Optimica—An Extension of Modelica Supporting Dynamic Optimization

Åkesson, Johan

2008

Optimica—An Extension of Modelica Supporting Dynamic Optimization

Johan Åkesson
Department of Automatic Control
Faculty of Engineering
Lund University
BOX 118, SE-221 00 Lund

Abstract

In this paper, an extension of Modelica, entitled Optimica, is presented. Optimica extends Modelica with language constructs that enable formulation of dynamic optimization problems based on Modelica models. There are several important design problems that can be addressed by means of dynamic optimization, in a wide range of domains. Examples include, minimum-time problems, parameter estimation problems, and on-line optimization control strategies. The Optimica extension is supported by a prototype compiler, the Optimica compiler, which has been used successfully in case studies.
Keywords: Optimica, Language extension, Dynamic optimization, The JModelica compiler

1 Introduction

Modelica is becoming a standard format for describing and communicating high-fidelity models of large-scale dynamic systems. Expert knowledge is being encoded into Modelica libraries, both in industry and in academia. The growing body of Modelica models also represents significant capital investments, and accordingly, Modelica models and libraries represent valuable assets for many companies. As a consequence, Modelica models are turning into legacy code, which cannot easily be replaced, simply because the cost of re-encoding the models in a different format is too large.

While the primary usage of Modelica models today is simulation, several other usages are emerging. Since it is not feasible, for the reasons mentioned above, to re-encode models for each new model usage, future Modelica tools, and also the Modelica language itself, should accommodate and promote new usages of Modelica models. This requirement has profound consequences for software design of Modelica tools, and also for the language design itself. In particular, some new usages may require new constructs, at the language level, in order to enable modeling of particular design problems.

One example of an emerging usage of Modelica models is dynamic optimization. A characteristic feature of realistic dynamic optimization problems is that the procedure of formulating such problems is highly iterative. It is common that extensive tuning of the cost function and constraints is required in order to obtain an acceptable solution. If a numerical algorithm is used to solve the dynamic optimization problem, there is an additional dimension that requires attention: the design of the transcription scheme. The scheme used to discretize the control and state variables often strongly influences the properties of the resulting solution. The choice of discretization method also affects the execution time for solving the problem, which is an important aspect in on-line applications. For these reasons, dynamic optimization problems are very rich in the sense that there are several aspects that require attention. Also, the user needs, and should be enabled to, model, using high-level language constructs, the optimization problem both in terms of cost functions and constraints and at the transcription level.

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[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.

There are several software packages supporting dynamic optimization, for example Dymola [6] and gPROMS [12]. However, most available software tools are restricted in the sense that they usually only support a particular optimization algorithm. While a particular algorithm may work well in some cases, the appropriate choice of numerical algorithm is usually dependent on the particular problem at hand. An analogy with differential equation solvers can be made. Stiff systems call for sophisticated, but potentially computationally demanding solvers, whereas less difficult systems may be more efficiently solved by a simpler algorithm. An additional goal in the development of tools supporting high-level formulation of dynamic optimization problems is therefore to provide an open architecture, where several different algorithms can be integrated.

In this paper, an extension of Modelica, entitled Optimica, will be presented. Optimica consists of a number of new language elements, which enable high-level formulation of dynamic optimization problems based on Modelica models. The syntax as well as the semantics of Optimica will be described. In addition, a prototype implementation of an Optimica compiler, which is a modular extension of the JModelica compiler [2, 1], will be presented.

The paper is organized as follows. In Section 2, issues related to extensions of languages are discussed. Different options regarding language extensions in Modelica are also treated. In Section 3, the scope of Optimica is discussed, i.e., the class of optimization problems that can be expressed using Optimica is defined. In Section 4 the syntax and the semantics of the Optimica extension are presented. Implementation issues related to the modular Optimica extension of the JModelica compiler are discussed in Section 5. The paper ends with a summary and conclusions in Section 6.

[1]The term low-level is relative, but is here used in relation to domain-specific languages like Modelica.

2 Motivation of the Optimica Extension

2.1 Isn't Modelica Enough?

Although being a very rich language in terms of expressive power for describing complex hybrid dynamical systems, Modelica lacks important features desirable for expressing optimization problems. This is quite natural, since Modelica was not developed with optimization in mind. For example, the notion of cost functions, constraints, variable bounds and initial guesses are not included in the Modelica language. Some of these quantities may indeed be modeled using standard Modelica, to some extent. For example, a particular variable may be given the meaning of cost, and the `min` and `max` attributes may be interpreted as variable bounds. However, while this approach may work in simple cases, it becomes intractable for more complex optimization problems. For example, complicated constraints, several use cases, and tailoring of the transcription method would be difficult to express. Further, the `min` and `max` attributes are usually used to express regions of validity for a model, and giving them a new semantic meaning would be potentially misleading.

2.2 What About Annotations?

Modelica offers a mechanism for adding information to model, which may not be part of the actual mathematical description, but which is convenient to store in the model. Typical examples include graphical annotations and documentation. Annotations can also be used to supply information that can be used by a particular tool, for example, in order to influence properties of the translation process. In principle, it would be possible to specify parts of an optimization problem by introducing suitable annotations. For example, a variable could be marked as a cost function, and the semantic meaning of the equality operator in an equation could be changed to that of the inequality operator. There are two reasons why it is not a good idea to strictly use this approach. Firstly, and most importantly, annotations are designed to supply *complementary* information, whereas in this case, the elements of an optimization problem are rather *primary* information, that is essential for solving the actual problem. Also, since annotations are not intended for formulation of design problems, they do not provide a convenient modeling environment for the user. Secondly, annotations cannot currently be changed by means of

modifications. Since modification is one of the cornerstones of Modelica, this is a severe restriction. Also, it is not currently well defined how annotations are treated in the case of inheritance. Since one of the main objectives of the Optimica extension is to enable convenient formulation of dynamic optimization problems using high-level constructs, using only annotations does not seem to be a feasible alternative.

Whereas the above arguments are applicable to core elements of an optimization problem, such as cost function and constraints, annotations may well be used to specify a *solution algorithm*, and associated parameters. This type of information is not part of the actual optimization formulation, but it might still be essential in order to efficiently solve the problem numerically. By introducing annotations for specifying, for example, the collocation scheme used in a direct method, the user is able to model both the actual optimization problem at hand and the transcription method in a unified high-level description language. This approach is also in line with the intentions of Modelica annotations, because of the separation between formulation of the actual problem (by means of dedicated language constructs), and specification of the solution technique (by means of annotations).

2.3 Tool-oriented Support for Optimization?

Another potential strategy for enabling dynamic optimization of Modelica models is to develop tool-oriented solutions, for example Graphical User Interfaces (GUIs), within a simulation-based software tool. This approach is used, for example, to enable optimization of Modelica models in Dymola. The user would then set up the optimization problem by entering information in dedicated fields in the GUI. Using this approach, the software tool needs to maintain an internal model of the optimization problem, as specified by the user. While this solution may be an attractive choice for interfacing a particular optimization method with existing simulation-based tools, it does not offer the flexibility, or portability, which is inherent in the Modelica language. It is therefore desirable to define, at the language level, a generic extension, which has a well defined syntax and semantics. Nevertheless, it may still be desirable to offer GUIs, in order to increase productivity in the design process, in the same way as current Modelica tools typically offer GUIs to simplify critical modeling tasks.

2.4 To Extend or to Complement?

A key issue is whether to extend Modelica by introducing new language constructs, or to define a new, separate, language which complements Modelica. By introducing a new language, the syntax and semantics of Modelica would be kept entirely intact, which may be advantageous since it makes design and maintenance of the language simpler. Also, if several extensions are introduced, defining the interaction between the extensions, both at a syntactic and semantic level, may be difficult. On the other hand, Modelica has many generic built-in constructs, e.g., classes, functions and declarative equations, which are widely applicable in many contexts. Reinventing such constructs in new languages does not seem to be an attractive alternative. Another argument in favor of language extension is that Modelica offers strong support for modularization of models. In the case of dynamic optimization, the user may construct the model separately from the formulation of the optimization problem, in which the model is used. In this way, the same model may still be used for other purposes than optimization, such as, for example, simulation.
It is essential, however, that language extensions targeted at particular usages of Modelica models do not interfere unnecessarily with the original language. Preferably, extensions should be *modular*, in the sense that the new constructs are only allowed in a well defined language environment.

3 Scope of 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 [3]. 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:

$$\min_{u(t),p} \psi(\bar{z},p) \tag{1}$$

subject to the dynamic system

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

and the constraints

$$c_{ineq}(x(t),y(t),u(t),p) \leq 0 \quad t \in [t_0,t_f] \tag{3}$$
$$c_{eq}(x(t),y(t),u(t),p) = 0 \quad t \in [t_0,t_f] \tag{4}$$
$$c^p_{ineq}(\bar{z},p) \leq 0 \tag{5}$$
$$c^p_{eq}(\bar{z},p) = 0 \tag{6}$$

where $x(t) \in R^{n_x}$ are the dynamic variables, $y(t) \in R^{n_y}$ are the algebraic variables, $u(t) \in R^{n_u}$ are the control inputs, and $p \in R^{n_p}$ 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. In addition, 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; $\bar{z} = [x(t_1),..,x(t_{N_p}),y(t_1),..,y(t_{N_p}),u(t_1),..,u(t_{N_p})]^T, \quad t_i \in [t_0,t_f]$, where N_p denotes the number of time points included in the optimization problem.

The constraints include inequality and equality path constraints, (3)-(4). In addition, inequality and equality point constraints, (5)-(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, $x_L(t)$, with the associated differential equation $\dot{x}_L(t) = L(x(t),u(t))$, and the cost function $\psi(t_f) = x_L(t_f)$. 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, $i \in 1 \ldots N_d$ is typically available. A cost function candidate is then:

$$\sum_{i=1}^{N_d} (y(t_i) - y_d(t_i))^T W (y(t_i) - y_d(t_i)) \tag{7}$$

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:

$$\begin{aligned} &\min_{u,p} \varphi(x,y,u,p) \\ &\text{subject to} \\ &F(0,x,y,u,p,t_s) = 0 \\ &c_{ineq}(x,u,p) \leq 0 \\ &c_{eq}(x,u,p) = 0 \end{aligned} \tag{8}$$

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_s denotes the time instant at which the static optimization problem is defined.

3.4 Transcription

In this paper a direct collocation method (see for example [4]) will be used to illustrate how also the transcription step can 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.

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:

$$\min_{u(t)} \int_0^{t_f} 1 \, dt \tag{9}$$

subject to the dynamic constraint

$$\begin{aligned} \dot{x}(t) &= v(t), \quad x(0) = 0 \\ \dot{v}(t) &= u(t), \quad v(0) = 0 \end{aligned} \tag{10}$$

and

$$\begin{aligned} x(t_f) &= 1, \quad v(t_f) = 0 \\ v(t) &\leq 0.5, \quad -1 \leq u(t) \leq 1 \end{aligned} \tag{11}$$

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 $u(t)$. A Modelica model for the double integrator system is shown in Listing 1.

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

```
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.

- 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`, `finalTime` and `static`
- A new section: `constraint`
- Inequality constraints
- An annotation for providing transcription information

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 problem (9)-(11), consider the `optimization` class:

```
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.

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[2]. 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 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.

[2]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

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.

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

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 argument 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.

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:

```
optimization DIMinTime
  extends 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 [11], 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 inherit-

ing 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 DIMinTime (
         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;
```

The default value of the class attribute `static` is `false`, and accordingly, it does not have to be set in this case. In essence, the keyword `extends` and the reference to the built-in class have been eliminated, and the modification construct is instead given directly after the name of the class itself. The class attributes may be accessed and modified in the same way as if they were inherited.

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; `di.x(finalTime)=1` and `di.v(finalTime)=0`. In addition, bounds may have to be specified for the `finalTime` class attribute. The resulting optimization formulation may now be written:

```
optimization DIMinTime (
     objective=cost(finalTime),
     startTime=0,
     finalTime(free=true,initialGuess=1))
  Real cost;
  DoubleIntegrator di(u(free=true,
                        initialGuess=0.0));
equation
  der(cost) = 1;
constraint
  finalTime>=0.5;
  finalTime<=10;
  di.x(finalTime)=1;
  di.v(finalTime)=0;
  di.v<=0.5;
  di.u>=-1; di.u<=1;
end DIMinTime;
```

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.

Following the guidelines for vendor-specific annota-

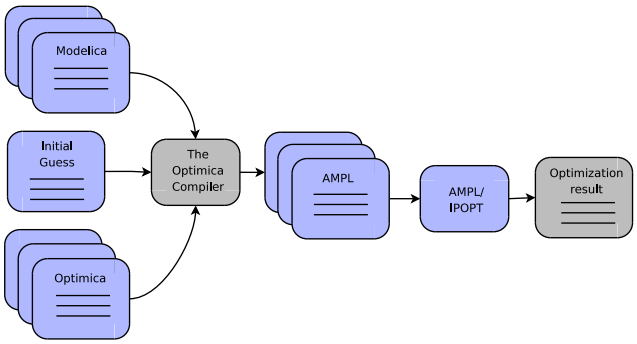

Figure 1: The transformation from Modelica/Optimica code to optimization result.

tions in the specification of Modelica 3.0 [13, p. 147], a hierarchical annotation for supplying the information needed to specify a direct collocation method based on interpolation polynomials has been introduced. This annotation is defined by the following Modelica `record`:

```
record DirectCollocationInterpolationPolynomials
  parameter Real mesh[:];
  parameter Real collocationPoints[:];
  parameter Real
            polynomialCoefficientsAlgebraic[:];
  parameter Real
            polynomialCoefficientsDynamic[:];
end DirectCollocationInterpolationPolynomials;
```

This annotation enables the user to influence the particular properties of the corresponding transcription scheme. For additional details, see [1].

5 The Optimica Compiler

A new Modelica compiler, entitled the JModelica compiler is currently under development [2, 1]. The compiler is developed in the compiler construction framework JastAdd, see [9], and in Java. One of the primary targets of the JModelica compiler is to provide an extensible compiler, which is suitable for modular implementation of new language features. A fundamental design concept is that of *modular extensibility*, which enables the core JModelica compiler to be kept intact, since new extensions may be implemented fully modularized.

A prototype implementation of the JModelica compiler, that also supports the Optimica extension has been developed. The extended compiler will be referred to as the Optimica compiler in the following. In terms of the front-end, the compiler supports a subset of Modelica, and an early version of Optimica. The syntax of Optimica that is supported by this compiler is different than the one presented in this paper, although the functionality is essentially the same. The new, improved syntax and semantics that have been presented in this paper, were defined based on the comments and experiences from the users of the very first version of Optimica. A new version of the Optimica compiler, supporting the revised Optimica syntax is currently under development, with the intention of replacing the initial prototype.

5.1 Code Generation to AMPL

One of the main features of the Optimica compiler is that it performs automatic transcription of continuous variables, using a direct collocation method. The user is thus relieved from the burden of encoding the collocation equations, which is a tedious and error-prone procedure. Whereas the prototype version of the Optimica compiler supported one particular collocation scheme, future versions will support the annotation introduced above to specify the transcription method.

In order to solve the transcribed optimization problem by means of a numerical algorithm, the Optimica compiler generates AMPL [7] code. The transcribed problem is purely static, and can therefore be encoded using the constructs available in AMPL. The AMPL representation of the optimization problem can be viewed as an additional intermediate representation format. The purpose of using AMPL is twofold. Firstly, AMPL provides an additional debugging level, that is very useful during compiler development. In particular, the AMPL tool offers a shell, where variables and constraints can be inspected. Secondly, the AMPL solver interface provides solvers with sparsity information, as well as first and second order derivatives. This information may be essential for performance and convergence of a numerical optimization algorithm. The numerical algorithm IPOPT [14] has been used to solve the non-linear program resulting from the transcription procedure. The result is then written to file for further analysis or implementation. See Figure 1 for an illustration of the transformation steps involved when using the Optimica compiler and AMPL to solve a dynamic optimization problem.

6 Summary and Conclusions

In this paper an extension of the Modelica language, Optimica, that enables high-level formulation of dynamic optimization problems, has been presented. The

Optimica extension enables the user to specify important elements of a dynamic optimization problem such as cost functions, constraints and optimization interval. The dynamic model, upon which the dynamic optimization problem is based, is expressed using standard Modelica. Optimica also supports an annotation that enables the user to specify the properties of a transcription method, based on direct collocation. Because of these properties, Optimica supports formulation of dynamic optimization problems, using high-level constructs, both at the mathematical level and at the numerical transcription level.

A prototype implementation of the Optimica compiler has been used in the work on start-up optimization of a plate reactor [8], in two master's thesis projects (see [5] and [10]) and in the PhD course "Optimization-Based Methods and Tools in Control", that was given at the Department of Automatic Control, Lund University in September 2007.

An important objective of the JModelica compiler is to offer a modularly extensible Modelica compiler. In this respect, the experiences and results from developing the Optimica extension are very promising. In particular, the coding effort needed to implement the extension of the compiler front-end, including extension of the name analysis framework and the flattening algorithm, was very moderate.

References

[1] J. Åkesson. Tools and Languages for Optimization of Large-Scale Systems. PhD thesis ISRN LUTFD2/TFRT--1081--SE, Department of Automatic Control, Lund University, Sweden, November 2007.

[2] J. Åkesson, T. Ekman, and G. Hedin. "Development of a Modelica compiler using JastAdd." In Seventh Workshop on Language Descriptions, Tools and Applications, Braga, Portugal, March 2007.

[3] P. Barton and C. K. Lee. "Modeling, simulation, sensitivity analysis, and optimization of hybrid systems." ACM Transactions on Modeling and Computer Simulation, **12:4**, 2002.

[4] L. Biegler, A. Cervantes, and A. Wächter. "Advances in simultaneous strategies for dynamic optimization." Chemical Engineering Science, **57**, pp. 575–593, 2002.

[5] H. Danielsson. "Vehicle path optimisation." Master's Thesis ISRN LUTFD2/TFRT--5797--SE, Department of Automatic Control, Lund University, Sweden, June 2007.

[6] Dynasim AB. "Dynasim AB Home Page." 2007. `http://www.dynasim.se`.

[7] R. Fourer, D. Gay, and B. Kernighan. AMPL – A Modeling Language for Mathematical Programming. Brooks/Cole — Thomson Learning, 2003.

[8] S. Haugwitz, J. Åkesson, and P. 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.

[9] G. Hedin and E. Magnusson. "JastAdd: an aspect-oriented compiler construction system." Science of Computer Programming, **47:1**, pp. 37–58, 2003.

[10] H. Hultgren and H. Jonasson. "Automatic calibration of vehicle models." Master's Thesis ISRN LUTFD2/TFRT--5794--SE, Department of Automatic Control, Lund University, Sweden, June 2007.

[11] C. Nytsch-Geusen. "MosiLab Home Page." 2007. `http://www.mosilab.de/`.

[12] Process Systems Enterprise. "gPROMS Home Page." 2007. `http://www.psenterprise.com/gproms/index.html`.

[13] The Modelica Association. "Modelica – a unified object-oriented language for physical systems modeling, language specification, version 3.0." Technical Report, Modelica Association, 2007.

[14] A. Wächter and L. T. Biegler. "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming." Mathematical Programming, **106:1**, pp. 25–58, 2006.