. The optimal control profiles are shown in Fig. 5. As can be seen, the state profiles are converging to the reference set-point.

The optimal control problem formulated above can also be used in conjunction with other algorithms available in JModelica.org. To demonstrate this, a simple non-linear model predictive controller (NMPC) has been implemented. The NMPC control strategy is based on the receding horizon principle, where an open loop optimal control problem is solved in each sample. Simulation of an NMPC requires joint simulation of the plant and solution of the optimal control problem. Simulation is supported in JModelica.org by the SUNDIALS DAE integrator, see Section 4. The algorithm for executing the NMPC and the simulation algorithm were implemented in a Python script.

In order to avoid numerical difficulties resulting from infeasible initial conditions in operating conditions where the temperature bound is active, the hard constraint was complemented by a soft constraint penalty term:

\[
\gamma(t) = \begin{cases} 345 & \text{if } T(t) \leq 345 \\ 0 & \text{else} \end{cases}
\]

(15)
which gives the augmented cost function:

$$\min_{T_c(t)} \int_0^{t_f} \left( (c_{\text{ref}} - c(t))^2 + (T_{\text{ref}} - T(t))^2 + (T_c_{\text{ref}} - T_c(t))^2 + \frac{1000}{CR(t)} \right) dt. \quad (16)$$

Notice that the term $\frac{1}{CR(t)}$ does not introduce any discontinuity in the cost function. Also, the temperature bound was kept in the optimization formulation. The result of executing the NMPC is shown in Fig. 6. The prediction horizon was set to 50 s and the sampling interval was assumed to be 2 s.

### 5.2. Distillation column load change

As a second example, we consider a distillation column previously reported in Benallou, Seborg, and Mellichamp (1986) and Hahn and Edgar (2002). The distillation column consists of 30 trays and separates a binary mixture containing components A and B. The relative volatility is assumed to have a constant value of 1.6. The feed stream, $x_F$, is introduced at the middle of the column, in tray 17, and has a constant composition of 0.5. The model has 32 states corresponding to the liquid compositions of component A, $x_i(t)$. The vapor compositions of component A are denoted $y_i(t)$. Apart from the 30 trays, volumes for the condenser and the reboiler, respectively, are included in the model. The controlled input of the model is the reflux ratio, $u(t)$.

A load change problem transferring the state of the distillation between two stationary operating points is considered. The operating points correspond, respectively to reflux ratios of 3.0 and 2.0. This in turn gives stationary values of the distillate and bottom purities of $y_1 = 0.96$, $y_{32} = 0.099$ and $y_1 = 0.90$, $y_{32} = 0.23$ respectively. The control objective of the optimal control problem is defined:

$$\min_{u(t)} \int_0^{50} \left( \alpha (0.90 - y_1(t))^2 + (2.0 - u(t))^2 \right) dt \quad (17)$$

where $\alpha = 1000$ is a constant weight. This objective penalizes deviations from the target operating point in the distillate vapor composition of component A and in the reflux ratio. The reflux ratio term is included in the cost function in order to guarantee a unique solution of the problem. In practice, the target values of the vapor composition and the reflux ratio are typically computed by a plant-wide coordinating optimization layer considering economic objectives. In addition, the reflux ratio is assumed to be constrained in the optimization by $1.0 \leq u(t) \leq 5.0$.

The dynamics of the system was discretized on a mesh consisting of 50 elements by means of collocation based on cubic Lagrange
Acknowledgements

This paper reports an extension of the Modelica modeling language, entitled Optimica, targeting dynamic optimization. Optimica extends the Modelica language to also include constructs needed when formulating optimization problems, including specification of optimization interval, the cost function, and the constraints.

The Optimica language is supported by the novel Modelica-based open source project JModelica.org, featuring code generation to C, a C API for evaluation of model equations and a Python integration. JModelica.org offers an accessible platform for modeling, optimization and simulation of complex physical systems, which are often encountered in process control applications. One of the main objectives of the JModelica.org project is to facilitate technology transfer between the academic community and industry by providing a platform suitable for experimental algorithm design in an environment where relevant industrial applications are easily accessible.

The capabilities of Optimica and JModelica.org have been demonstrated by means of two examples. Firstly, a CSTR system was controlled by an NMPC controller. This example demonstrates the flexibility provided by the Python scripting environment, where in this case, simulation and optimization tasks were executed jointly. In a second example, an optimal control problem for a distillation column was solved. This example demonstrates the ability of JModelica.org to solve large-scale problems.

Experiences from using Optimica and JModelica.org in a number of projects in different application domains indicate that Modelica/Optimica specifications are perceived as compact and intuitive. Also, students and engineers with a control background but lacking expertise in dynamic optimization are typically able to quickly set up and solve realistic problems. In this respect, the use of Python serves as a convenient environment for development of prototype applications involving dynamic optimization.

While the current version of JModelica.org features two particular methods for solving dynamic optimization problems, one multiple shooting method and one based on collocation, it is expected that additional algorithms will be interfaced with the platform. This will further extend the scope of the JModelica.org and increase its applicability.

References


