D5.1.1 - Extension of Modelica with language elements to describe optimization problems, and interfaces to execute NMPC in a virtual environment

WP 5.1 Nonlinear Model Predictive Control
WP 5 Optimized system operation

MODRIO (11004)

Version 3.0

Date 30/10/2015

Authors

Andreas Pfeiffer, Martin Otter	DLR
Vitalij Ruge, Bernhard Bachmann	FH Bielefeld
Toivo Henngingsson, Johan Åkesson	Modelon
Hilding Elmqvist	DS AB

Executive summary

This deliverable investigates different approaches to standardize the formulation of optimization problems in Modelica, and puts forward an approach based on custom annotations in Modelica. Standardized optimization formulations are the basis for tool development in the other tasks of WP 5.

In the first part of the document the current state of the art concerning optimization with Modelica is shown. The report describes some optimization environments using Modelica models to compute criteria values for optimization algorithms. Then, various standardization possibilities are discussed.

The new concept of using custom Modelica annotations to define optimization problems is introduced in detail and explained by examples. This approach is suggested to be standardized within the Modelica Association. The Modelica library "Optimization_CustomAnnotation.mo" that supports users in modeling optimization problems by custom annotations is delivered with this document. In D5.2.3 a prototype implementation of reading the custom annotations in Dymola is presented, including a running optimization.

Contents

1. State of the art – Optimization Library in Dymola

Since 2008 an Optimization Library for Modelica has been developed in Dymola by DLR [1]. The library consists of different Optimization tasks that can be configured by the user and that can be solved numerically. In this section the underlying mathematical problem description is shown, a typical optimization setup formulation in standard Modelica is explained and a graphical representation of the optimization setup for user interaction is illustrated.

1.1. Mathematical Optimization problem

A multi-criteria optimization problem considered in the Optimization library can be formulated as follows:

$$\min_{p \in B} z\left(\mathrm{diag}(d_1)^{-1} c_1(p)\right)$$

$$\text{such that} \quad c_2(p) \le d_2, \quad c_3(p) = d_3$$

$$\text{with} \quad c = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix}, \quad d = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix} \quad \text{and}$$

$$z = \begin{cases} \max & \text{... Maximum of criteria values, or} \\ \|\cdot\|_2^2 & \text{... Sum of squared criteria values, or} \\ \|\cdot\|_1 & \text{... Sum of absolute criteria values.} \end{cases}$$

We use the following abbreviations:

Name	Identifier	Description
Tuner parameters	p	Tuner parameters are the free variables (e.g. some Modelica parameters in models) to be varied during the optimization process.
Criteria	c	The first part of the criteria vector represents the objectives of the optimization (e.g. the "overshoot" of a variable in a model). The goal is to minimize all these objectives. The criteria components that define inequality or equality constraints are optional and need not to be present. They enable to formulate conditions on some criteria components if needed.
Demand vector	d	The demand values serve as reciprocal scaling factors of the criteria. They enable a different weighting of the individual criteria to be minimized.
Tuner Box	B	The Tuner Box B is defined by minimum and maximum values for each tuner parameter and restricts the domain in which the tuner parameters can be varied.
Objective function	z	The objective function combines the criteria components to be minimized to one scalar function to be minimized by the optimization algorithm. One can choose between the maximum of the criteria, the sum of the squares of the criteria or the sum of the absolute values of the criteria.

Typically, the criteria (including the left hand sides of inequality and equality constraints) result from a Modelica model that has to be integrated numerically from t_0 to t_e:

$$\dot{x} = f(x, p, t), \qquad x(t_0) = x_0(p),$$
$$y = h(x, p, t)$$

with an ordinary differential equation (ODE) and its right hand side f, states x, tuner parameters p, initial values x_0, algebraic output variables y and the output function h.

From the numerical solution of this system of equations, the criteria are computed by $c(p) := y(p, t_e)$, i.e. the values of the output variables at the final integration time. The criteria functions are assumed to be as smooth with respect to the parameters p as necessary for applying different numerical optimization algorithms.

1.2. Optimization problem description in Modelica

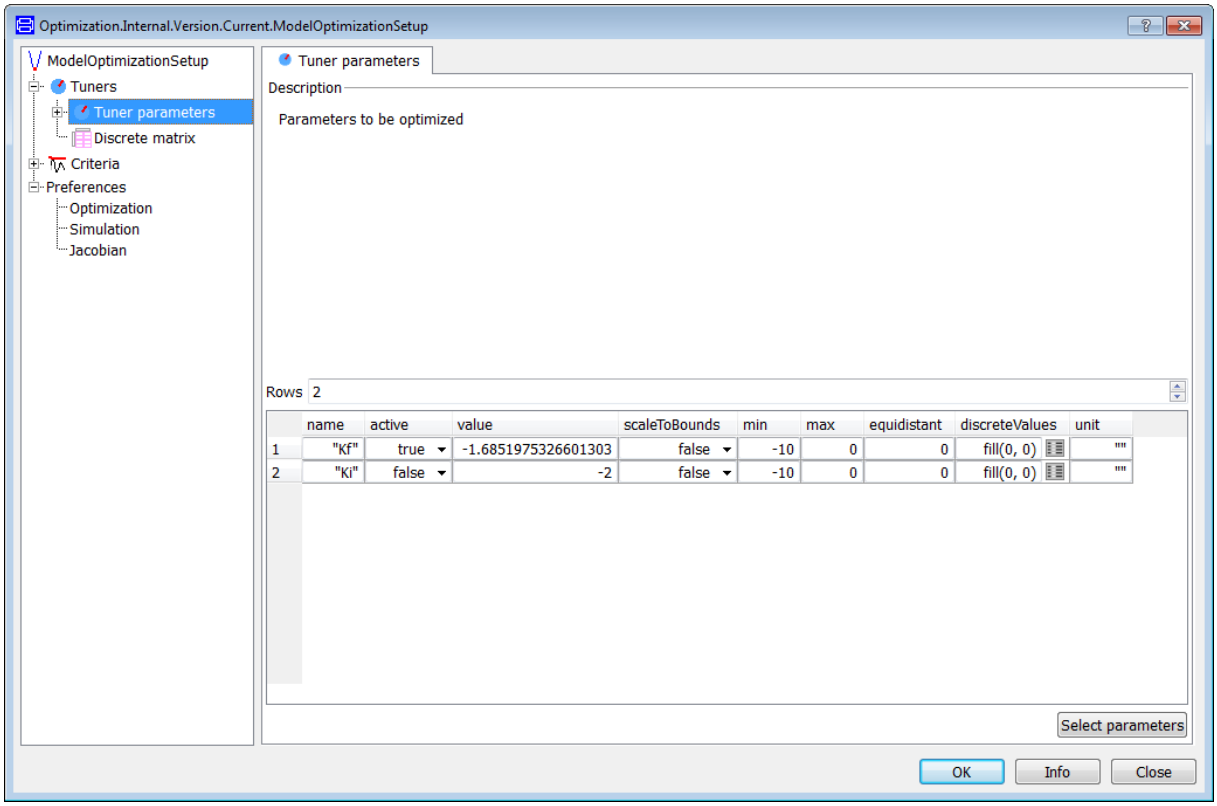
In the following a complete optimization setup for a model optimization task is listed. This Modelica record fully describes the mathematical optimization problem by implicitly defining the tuner parameters p, the objective function z, the demand values d and the criteria c. The tuner parameters are defined by the names of model parameters (here: *Kf* and *Ki*), the criteria are defined by names of model output variables (here: *overshoot*, *maxVelocity*, *riseTime* and *settlingTime*). The model to be numerically integrated to compute the criteria is defined by its name, the settings of the numerical integration are defined in the `simulationOptions` under `preferences`. Some further settings are defined that have proved to positively influence the numerical optimization or to inform the user about the status during the optimization process. Because the setup completely defines an optimization problem, it is also used as format to save the optimization setup information.

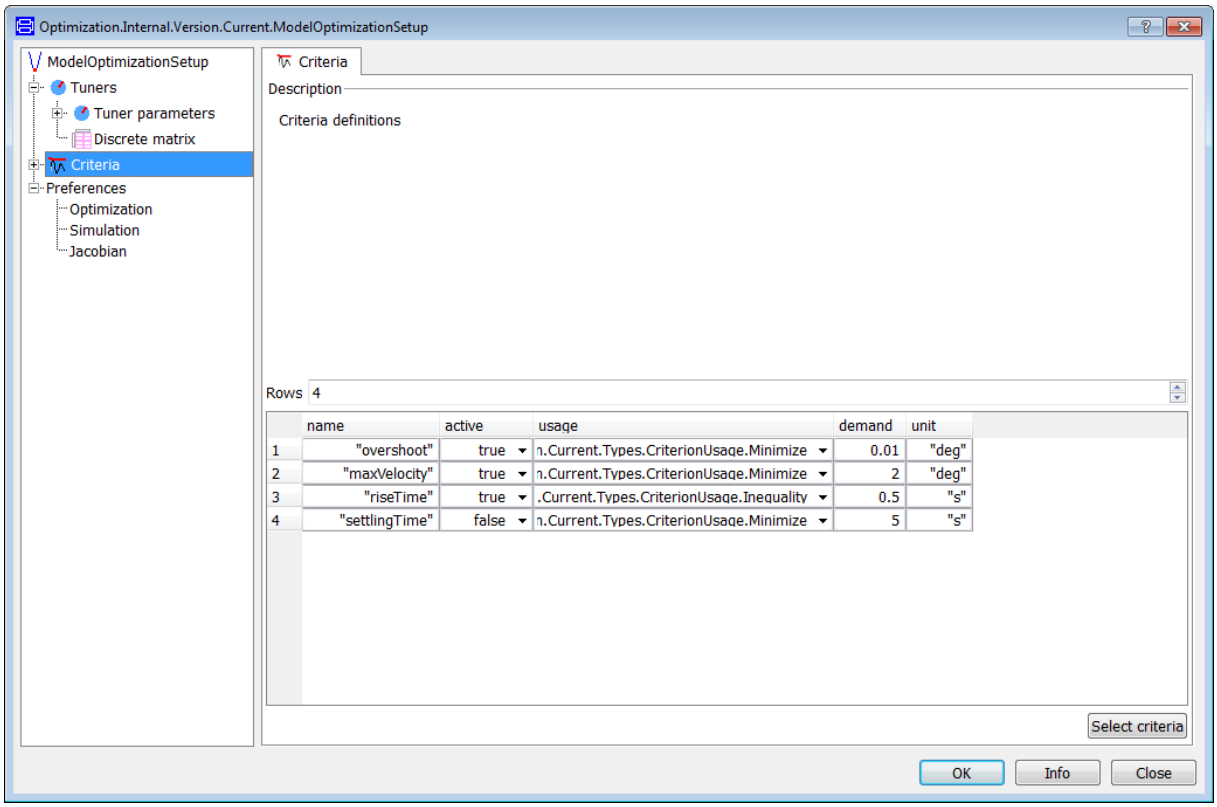
```
ModelOptimizationSetup(
  modelName="ControllerDesign",
  plotScript="",
  saveSetup=true,
  saveSetupFilename="OptimizationLastRunModel.mo",
  convertSetup=false,
  askForTunerReUse=true,
  tuner=Tuner(
    tunerParameters=
    {
       TunerParameter(
         name="Kf",
         active=true,
         Value=-1.6851975326601303,
         min=-10,
         max=0,
         unit=""),
       TunerParameter(
         name="Ki",
         active=false,
         Value=-2,
         min=-10,
         max=0,
         unit="")
    },
  ),
```

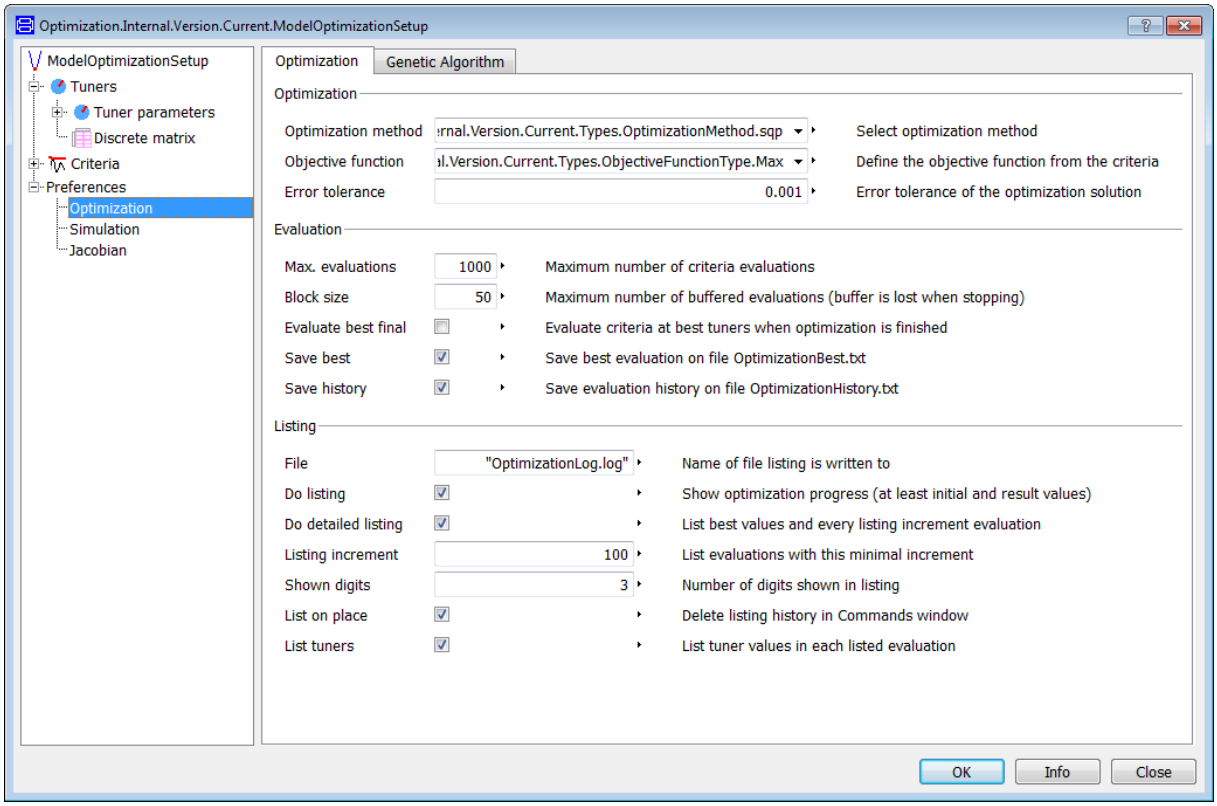
```
  criteria=
  {
    Criterion(
       name="overshoot",
       active=true,
       usage=Types.CriterionUsage.Minimize,
       demand=1e-2,
       unit="deg"),
    Criterion(
       name="maxVelocity",
       active=true,
       usage=Types.CriterionUsage.Minimize,
       demand=2,
       unit="deg"),
     Criterion(
       name="riseTime",
       active=true,
       usage=Types.CriterionUsage.Inequality,
       demand=0.5,
       unit="s"),
     Criterion(
       name="settlingTime",
       active=false,
       usage=Types.CriterionUsage.Minimize,
       demand=5,
       unit="s")
   },
   preferences=Preferences(
     optimizationOptions=OptimizationOptions(
       method=Types.OptimizationMethod.sqp,
       ObjectiveFunctionType=Types.ObjectiveFunctionType.Max,
       OptTol=1e-3,
       ...),
     simulationOptions=SimulationOptions(
       startTime=0,
       stopTime=10,
       integrationMethod=Types.IntegrationMethod.Dassl,
       integrationTolerance=1e-7,
       ...),
     sensitivityOptions=SensitivityOptions(
       TypeOfSensitivityComputation=
                     Types.SensitivityMethod.ExternalDifferencesSymmetric,
       ...))
)
```

1.3. Graphical User Interface in Dymola

For user convenience the optimization setup information shown in Section 1.2 is displayed by a graphical user interface (GUI) in Dymola. This GUI is only based on the record and type definitions and some annotations in them (not shown in Section 1.2 for simplification). The user can edit each of the record elements by browsing through a tree browser on the left side of the GUI. In the following the GUI is shown for different selections in the tree browser. The content that can be edited is exactly the same as for the text-based example in Section 1.2. If the user has configured the setup the numerical optimization process is started by pressing the OK-button.


Optimization.Internal.Version.Current.ModelOptimizationSetup
ModelOptimizationSetup
Tuners
Tuner parameters
Discrete matrix
Criteria
Preferences
Optimization
Simulation
Jacobian
Tuner parameters
Description
Parameters to be optimized
Rows 2
name active value scaleToBounds min max equidistant discreteValues unit
1 "Kf" true -1.6851975326601303 false -10 0 0 fill(0, 0) ""
2 "Ki" false -2 false -10 0 0 fill(0, 0) ""
Select parameters
OK
Info
Close


Optimization.Internal.Version.Current.ModelOptimizationSetup
ModelOptimizationSetup
Tuners
Tuner parameters
Discrete matrix
Criteria
Preferences
Optimization
Simulation
Jacobian
Criteria
Description
Criteria definitions
Rows 4
name active usage demand unit
1 "overshoot" true n.Current.Types.CriterionUsage.Minimize 0.01 "deg"
2 "maxVelocity" true n.Current.Types.CriterionUsage.Minimize 2 "deg"
3 "riseTime" true .Current.Types.CriterionUsage.Inequality 0.5 "s"
4 "settlingTime" false n.Current.Types.CriterionUsage.Minimize 5 "s"
Select criteria
OK
Info
Close


Optimization.Internal.Version.Current.ModelOptimizationSetup
ModelOptimizationSetup
Tuners
Tuner parameters
Discrete matrix
Criteria
Preferences
Optimization
Simulation
Jacobian
Optimization
Genetic Algorithm
Optimization
Optimization method
rnal.Version.Current.Types.OptimizationMethod.sqp
Select optimization method
Objective function
l.Version.Current.Types.ObjectiveFunctionType.Max
Define the objective function from the criteria
Error tolerance
0.001
Error tolerance of the optimization solution
Evaluation
Max. evaluations
1000
Maximum number of criteria evaluations
Block size
50
Maximum number of buffered evaluations (buffer is lost when stopping)
Evaluate best final
Evaluate criteria at best tuners when optimization is finished
Save best
Save best evaluation on file OptimizationBest.txt
Save history
Save evaluation history on file OptimizationHistory.txt
Listing
File
"OptimizationLog.log"
Name of file listing is written to
Do listing
Show optimization progress (at least initial and result values)
Do detailed listing
List best values and every listing increment evaluation
Listing increment
100
List evaluations with this minimal increment
Shown digits
3
Number of digits shown in listing
List on place
Delete listing history in Commands window
List tuners
List tuner values in each listed evaluation
OK
Info
Close

2. State of the art – Optimization with Jmodelica.org and Optimica

The Optimica extension [2] relies on the following mathematical description of an optimization problem constrained by differential algebraic equations (DAE):

$$\min_{u,p,t_0,t_f} \phi(\bar{z},p,t_0,t_f) + \int_{t_0}^{t_f} L(\dot{x}(t),x(t),w(t),u(t),t,p)\,dt$$
$$F(\dot{x}(t),x(t),w(t),u(t),t,p) = 0, t \in [t_0,t_f], F \in C^2$$
$$F_0(\dot{x}(t_0),x(t_0),w(t_0),u(t_0),t,p) = 0, t \in [t_0,t_f], F_0 \in C^2$$
$$C_{eq}(\dot{x}(t),x(t),w(t),u(t),t,p) = 0,\ t \in [t_0,t_f], C_{eq} \in C^2$$
$$C_{ineq}(\dot{x}(t),x(t),w(t),u(t),t,p) \le 0, t \in [t_0,t_f], C_{ineq} \in C^2$$
$$H_{eq}(\bar{z},p,t_0,t_f) = 0, H_{eq} \in C^2$$
$$H_{ineq}(\bar{z},p,t_0,t_f) \le 0, H_{ineq} \in C^2$$
$$\bar{z} = \left[\dot{x}(t_1),x(t_1),w(t_1),u(t_1),\dots,\dot{x}(t_{n_p}),x(t_{n_p}),w(t_{n_p}),u(t_{n_p})\right]$$
$$x_{min} \le x(t) \le x_{max}, w_{min} \le w(t) \le w_{max}, u_{min} \le u(t) \le u_{max}$$

Key elements include a Bolza cost function, a DAE equation for representing system dynamics, initial equations, inequality path and point constraints and equality path and point constraints.

The Optimica language extensions add to the Modelica language 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, finalTime and static
- A new section: constraint
- Inequality constraints

As an example of an Optimica specification, we consider the following Modelica model

```
model DoubleIntegrator
  Real x(start=0);
  Real v(start=0);
  input Real u;
equation
  der(x)=v;
  der(v)=u;
end DoubleIntegrator;
```

and the Optimica 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;
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;
```

The Optimica extension is supported by JModelica.org [3], where collocation and pseudospectral optimization algorithms are available to solve the corresponding discretized optimization problems. It is also supported by OpenModelica [5].

3. Optimization problems and NMPC tasks in Modelica

3.1. Techniques in Modelica to formulate optimization problems

There was an intensive discussion in the MODRIO group how to describe optimization setups in Modelica. In the appendix of this document a list of requirements can be found. The following options have been discussed to fulfill most of the requirements:

1. *Setup defined in a Modelica based scripting language:*
   Here, Modelica is used as scripting language and the optimization setup is defined in the Modelica scripting language. This is the approach used by the DLR Modelica Optimization library. The drawbacks are:
   (a) There is no "scripting language" standard for Modelica and only two tools (Dymola, OpenModelica) use Modelica as scripting language (but with some differences) and other Modelica tools use other scripting languages. Therefore, standardization of optimization setups seems to be not possible.
   (b) The basic setup is tedious since information available in the model needs to be repeated (such as hierarchical name of the parameter to be optimized to be provided as string, description text and unit of this parameter etc.). A tool can help with some GUI support, but this seems to be suboptimal.
2. *Setup defined by extensions to the Modelica language:*
   Here, new Modelica language elements are introduced to define the additional elements needed for an optimization setup. This is the approach used by the Modelon JModelica.org/Optimica environment. The advantage is that the optimization setup can be described with a nice syntax that is close to the mathematical problem formulation. The drawback is that every change/improvement in the optimization problem formulation needs to be newly designed with language elements and needs to be standardized via the Modelica Association. Furthermore, all tool vendors need to support the new language elements. It will be desirable to be able to express optimization problems in different ways for different applications, including restricted formulations that fit a particular solver. Unless a general formulation can be found to encompass these usages, the language extension will likely need to be updated several times.
3. *Setup defined by custom annotations:*
   Here, Modelica is extended in a generic way to add meta-information to a model. Then, the optimization setup is attached to the model as meta-information with this generic mechanism. A tool needs to provide a generic way to inquire the meta-information stored in a model in its scripting environment. Once this is available, a tool vendor has to extract the optimization meta-information from a model with its generic extracting mechanism and use this information to provide the setup of the model in the form needed by the optimization environment of the tool vendor. Meta-information for some standard optimization problems (like continuous-time nonlinear model predictive control (NMPC), or parameter optimization with equality and inequality constraints) can be standardized and then optimization setups can be very easily exchanged between tools (together with the underlying nonlinear models). However, a tool vendor has also an easy possibility to provide its own, specialized or generalized setups that will then only work in the respective tool, but using the same infrastructure (meta-information).

The approach that has been chosen in this body of work is approach (3) with custom annotations. Furthermore, the same approach could also be used for other MODRIO Work Packages, especially WP 2, 3, 4. With the "Modelica Change Proposal MCPI-0008", a formal proposal for custom annotations was made and discussed/improved in the Modelica Association. There is a prototype in Dymola supporting the latest custom annotation proposal. In the next sections the custom annotations approach is discussed in more detail on hand of the developed Modelica library `Optimization_CustomAnnotation`.

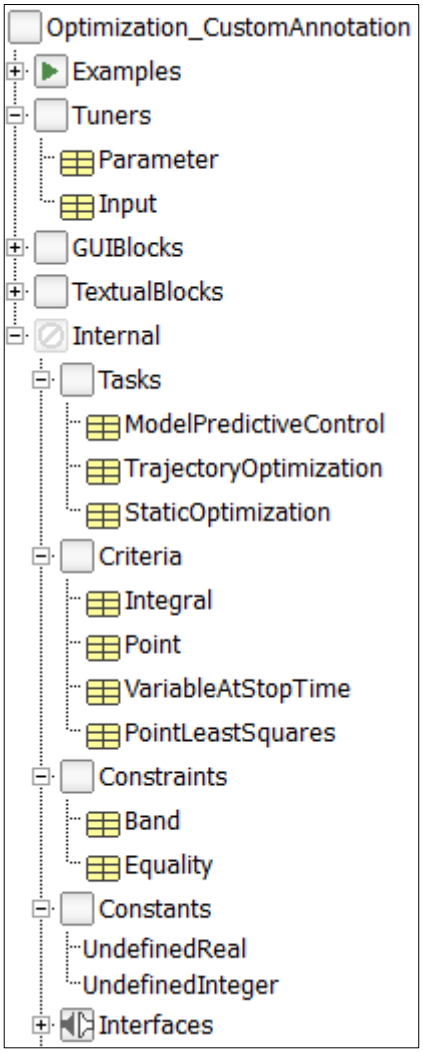
3.2. Custom Annotations

There have been several design discussions in MODRIO for "user-defined annotations" or "user-defined attributes" or "custom annotations" and several variants have been developed. The most promising approach has been formally defined in a "Modelica Change Proposal MCPI-0008_CustomAnnotations"" draft. This draft is available as standardization deliverable of MODRIO.

This section describes the proposal for optimization specifications in Modelica based on custom annotations. The accompanying Modelica file `Optimization_CustomAnnotation.mo` contains the actual specification.

3.2.1. Base Records

In a first step Modelica records are defined that reflect the content of the custom annotations for optimization setups. They are structured in records for optimization tasks, tuners, criteria and constraints (see library structure on the right). The goal is to only define the minimum set of custom annotations and to map lightly other information to these basic records. All packages with records are contained in the sublibrary `Internal` – only the package `Tuners` is on the first level, because it has to be used directly in all variants of optimization definitions, see the examples in Section 3.5.



The records in the sublibrary `Tasks` contain the specific data for the different optimization problem formulations that are not related to one of the model variables to be considered in the optimization task. Here, we can especially put information about the numerical simulation of the model (start time, stop time, solver, etc.):

```
record TrajectoryOptimization
  parameter String method="";
  parameter Real tolerance=1e-3;
  parameter Modelica.SIunits.Time startTime=0;
  parameter Boolean freeStopTime=false;
  parameter Modelica.SIunits.Time stopTime=1;
  parameter Modelica.SIunits.Time minStopTime=
                                  UndefinedReal;
  parameter Modelica.SIunits.Time maxStopTime=
                                  UndefinedReal;
end TrajectoryOptimization;
```

Three different kinds of optimization tasks are recognized: Model Predictive Control (online), trajectory optimization (to find optimal trajectories for e.g. states and inputs, along with optimal parameter values), and static optimization problems that have no time dependent variables.

In the sublibrary `Tuners` the records for a parameter and for an input time dependent signal are available to be varied by the optimization algorithm. The record `Tuners.Input` contains the following information for a trajectory optimization problem:

```
record Input
  parameter Boolean active=true;
  parameter Real min=-Modelica.Constants.inf;
  parameter Real max= Modelica.Constants.inf;
  parameter Real derMin=-Modelica.Constants.inf;
  parameter Real derMax= Modelica.Constants.inf;
  parameter Real u_tStart=Constants.UndefinedReal;
```

```
  parameter Real der_u_tStart=Constants.UndefinedReal;
  parameter Real der2_u_tStart=Constants.UndefinedReal;
  ...
  parameter Real u_init[:, 2]= [Internal.Constants.UndefinedReal,
                                Internal.Constants.UndefinedReal];
end Input;
```

Besides `active` all other parameters are optional (marked by `Constants.UndefinedReal`) and allow to define minimum and maximum values for the input as well as of its derivative. Furthermore, at the start and stop time, the value of the input, and/or its first and second derivatives can be pre-defined (as required by some applications). In order to start the optimization, an initial trajectory for the input is needed. In this version, a simple approach is used to define a two-column matrix of (time,input) rows through which the input trajectory shall pass. Without further information, these points shall be linearly interpolated.

The objective function of the optimization problem is constructed as a sum of maybe several criteria. Each criterion can be defined by means of one of the records in the sublibrary `Criteria`. The record `Integral` defines a Lagrange term of the optimization objective and has to be integrated (depending on the optimization method within the optimization algorithm) whereas `VariableAtStopTime` represents a kind of a Mayer term of the objective function.

Optimization constraints are defined via the records `Constraints.Band` and `Constrains.Equality`. Inequality constraints are special cases of band constraints $a \leq y \leq b$ for a variable y. Therefore, only a band record is provided. If a or b is left undefined in a band constraint, the corresponding inequality is not considered part of the optimization problem. Note that the Modelica attributes `min` and `max` are not considered to imply constraints to be used in the optimization (though they can be used to check the validity of an obtained optimization result).

The equality record has a parameter `required` which is the desired value of the variable to be annotated by the `Equality` record:

```
record Equality "The variable is constraint to a required value"
  parameter Boolean active=true "= true, if constraint is active";
  parameter Real required=0.0;
  parameter Real nominal=1.0;
  ...
end Equality;
```

3.2.2. Direct Application of Custom Annotations

In the second step the basic records are used in custom annotations, e.g. to completely define the Optimization setup from Section 1.2. A typical approach is to first define a model, say model `Aircraft`:

```
model Aircraft "Aircraft with simple controller"
  import SI    = Modelica.SIunits;
  import NonSI = Modelica.SIunits.Conversions.NonSIunits;
  parameter Real Kf(min=-12, max=1) = -6 "Controller parameter";
  parameter Real Ki(min=-12, max=1) = -2 "Controller parameter";
  output NonSI.Angle_deg overshoot    "Overshoot of angle of attack";
  output NonSI.Angle_deg maxElevator  "Maximum elevator deflection";
  output SI.Time         riseTime     "Rise time of angle of attack";
  output SI.Time         settlingTime "Settling time of angle of attack";
  ...
end Aircraft;
```

In this model the parameters to be optimized are defined as parameters (here: `Kf`, `Ki`). Furthermore, variables in the model are selected to be optimized, e.g., variable `overshoot` is the overshoot of a unit step response and it should be minimized. Typically, specialized, generic signal blocks are

provided to compute the desired criterion value from available model variables.

Finally, a new model is extended from the model above and an optimization setup is defined in this extended model:

```
model ControllerDesign "Optimization setup for controller design"
  extends Aircraft(
    Kf = -1.7 annotation(Tuners.Parameter(min=-10,
                                          max=0)),
    Ki = -2.0 annotation(Tuners.Parameter(active=false,
                                          min=-10,
                                          max=0),
    overshoot    annotation(Criteria.VariableAtStopTime(nominal=0.1)),
    maxElevator annotation(Criteria.VariableAtStopTime(nominal=2)),
    riseTime     annotation(Constraints.Band(max=0.5)),
    settlingTime annotation(Criteria.VariableAtStopTime(active=false,
                                        nominal=5)));
  annotation (Tasks.StaticOptimization(
         method="sqp", tolerance=1e-3, startTime=0, stopTime=10));
end ControllerDesign;
```

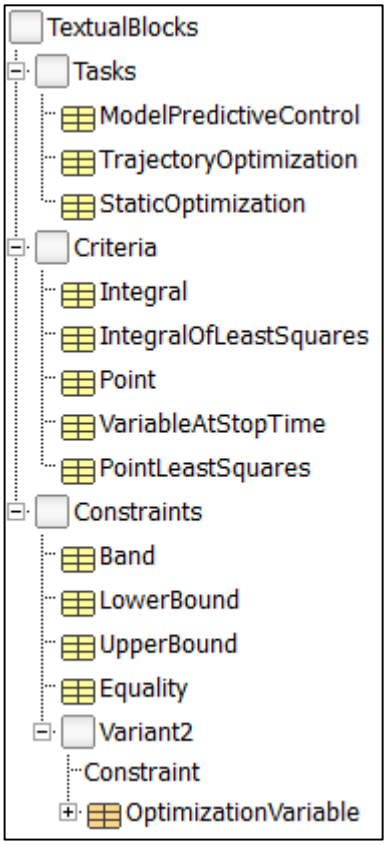
The default values for the parameters (here: `Kf`, `Ki`) are used as start values for the optimization (since no explicit initial guesses are given for them). The criteria values are scaled with the nominal value (e.g. the goal of the `overshoot` is to be smaller than 0.1 deg, and the goal of the `riseTime` is a hard constraint of 0.5 s). Then, the model `ControllerDesign` is translated and a simulation model is generated (no change to current Modelica models). Furthermore, the tool extracts the meta-information from the model (such as `Tuners.Parameter` or `Criteria.VariableAtStopTime`) and configures its optimization environment with it.

In order to be as user friendly as possible, all the configuration options might be just used to fill a GUI for the optimization and then the user can interactively modify exactly the specified tuners, criteria and other options, see [4] for this feature in the Optimization Library for Dymola. The direct application of custom annotations for optimization setups shown in this section requires typing some text that is difficult to read by the user. To support the users in specifying optimization setups two different sublibraries `TextualBlocks` and `GUIBlocks` have been developed, see the following Sections 0 and 3.4.

3.3. Textual Blocks for Modeling of Optimization Setups

The sublibrary `TextualBlocks` (see to the right) provides models and blocks that shall be used for textual modeling of optimization setups. The library is structured in the subpackages

- `Tasks`,
- `Criteria`,
- `Constraints`.



Each block (only `IntegralOfLeastSqaures` is slightly different) in the sublibraries `Criteria` and `Constraints` contains an input definition like the following:

```
parameter Real <p> = -10.0;
input Real signal
annotation (Optimization_CustomAnnotation(
             <record component> = <p>, ...));
```

For example, the block `Constraints.Equality` contains:

```
parameter Boolean active=true;
parameter Real nominal=1.0 "Scale";
parameter Real required "Required value";
parameter Real T[:]=fill(0.0, 0);
input Real signal annotation (
   Optimization_CustomAnnotation(Internal(
     Constraints(Equality(active=active,
     required=required, nominal=nominal, T=T)))));
```

All the relevant record components of the corresponding custom annotation are set in the annotation of the input instantiation. The values are transferred from parameters of the blocks. The advantage for the user is, that the user only has to provide parameter values to blocks and models – the annotations themselves are hidden to the user, making the optimization description more readable.

3.3.1. Sublibrary Tasks

This sublibrary defines the task to be carried out and the parameters for the respective task. The optimization model must instantiate one of these models, such as `ModelPredictiveControl`. The instantiated model has instances of the parameters of the respective task:

```
model ModelPredictiveControl "Global options for model predictive control"
  extends Internal.Tasks.ModelPredictiveControl;
  annotation (Optimization_CustomAnnotation(Internal(Tasks(
             ModelPredictiveControl(method=method, tolerance=tolerance,
             samplePeriod=samplePeriod, modelHorizon=modelHorizon,
             controlHorizon=controlHorizon)))));
end ModelPredictiveControl;
```

The basic record for `ModelPredictiveControl` is the following:

```
record ModelPredictiveControl "Global options for model predictive control"
  parameter String method="";
  parameter Real tolerance=1e-3;
  parameter Modelica.SIunits.Time samplePeriod=0.1;
  parameter Integer modelHorizon(min=1)=1;
  parameter Integer controlHorizon(min=1)=1;
end ModelPredictiveControl;
```

In the task model the values of all the record parameters are set to the custom annotation components to support the user.

3.3.2. Sublibrary Criteria

The sublibrary `Criteria` contains various blocks to define the optimization criteria. Several criteria can be defined in a model. Every criterion has a parameter `active` to activate or deactivate the criterion and parameter `nominal` to define a nominal criterion value. The nominal values are used to scale the criteria with respect to each other. If several criteria are defined in a model, then the sum of all the (scaled) criteria shall be minimized.

For example, in the block `VariableAtFixedTime` all the record parameters of `Internal.Criteria.VariableAtFixedTime` are transferred to the corresponding annotation of the input signal `signal`:

```
block VariableAtFixedTime "The variable value at a fixed time instant"
  input Real signal annotation (Optimization_CustomAnnotation(Internal(
                                Criteria(VariableAtFixedTime(active=active,
                                                    nominal=nominal, T=T)))));
  extends Internal.Criteria.VariableAtFixedTime;
end VariableAtFixedTime;
```

Here, the basic record is

```
record VariableAtFixedTime "The variable value at a fixed time instant"
  parameter Boolean active=true "= true, if criterion is optimized";
  parameter Real nominal=1.0 "Nominal value of criterion";
  parameter Modelica.SIunits.Time T=0.0
                       "The value of the variable at time T is minimized.";
end VariableAtFixedTime;
```

It is used to define a criterion for the value of the input signal `signal` at a fixed time instant `T`.

3.3.3. Sublibrary Constraints

The same principle as for criteria is also applied for constraint blocks. For example, the block `TextualBlocks.Constraints.Band` is defined in the following:

```
block Band "The variable is constrained to stay in a defined band"
  input Real signal annotation (
      Optimization_CustomAnnotation(Internal(Constraints(Band(
          active=active, min=min, max=max, nominal=nominal, T=T)))));
  extends Internal.Constraints.Band;
end Band;
```

It extends from the basic record Band:

```
record Band "The variable is constraint by an upper and/or lower bound"
  parameter Boolean active=true "= true, if constraint is active";
  parameter Real min=Constants.UndefinedReal "Optional minimum value";
  parameter Real max=Constants.UndefinedReal "Optional maximum value";
  parameter Real nominal=1.0 "Scaling: (min <= variable <= max) / nominal";
  parameter Real T[:]=fill(0.0, 0) "Time instants at which the band
                                          constraint shall be fulfilled";
end Band;
```

3.3.4. Definition of Constraints by Overloaded Operators

The overloaded operators approach allows to formulate constraints with operators $<$, $\leq$, $\geq$, $>$, $==$. In literature constraints are mostly formulated by relations, e.g. $x(t) \leq 0.5$. It is desirable for a user to formulate these relations directly in Modelica.

In Modelica a relation, like $x(t) \leq 0.5$ is a Boolean-expression, but for an optimization algorithm more is needed than the information whether a constraint is fulfilled or not. Generally, optimization methods work with the (real valued) residuum of the constraint, e.g. $x(t) - 0.5$. Therefore a mapping from a relation to a residuum is realized by overloaded operators:

```
operator record OptimizationVariable
  Real exp;

  encapsulated operator 'constructor'
    function fromReal
      input Real exp;
      output OptimizationVariable res(exp=exp);
    algorithm
    end fromReal;
  end 'constructor';

  encapsulated operator function '<'
    import Optimization_CustomAnnotation....Interfaces.BasicConstraint;
    input OptimizationVariable lhs;
    input OptimizationVariable rhs;
    output BasicConstraint res(res=lhs.exp - rhs.exp, op="<");
    // lhs < rhs
    algorithm
  end '<';

  encapsulated operator function '<='
    import Optimization_CustomAnnotation....Interfaces.BasicConstraint;
    input OptimizationVariable lhs;
    input OptimizationVariable rhs;
    output BasicConstraint res(res=lhs.exp - rhs.exp, op="<=");
    // lhs < rhs
    algorithm
  end '<=';
⋮

end OptimizationVariable;
```

The new operators return a record with a residual expression and the operator type e.g. $x \leq 0.5$ will return $\begin{pmatrix} x - 0.5 \\ " \leq " \end{pmatrix}$. In the next step it is possible to use the record `Constraint.Band` (see Section 3.2.1) in a custom annotation like the following:

```
block Constraint
    input Internal.Interfaces.BasicConstraint condition;
    input Real[:] T=fill(0, 0);
    input Real nominal=1.0;
    input Boolean active=true;
    Real res=condition.res annotation (Optimization_CustomAnnotation(
            Internal(Constraints(Band(
            active=active,
            min=minValue,
            max=0.0,
            nominal=nominal,
```

```
            T=T)))));
protected
    Real minValue;
algorithm
    minValue := if condition.op <> "==" then 0.0 else -Modelica.Constants.inf;
end Constraint;
```

The operator type leads to a value `-Modelica.Constants.inf` or `0` for the record component `min`. For technical reasons, the constraint is mapped to the record `Constraints.Band` even in the case of equality constraints. This is different to the approach in the previous sections with `Constraints.Equality`, but both ways are equivalent.

To define an inequality constraint the following two lines of Modelica code have to be written:

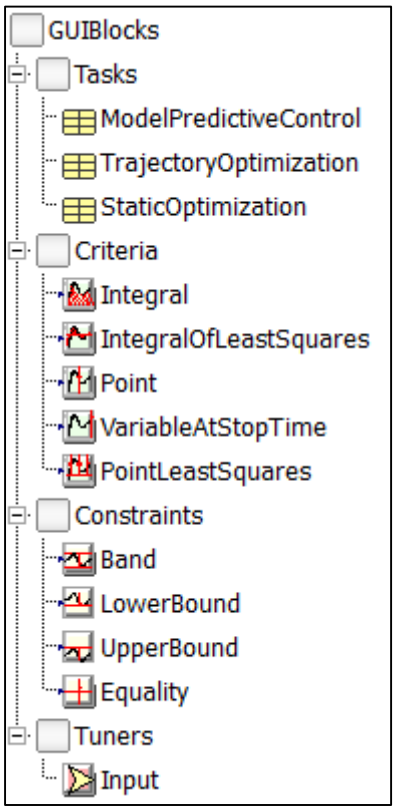
```
OptimizationVariable x_o = OptimizationVariable(x);
Constraint x_Constraint(condition=x_o <= 0.5, nominal=7.9);
```

In the same way an equality constraint can be constructed:

```
OptimizationVariable x_o = OptimizationVariable(x);
Constraint x_Constraint(condition=x_o == 0.5, nominal=7.9);
```

3.4. Library for Graphical Modeling of Optimization Setups

The complete setup for an optimization task may contain a complex and large record structure with several hierarchies. To support the user in providing the setup, graphical modeling of custom annotations (here for optimization setups) is proposed in the following.



The idea is that the optimization setup is defined by dragging and connecting blocks and that in these blocks the custom annotations are present. Therefore, the sublibrary `GUIBlocks` is provided, see figure to the right. In the different subpackages

- `Tasks`,
- `Criteria`,
- `Constraints`
- `Tuners`

the "user" blocks are present that shall be used by the user to define the setup graphically. Each of the blocks in the sublibraries `Criteria` and `Constraints` contains an input definition like the following:

```
parameter Real <p> = -10.0;
Modelica.Blocks.Interfaces.RealInput u
annotation (Optimization_CustomAnnotation(
             <record component> = <p>, ...));
```

For example, the block `Constraints.LowerBound` contains:

```
parameter Real min=0 "Minimum value";
parameter Real nominal=1.0 "Scale";
parameter Real T[:]=fill(0.0, 0);
Modelica.Blocks.Interfaces.RealInput u
annotation (Optimization_CustomAnnotation(Internal(Constraints(Band(
                        active=active, min=min, nominal=nominal, T=T)))));
```

Analog to the textual variant of optimization setups in Section 0, all the relevant record components of the corresponding custom annotation are set in the annotation of the input instantiation. The values

are transferred from parameters of the blocks that may be edited by the user in the parameter menu of the blocks. The user only has to connect the corresponding signal to the input of the criteria or constraint block and to provide values for the parameter of the blocks.

3.4.1. Sublibrary Tasks

For example, the parameter menu for the task `ModelPredictiveControl` looks like the following:

method	""		Optimization method (empty string means: default method)
tolerance	1e-3		Relative error tolerance for the optimization solution
samplePeriod	0.1	s	Sample period for the online optimization
modelHorizon	1		Number of samples for which the model behavior is predicted
controlHorizon	1		Number of samples provided for the predicted control inputs

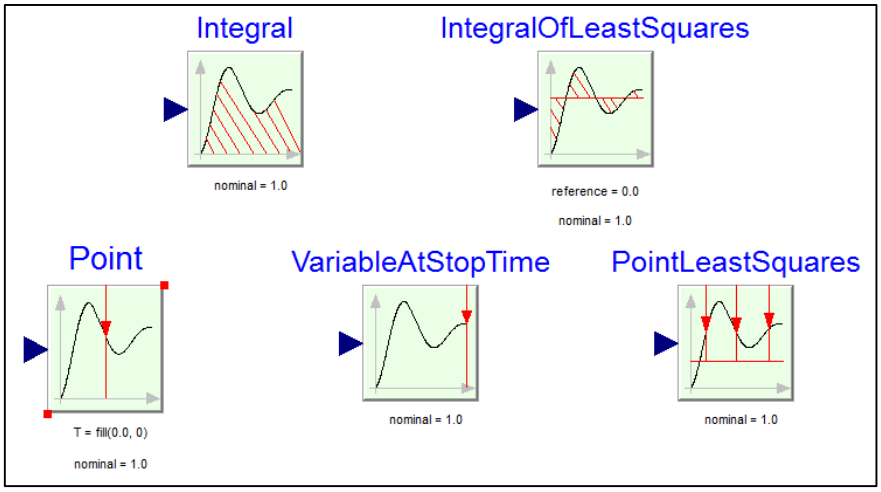
On the other hand, the task `TrajectoryOptimization` gives the following menu:

method	""		Optimization method (empty string means: default method)
tolerance	1e-3		Relative error tolerance for the optimization solution
startTime	0.0	s	Start time of simulation
freeStopTime	☐		= true: stop time shall be optimized
stopTime	1.0	s	Stop time of simulation (freeStopTime=false) or initial guess of stop time (freeStopTime=true)
minStopTime	UndefinedReal	s	Minimum value of stopTime (if freeStopTime=true)
maxStopTime	UndefinedReal	s	Maximum value of stopTime (if freeStopTime=true)

In this task it is possible to define that the stop time shall be optimized (`freeStopTime` = true).

3.4.2. Sublibrary Criteria

Currently, there are 5 criteria blocks. The functionality is already described in previous sections.



For example, block `IntegralOfLeastSquares` is defined by the input signal `u` to the block and the integral of the least squares error with respect to a reference value is minimized (special Lagrange term).

The menu of this block is:

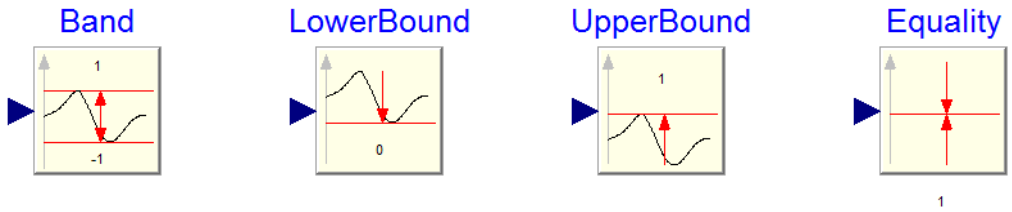
active	☑	= true, if criterion is optimized
nominal	1.0	Nominal value of criterion (input/nominal is minimized)
reference	0.0	Reference value. Minimize Integral((input-reference)^2/nominal * dt)

The block is formulated in Modelica with auxiliary variables and then mapped to the generic Lagrange term in the record `Internal.Criteria.Integral`:

```
block IntegralOfLeastSquares "The integral of the least squares error"
  extends Internal.Criteria.Integral;
  Modelica.Blocks.Interfaces.RealInput u;
  final output Real leastSquares=(u - reference)^2
        annotation (Optimization_CustomAnnotation(Internal(Criteria(
                      Integral(active=active, nominal=nominal)))));
end IntegralOfLeastSquares;
```

3.4.3. Sublibrary Constraints

The sublibrary `Constraints` contains the following blocks to define inequality and equality constraints:

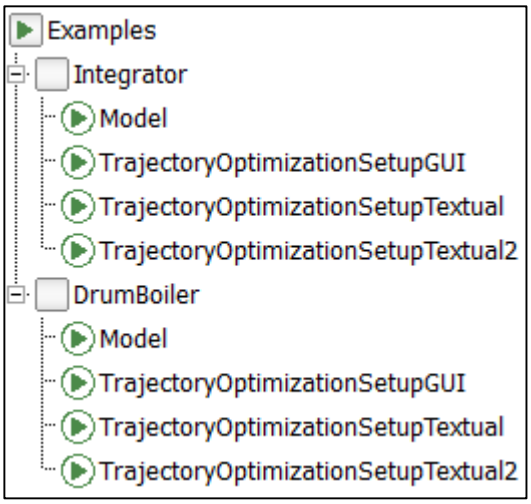


All constraints can be defined to hold continuously at all-time instants, or only at particular time instants. For example, the menu of block `Band` is:

active	☑	= true, if constraint is active
nominal	1.0	Nominal value of constraint (scaled constraint is formulated with input/nominal)
min	UndefinedReal	Minimum value of variable
max	UndefinedReal	Maximum value of variable
T	fill(0.0, 0)	Time instants at which the band constraint shall be fulfilled. If zero-dimension, band constraint holds for all time instants

3.5. Examples

Currently, there are two examples available in the library that are structured in a similar way:



`Integrator` and `DrumBoiler` are the names of the examples. The corresponding subpackages contain generic models `Model` that are independent of the individual optimization task. The other models, such as `TrajectoryOptimizationSetupGUI` are inherted from the respective `Model` and add a particular setup, e.g.

```
model TrajectoryOptimizationSetupGUI
"Optimization setup for model Integrator"
  extends Model(...);
  ...
end TrajectoryOptimizationSetupGUI;
```

3.5.1. Integrator

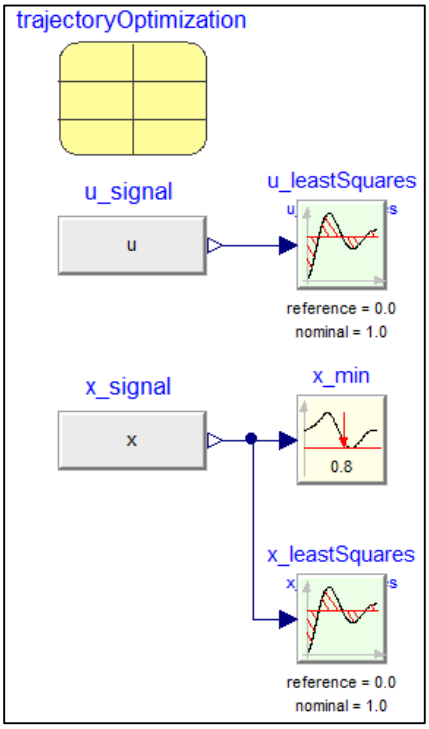
The example `Integrator.Model` (initially provided by Johan Åkesson) is defined purely textually:

```
model Model "Model to be used for optimization"
  extends Modelica.Icons.Example;
  input Real u;
  Real x(start=1, fixed=true);
equation
  der(x) = u;
end Model;
```

It describes an integrator. The mathematical optimal control problem to be solved for the integrator is

$$\min_{-3\le u(t)\le 3} \int_0^1 (u(t)^2 + x(t)^2)\,dt \quad \text{such that} \quad x(t) \ge 0.8,$$

$$\dot{x}(t) = u(t) \quad \text{for} \quad 0 \le t \le 1 \quad \text{and} \quad x(0) = 1.$$



A trajectory optimization setup is defined in `TrajectoryOptimizationSetupGUI` in a graphical way, by first providing the needed signals `u` and `x` via `Modelica.Blocks.Sources.RealExpression` blocks in the diagram layer and then adding the optimization setup with the configuration blocks, see figure on the right. Additionally, for the input tuner variable `u`, the following textual definition has to be made:

```
extends Model(u annotation (
          Optimization_CustomAnnotation(
            Tuners(Input(min=-3, max=3)))));
```

It is still an open issue how to graphically put this information into the model, see Section 0.

The textual variants for the setup definitions are contained in the models `TrajectoryOptimizationSetupTextual*`. In the first variant we have

```
model TrajectoryOptimizationSetupTextual
  "Optimization setup for model Integrator with textual definition"
```

```
  import Optimization_CustomAnnotation.TextualBlocks.*;
  extends Model(u annotation (Optimization_CustomAnnotation(Tuners(Input(
              min=-3, max=3)))));
  Tasks.TrajectoryOptimization trajectoryOptimization(stopTime=1);
  Criteria.Integral criterion(signal=x^2 + u^2);
  Constraints.LowerBound lowerBound(signal=x, min=0.8);
end TrajectoryOptimizationSetupTextual;
```

Here, analog to the graphical definition the annotation to the input variable has to be put directly into the modifier of the inheritance of the model `Model`. For the criteria and constraints, the value of the variables can be directly defined by setting the input variable `signal`.

In Variant 2 of the setup, the only difference is the definition of the constraint $x \geq 0.8$. It is defined by

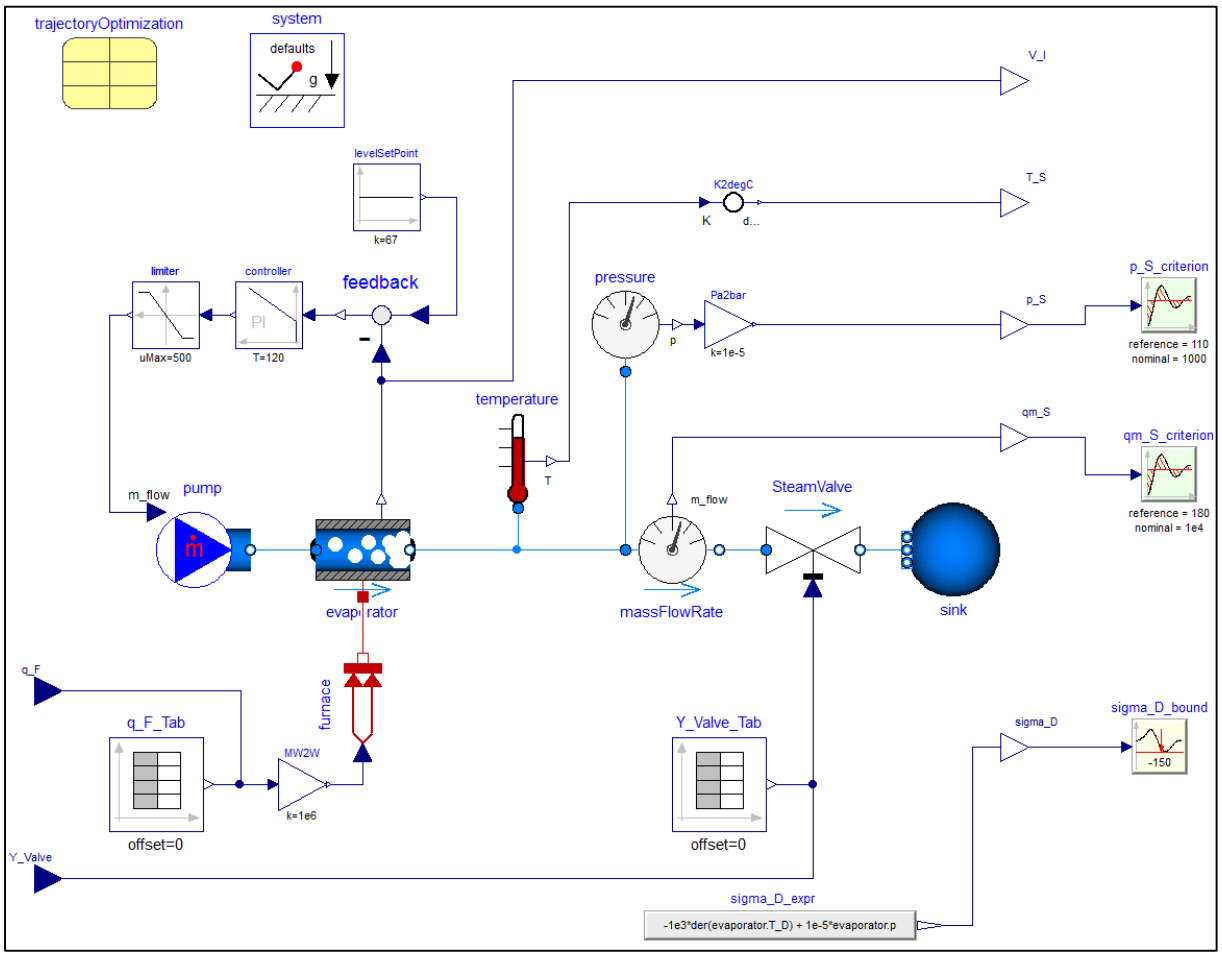
```
  OptimizationVariable x_o=OptimizationVariable(x);
  Constraint x_Constraint(condition=x_o >= 0.8);
```

instead of

```
  Constraints.LowerBound lowerBound(signal=x, max=0.8);
```

3.5.2. DrumBoiler

The `DrumBoiler.Model` example (provided by Rüdiger Franke) is defined graphically using blocks of the `Modelica.Fluid` library. The model `DrumBoiler.TrajectoryOptimizationSetupGUI` inherits from `Model` and adds the optimization setup mostly graphically:



The outputs `p_S`, `qm_S` and `sigma_D` are connected to blocks of the library

`Optimization_CustomAnnotation.GUIBlocks` to define criteria and constraints.

The following optimal control problem is considered with the translated model equations $\dot{x} = f(x, q_F, Y_{Valve})$ for the Modelica model `DrumBoiler.Model`:

$$\min_{q_F(t),\, Y_{Valve}(t)} \int_0^{3600} \left(\frac{1}{1000}(p_S(t) - 110)^2 + \frac{1}{10000}(qm_S(t) - 180)^2 \right) dt \qquad \text{such that}$$

$$\sigma_D(t) \geq -150, \qquad 0 \leq Y_{Valve}(t) \leq 1, \qquad 0 \leq q_F(t) \leq 500, \qquad -25/60 \leq \dot{q}_F(t) \leq 25/60, \qquad q_F(0) = 0,$$

$$\dot{x} = f(x, q_F, Y_{Valve}) \qquad \text{for} \qquad 0 \leq t \leq 3600.$$

Using the parameter menus of these blocks and of the `trajectoryOptimization` record (in the upper left of the model diagram) most of the relevant information of the optimization problem can be graphically defined. Only, the annotations of the input tuner variables have to be defined textually (analog to the integrator example in the previous section), see code below.

The textual variants of the setups look like the following:

```
model TrajectoryOptimizationSetupTextual
  import Optimization_CustomAnnotation.TextualBlocks.*;
  extends Model(
    use_inputs=true,
    q_F annotation (Optimization_CustomAnnotation(Tuners(Input(
            min=0,
            max=500,
            derMin=-25/60,
            derMax=25/60,
            u_tStart=0,
            u_init=[0.0, 0.0; 60*60, 400])))),
    Y_Valve annotation (Optimization_CustomAnnotation(Tuners(Input(
            min=0,
            max=1,
            u_init=[0.0, 1.0])))));
  Tasks.TrajectoryOptimization trajectoryOptimization(stopTime=3600,
                                                      tolerance=1e-6);
  Criteria.IntegralOfLeastSquares p_S_criterion(signal=p_S, reference=110,
                                                nominal=1000);
  Criteria.IntegralOfLeastSquares qm_S_criterion(signal=qm_S,
                                                 reference=180, nominal=1e4);
  Constraints.LowerBound sigma_D_bound(signal=sigma_D, min=-150);
end TrajectoryOptimizationSetupTextual;
```

The task, criteria and constraints are defined by the corresponding blocks in the sublibrary `TextualBlocks`, each one modifying the input variable `signal` and the corresponding record parameters. In Variant 2 the constraint $\sigma_D(t) \geq -150$ is formulated with

```
  OptimizationVariable sigma_D_o=OptimizationVariable(sigma_D);
  Constraint sigma_D_Constraint(condition=sigma_D_o >= -150);
```

instead of

```
  Constraints.LowerBound sigma_D_bound(signal=sigma_D, min=-150);
```

4. Limitations for Custom Annotations

The main limitation of the current concept is the following:

- **Tuner parameters and Input tuners not defined graphically:**
  It is not yet clear how "parameters" or "inputs" used as tuners can be represented in a graphical way or simple textual way. With the current library the user has to define a hierarchical modifier on the annotation of a parameter/input signal and define that the parameter/input signal is a tuner.
  Because it is a general problem of custom annotations (without any relation to optimization) how to support the user to provide the information to parameters and input signals, this has to be discussed in a more general context. A possible solution could be that a Modelica environment provides a separate GUI for custom annotations.

5. References

[1] Andreas Pfeiffer: *Optimization Library for Interactive Multi-Criteria Optimization Tasks*. Proc. of 9th International Modelica Conference, pp. 669-679, Munich, Germany, Sept. 2012.

[2] Johan Åkesson: "Optimica---An Extension of Modelica Supporting Dynamic Optimization". In *In 6th International Modelica Conference 2008*, Modelica Association, March 2008.

[3] Johan Åkesson, Karl-Erik Årzén, Magnus Gäfvert, Tove Bergdahl, Hubertus Tummescheit: "Modeling and Optimization with Optimica and JModelica.org—Languages and Tools for Solving Large-Scale Dynamic Optimization Problems". *Computers and Chemical Engineering,* 34:11, pp. 1737–1749, November 2010.

[4] Deliverable D5.2.3: *Prototype for Optimization Toolchain in Dymola-Optimization*, ITEA2 project MODRIO (11004), Version 1.0, 2014.

[5] Peter Fritzson et al.: *OpenModelica.* url: https://openmodelica.org/

Appendix - Requirements on Modelica defined Optimization Setups

ID	Requirement	Compliance statement for the Custom Annotation approach	Comment
MOE-01	The Modelica Optimization Extension (MOE) shall be non-discriminative with respect to optimization algorithm	comply	See MOE-02c, however
MOE-02a	The MOE shall support optimization based on systems in ODE form	comply	Before the model is passed to the optimizer, it is transformed into an ODE
MOE-02b	The MOE shall support optimization based on systems in semi-explicit DAE form	comply	Before the model is passed to the optimizer, variables are eliminated that can be eliminated analytically. Remaining algebraic equations are exposed as residuals.
MOE-03a	The MOE shall allow for cost functions yielding convex optimization problems whenever possible, when the system is represented in ODE or semi-explicit DAE form	comply	This is not a problem since a cost expression can be represented with an intermediate variable that is eliminated; see MOE-03b
MOE-04a	The MOE shall allow for constraints yielding convex optimization problems whenever possible, when the system is represented in ODE or semi-explicit DAE form	comply	This is not a problem since a constraint expression can be represented with an intermediate variable that is eliminated; see MOE-04b
MOE-05	The MOE shall provide means to specify Modelica variables to optimize: both parameters (i.e., tuners) and top-level inputs	comply	
MOE-07	The MOE shall support common cost function types for dynamic optimization problems such as Lagrange, Mayer and Bolza in such a way that they can be efficiently exploited by numerical algorithms	comply	

MOE-08	The MOE shall support cost functions involving measurement data in such a way that it can be efficiently exploited by numerical algorithms, e.g., Gauss-Newton	comply	
MOE-09	The MOE shall support specification of initial and terminal equality and inequality constraints	comply	Initial constraints are important in MHE estimation formulations
MOE-11	The MOE shall support specification of point equality and inequality constraints, i.e., specification of variable values at specified point in time in the optimization interval	comply	
MOE-13	The MOE shall support minimum time problems	comply	
MOE-14	The MOE shall allow to express an optimization problem such that a compiler can process all elements of the optimization formulation symbolically	comply	Modelica constructs such as external C functions shall still be allowed
MOE-17	The MOE shall enable a compiler to issue certificates regarding smoothness of the functions in the optimization problem	comply	Modelica offers constructs to express smoothness order of expressions and functions, so this should not be a problem. It would be desirable to, e.g., let FMUs indicate level of smoothness to aid a tool in choosing an appropriate solver.
MOE-18	The MOE shall support bounds to be specified on all model variables	comply	
MOE-19	The MOE shall support initial guesses to be specified for optimization parameters	comply	Note that this is different than the start attribute, which specifies the value of a variable at initial time. Allow to specify a function of time as initial guess for a trajectory.
MOE-20	The MOE shall provide means to specify bounds on derivative variables	comply	
MOE-22	The MOE shall provide means to specify nominal values on optimization variables for appropriate problem scaling	comply	By MOE-24a, the Modelica nominal attribute should be used

MOE-23	Non Functional Requirement: The MOE shall allow the definition of the optimization problem based on all relevant information (cost function, constraints, …) at one place.	comply	It may also be possible to put parts of the optimization problem in a model, and parts in the actual problem formulation
MOE-24a	Modelica attributes such as nominal, start, etc., other than min/max, should be taken as part of the optimization problem whenever reasonable	comply	
MOE-24b.1	The Modelica attributes min and max should be disregarded in the optimization problem, and another way should be supplied to express bounds on optimization variables.	comply	The function of the min and max attributes is to express the region of validity for a model, while bounds in optimization limit the search space. These are different things. It is undesirable that an optimization problem changes because when a variable is changed from one Real subtype to another.
MOE-25	The scope of the MOE is how to express optimization problems in Modelica, not scripting that one might want to use with such problems.	comply	
MOE-27	It is desirable that Modelica code written to the MOE be human-friendly to read and write. This should be used to help guide the design.	comply	Comply as much as possible within the limits of what custom annotations allow.
MOE-28	The MOE shall support modeling of time delays.	comply	A tool may support the Modelica delay operator in optimization problems.
MOE-29	The MOE shall support to pose arbitrary NLPs of the form min f(x) subject to g(x)=0, h(x)>=0, where g and h are vector valued, and the x vector may contains model parameters.	comply	This should not be a problem since the problem space that we want to cover is larger, but it serves as a sanity check. Perhaps not all functions f, g, h, expressible in Modelica may be encoded, but it should be possible to encode an equivalent problem.
MOE-31	The MOE shall be extendable	comply	

MOE-02c	The MOE shall support optimization based on systems in fully implicit DAE form	do not comply	The model is passed to the optimizer in its original DAE form, without eliminating any variables. This poses the need to avoid introducing intermediate variables for, e.g., costs and constraints.
MOE-03b	The MOE shall allow for cost functions yielding convex optimization problems whenever possible, even when the system to be optimized is represented a fully implicit DAE where no variables have been eliminated.	do not comply	A convex optimization formulation, e.g., min_{x,y} x^2 + y^2 subject to x + y=1 becomes non-convex when introducing an intermediate cost variable: min_{x,y,z} z subject to z = x^2 + y^2 x + y = 1 as long as the cost function is nonlinear. To allow the former kind of formulation even if no variables are eliminated, it is necessary be able to supply cost expressions in the model, not just cost variables.

MOE-04b	The MOE shall allow for constraints yielding convex optimization problems whenever possible, even when the system to be optimized is represented a fully implicit DAE where no variables have been eliminated.	do not comply	A convex optimization formulation, e.g., min_{x,y} x^2 + y^2 subject to y-x^2>=0 becomes non-convex when introducing an intermediate constraint variable: min_{x,y,z} x^2+y^2 subject to z = y-x^2 z>=0 as long as the constraint function is nonlinear. To allow the former kind of formulation when no variables are eliminated, it is necessary be able to supply constraint expressions, not just constrained variables.
MOE-24b.2	The Modelica attributes min and max should be used to form interval constraints on variables in the optimization problem; no other dedicated feature should be created for this.	do not comply	This is an alternative to MOE-24b.1. It is usually desirable to respect the region of validity when solving an optimization problem. Providing more than one way to bound a variable only leads to confusion.
MOE-26	The MOE shall keep the notion of an equation based language. This means that constraints are formulated in an equation section. It shall therefore be avoided to switch equations on and off according to settings in an annotation.	do not comply	
MOE-12	The MOE shall support multi-objective optimization	leave out initially	This is probably not so much of relevance for NMPC, but may still be of interest for other optimization applications

MOE-15	The MOE shall support multi-phase systems	leave out initially	A multi-phase system is a multi-mode system where the switching sequence is fixed beforehand, though the duration of each phase may be unknown. This is much easier to optimize than a full multi-mode system.
MOE-16	The MOE shall support mixed integer optimization problems	leave out initially	
MOE-21	The MOE shall allow to embed an NMPC controller in a Modelica model, where the optimization problem is specified according to the MOE; all within Modelica	leave out initially	
MOE-30	The MOE shall support criteria defined in the optimization environment and criteria defined in the model	does not apply	We do not aim to standardize the optimization environment, only the modeling environment.