Project Deliverable

Project Number:	Project Acronym:	Project Title:
611281	DYMASOS	Dynamic Management of Physically Coupled Systems of Systems

Instrument:	Thematic Priority
Collaborative Project	ICT

Title

D4.2: Report on Modelica Extensions for the Specification of Distributed Optimization Problems

Contractual Delivery Date:	Actual Delivery Date:
March 2015 (M18)	April 02, 2015

Start date of project:	Duration:
October 1, 2013	36 months

Organization name of lead contractor for this deliverable:	Document version:
TUDO	V1.0

Dissemination level (Project co-funded by the European Commission within the Seventh Framework Programme)		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission	
RE	Restricted to a group defined by the consortium (including the Commission)	
CO	Confidential, only for members of the consortium (including the Commission)	

Authors (organizations) :
Shaghayegh Nazari (TUDO), Christian Sonntag (TEX)

With the help of:
Yorrick Vissers (TU Eindhoven)

Reviewers (organizations) :
Mato Baotic (UNIZG-FER), Sebastian Engell (TUDO)

Abstract:

The simulation and validation framework that is currently developed in WP4 provides a structured approach for the simulation-based validation of complex automated physically connected Systems of Systems. It provides a structured approach to the implementation and interconnection of design / optimization and validation models, as well as of local and global optimization problem formulations and algorithms, and it offers standard interfaces for the validation of management methods on models of technical SoS.

A first prototype was applied successfully to a version of an integrated petrochemical site, the industrial case study by partner INEOS. The site is coordinated by a distributed price-based approach that was developed in WP2 of DYMASOS. However, in this prototype the model was configured manually in *Modelica*, and the optimization problem formulations were integrated with the implementations of the coordination algorithms.

This deliverable presents two crucial advances that have been made for the DYMASOS simulation and validation framework based on the conceptual specification that was described in [1]:

We have defined a XML-based configuration specification that will allow users to completely specify the structure and parameters of a DYMASOS CPSoS simulation model. This configuration specification will serve as an input for the *Modelica model generator* that will generate a ready-to-use CPSoS simulation model, including all interconnections, components, and interfaces, from the component repositories.

Furthermore, we have defined a general-purpose language for the specification of optimization problems that will enable users to extract the definition of optimization problems from their controller implementations and will thus facilitate the connection of different algorithms to solve the same optimization problem.

Over the next months, we will integrate the configuration specification with the information platform developed by partner RWTH and will aim to devise a standardized version that is based on established formalisms for automation systems exchange, such as *AutomationML*, and will implement both of these results into the simulation and validation framework prototype.

Keywords:

DYMASOS Simulation and Validation Framework, Optimization Problem Formulation, Configuration Specification

Disclaimer

THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Any liability, including liability for infringement of any proprietary rights, relating to use of information in this document is disclaimed. No license, express or implied, by estoppels or otherwise, to any intellectual property rights are granted herein. The members of the project DYMASOS do not accept any liability for actions or omissions of DYMASOS members or third parties and disclaims any obligation to enforce the use of this document. This document is subject to change without notice.

Revision History

The following table describes the main changes done in the document since it was created.

Revision	Date	Description	Author (Organisation)
V0.1	Jan. 2015	Creation	Christian Sonntag (TEX)
V0.2	Feb. 2015	Optimization problem language (first version)	Shaghayegh Nazari (TUDO), Christian Sonntag (TEX)
V0.3-V0.4	Mar. 2015	Survey of optimization tools	Shaghayegh Nazari (TUDO), Christian Sonntag (TEX)
V0.5-V0.6	Mar. 2015	Optimization problem language (revised version)	Shaghayegh Nazari (TUDO), Christian Sonntag (TEX)
V0.7	Mar. 2015	Configuration specification	Christian Sonntag (TEX)
V0.8	Mar. 2015	Version for review	Shaghayegh Nazari (TUDO), Christian Sonntag (TEX)
V0.9	Mar. 2015	Review comments submitted	Mato Baotic (UNIZG-FER)
V1.0	Apr. 2015	Final version	Christian Sonntag (TEX)

Table of Contents

1 INTRODUCTION AND SUMMARY 7

2 CONFIGURATION SPECIFICATION FOR THE DYMASOS MODEL GENERATOR ... 10

2.1 SIMULATION MODEL SPECIFICATION: FILE STRUCTURE 10

2.2 OVERVIEW OF THE CONFIGURATION SPECIFICATION 11

2.2.1 Component Definitions 11

2.2.1.1 Generic Interface between Controllers 12

2.2.1.2 Generic Interfaces between Controllers and Validation Models 12

2.2.1.3 Generic Interface between Validation Models 12

2.2.1.4 Event Interfaces 13

2.2.2 Sampling Times and Execution Delays 13

2.2.3 Structural Model Definition 13

2.2.3.1 Connections between Validation Models 13

2.2.3.2 Connections between Validation Models and Controllers 14

2.2.3.3 Event and Sampling Time Subscriptions 14

3 A SURVEY OF OPTIMIZATION PROBLEM FORMULATIONS 15

3.1 JMODELICA 15

3.1.1 Formulation of an Optimization Problem in JModelica 16

3.2 THE MATLAB OPTIMIZATION TOOLBOX 17

3.2.1 Dynamic Optimization in Matlab 18

3.3 GPROMS / GOPT 19

3.3.1 Definition of Cost Function, Process Model, and Control Vector Values 19

3.3.2 Definition of Variables 20

3.3.3 Definition of Constraints 20

4 DEFINITION OF THE GENERAL OPTIMIZATION PROBLEM FORMULATION LANGUAGE 22

4.1 VARIABLE DEFINITIONS 24

4.2 DEFINITION OF THE COST FUNCTION 25

4.3 EQUALITY AND INEQUALITY CONSTRAINTS 25

4.3.1 Nonlinear Equality and Inequality Constraints 26

4.3.2 Linear Equality and Inequality Constraints 26

4.4 INTERIOR-POINT CONSTRAINTS 27

4.4.1 Nonlinear Interior-point Constraints 27

4.4.2 Linear Interior-point Constraints 27

4.4.3 Initial-time and Final-time Constraints 27

4.5 OPTIMIZATION OPTIONS 28

5 CONCLUSIONS AND FUTURE WORK 30

BIBLIOGRAPHY 31

A EXAMPLE OF A XML CONFIGURATION FILE 32

List of Figures

Figure 1: Scope of Work Package 4. 7

Figure 2: Conceptual outline of the modeling and simulation framework. 8

Figure 3: Integrated workflow of the DYMASOS engineering platform. 8

Figure 4: Workflow for the generation and execution of automated CPSoS models. 9

Figure 5: Example of a file structure of a DYMASOS simulation model specification. 10

Figure 6: Revised interface definitions of the modelling and simulation framework. 12

Figure 7: Definition of the process model, the cost function, the type of the optimization problem, and the discretization values of the control vector parameterization. 20

Figure 8: Deffintion of optimization variables using the gPROMS GUI. 20

Figure 9: Definition of the constraints. 21

Figure 10: Conceptual UML sequence diagram illustrating the black-box access mode. 23

Acronyms and Definitions

Acronym	Definition
CAEX	Computer Aided Engineering Exchange
CPSoS	Cyber-physical System of Systems
CVP	Control Vector Parameterization
DAE	Differential-algebraic Equations
DLL	Dynamic Linked Library
FMI	Functional Mock-up Interface
FMU	Functional Mock-up Unit
GUI	Graphical User Interface
IPOPT	Interior Point Optimizer
JMI	JModelica Model Interface
JMU	JModelica Model Unit
LP	Linear Program
MINLP	Mixed-integer Nonlinear Program
MPC	Model-predictive Control
NLP	Nonlinear Program
ODE	Ordinary Differential Equations
QP	Quadratic Program
SoS	System of Systems
WP	Work Package
XML	Extensible Markup Language

1 Introduction and Summary

In Work Package 4 of DYMASOS, partners RWTH, TUDO, and TEX are developing the *DYMASOS engineering platform* that will facilitate the validation, deployment, and application of the developed management strategies for Cyber-Physical Systems of Systems (CPSoS). Since experiments with different strategies will be made in simulations of the real systems, the engineering tools have two main tasks: Exchanging data between the real-world CPSoS and its virtual representation, and building and running a CPSoS simulation based on of individual system simulations, see Figure 1.

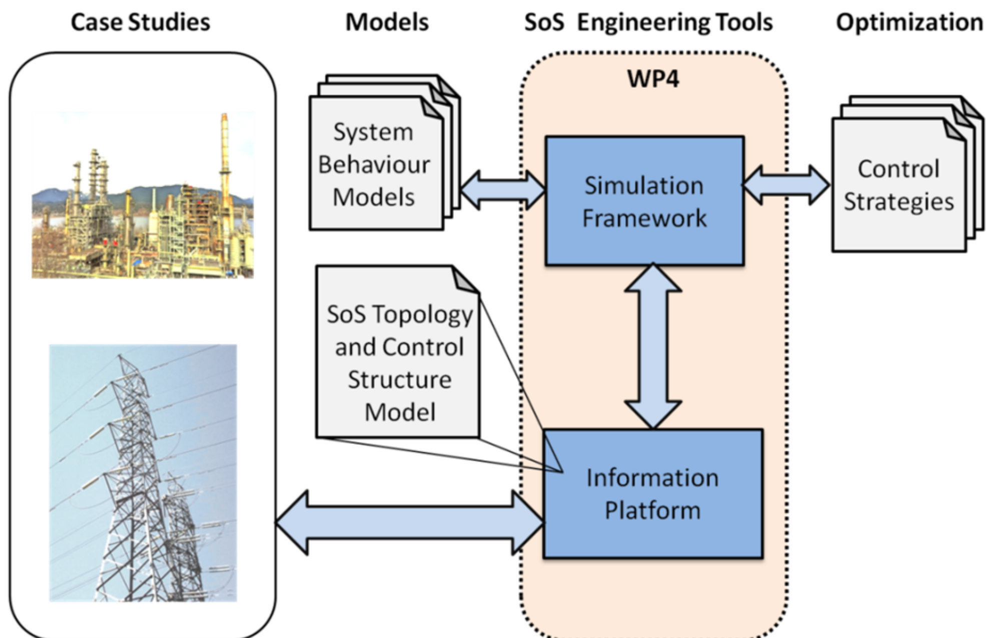

Figure 1: Scope of Work Package 4.

The simulation and validation framework is tailored towards the testing and validation of distributed management systems by:

- Providing a structured approach to the implementation and interconnection of design / optimization and validation models, as well as of local and global optimization problem formulations and algorithms, and
- Offering standard interfaces for the validation of management methods on models of technical SoS.

The conceptual outline of the simulation and validation framework is shown in Figure 2. Each subsystem of the automated CPSoS is represented by one of four different model elements: a *design model*, which is usually an abstract representation of the CPSoS subsystem that is used as a predictive tool by a *local control algorithm* to determine a locally optimal subsystem operation with respect to a *local problem formulation*. The local control algorithms perform the real-time control of the physical CPSoS that is represented by a set of *validation models*. These models are more detailed than the design models and accurately represent the real CPSoS. Optionally, a global coordination algorithm can be connected to the local control algorithms to enable the implementation of hierarchical management schemes.

An essential feature for the simulation and validation framework is that it provides standardized interfaces to which the management algorithms and the different types of models can be easily connected. Such standard interfaces also provide a straightforward avenue for the deployment of management solutions to industrial hardware systems at a later stage.

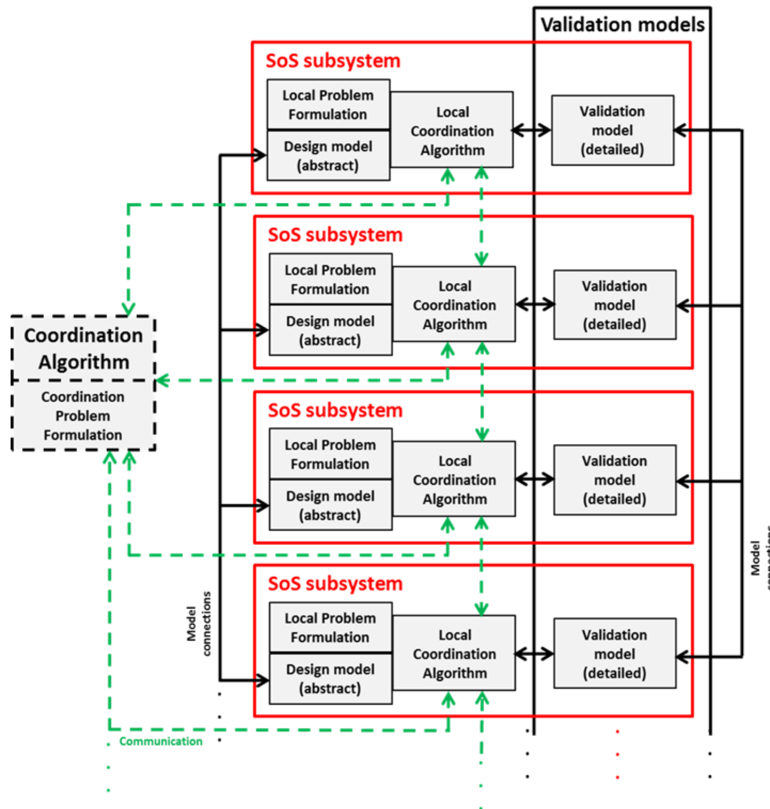

Figure 2: Conceptual outline of the modeling and simulation framework.

The workflow that we envision for the integrated application of the two tools of the DYMASOS engineering platform is shown in Figure 3. The information and integration framework that is developed by partner RWTH will, in addition to its other functionalities, provide facilities to model the structure, as well as all necessary parameters, of a CPSoS that are needed by the simulation and validation framework. The simulation and validation platform will then generate an overall CPSoS model based on this configuration information and existing model/algorithm repositories (see also Figure 4). Subsequently, the user can iteratively test and correct the developed coordination algorithms using simulations. Once the algorithms are correct, the user can then employ the information and integration platform to connect them to industrial hardware systems. To this end, suitable adapters must be provided for the general interfaces that are employed by the simulation and validation framework.

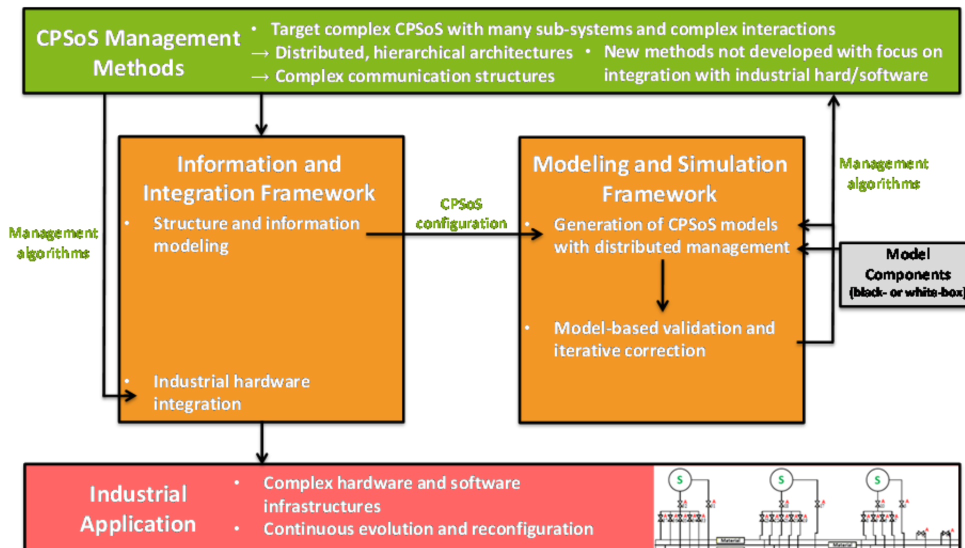

Figure 3: Integrated workflow of the DYMASOS engineering platform.

Over the last months, a first prototype of the simulation framework has been developed that demonstrates the viability of our approach [2]. This prototype was applied successfully to a version of an integrated petrochemical site, the industrial case study by partner INEOS (developed in Work Package 5). The site is coordinated by a distributed price-based approach that was developed in Work Package 2. This approach is implemented in Matlab and was integrated integrated with the case study model as standalone implementations in DLL form.

In the prototype of the simulation and validation framework used in [2], the model was configured manually in *Modelica*, and the optimization problem formulations were integrated with the Matlab-based implementations of the coordination algorithms. This deliverable presents two crucial advances that have been made for the simulation and validation framework based on the conceptual specification that was described in [1]:

- **Definition of an XML-based configuration specification** that will allow users to completely specify the structure and parameters of a DYMASOS CPSoS simulation model. As shown in Figure 4, this configuration specification will serve as an input for the *Modelica model generator* that will generate a ready-to-use CPSoS simulation model, including all interconnections, components, and interfaces, from the component repositories. In addition, a *Modelica-based model management engine* is generated that will be responsible for the coordination of all components.
- **Definition of a general-purpose language for the specification of optimization problems** that will enable users to extract the definition of optimization problems from their controller implementations and will thus facilitate the connection of different algorithms to solve the same optimization problem.

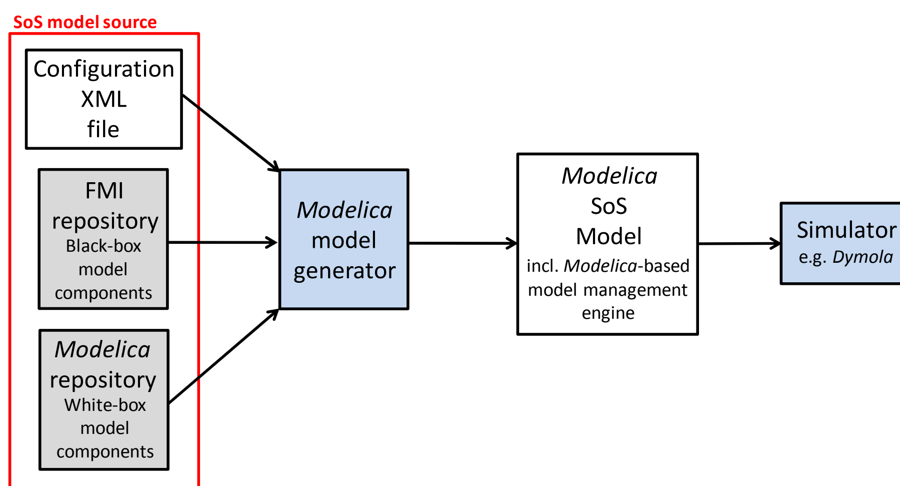

Figure 4: Workflow for the generation and execution of automated CPSoS models.

The remainder of this report is structured as follows: in section 2, the XML-based configuration specification is briefly described. Section 3 provides a brief of existing languages for the formulation of general optimization problems, followed by the presentation of our new formulation language in section 4. Finally, section 5 draws conclusions and provides an outlook on future work.

2 Configuration Specification for the DYMASOS Model Generator

The XML-based configuration definition that was developed in WP4 concretizes all of the concepts that were presented in [1] (sections 3.1, 3.2, and 4.2) into a specification language. The file can either be generated by the user or, at a later stage, by the information and integration framework, and will serve as an input to the *Modelica* model generator component.

This section provides a brief overview of this new specification. The example file on which this overview is based is provided in appendix A. Note that the example file contains extensive comments and should be referred to for more detailed descriptions of the file concepts and elements.

2.1 Simulation Model Specification: File Structure

A DYMASOS simulation model source structure is a folder structure that contains (see Figure 5):

- A single XML configuration file,
- A root folder that serves as the base folder for all references to the contained components within the configuration file, and
- An arbitrary number of black-box and white-box components that can represent both, simulation model components and control algorithms. These components can be located in arbitrary subfolders to enable the user to structure the contents as desired.

The exemplary structure shown in Figure 5 contains three white-box *Modelica* model components, a co-simulation model component as a Functional Mock-up Unit file, and four different controller implementations (which, in this case, have been exported from *Matlab*).

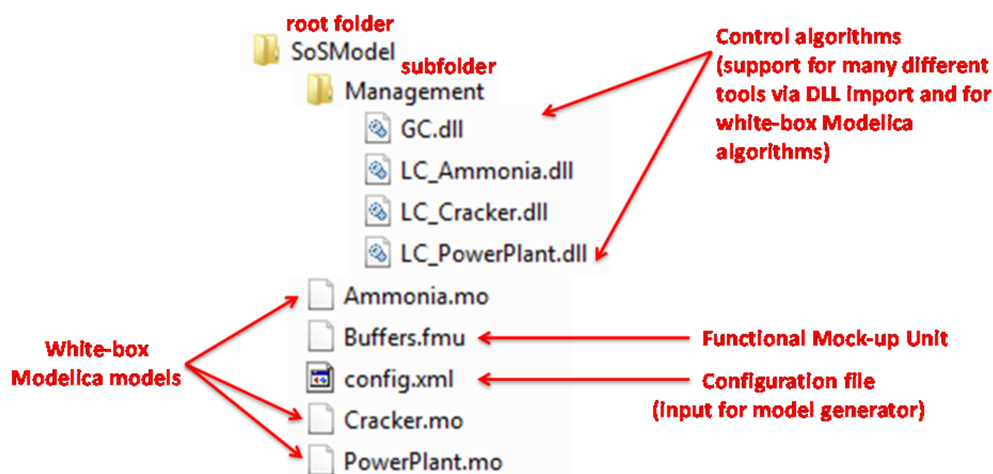

Figure 5: Example of a file structure of a DYMASOS simulation model specification.

During the model generation step, the DYMASOS simulation and validation framework generates a single *Modelica* model from this structure. To this end, it parses the configuration file, interconnects all of the model and algorithm components accordingly, and parameterizes all of the components (as well as the model management engine) according to the parameters that are provided in the configuration file.

2.2 Overview of the Configuration Specification

During the development of the configuration specification, several extensions and modifications have been made with respect to the initial, conceptual version that is described in [1]. The main features of the configuration specification are:

- **Event-driven communication and execution architecture:** During development of the configuration, we decided to employ a completely event-driven communication and execution architecture (in contrast to the request-reply-based architecture described in [1]), since it produces less overhead, and is a natural fit for the iterative execution of large automation architectures with many interconnected controller components.
- **Support for arbitrarily many sampling times and event-driven models:** Arbitrarily many sampling times can be defined for a CPSoS model. Here, sampling times are regular time intervals with discrete time instants at which a control system execution can be triggered. In addition, discrete events in validation model components can trigger the execution of connected control systems or other validation model elements. This enables, on the one hand, the implementation of complex, hierarchical automation architectures that often employ a large variety of different sampling times for different tasks and, on the other hand, the implementation of discrete-event validation models with or without continuous dynamics.
- **Hierarchical definition of execution delays:** Execution delays for control systems (i.e. the time spans that the executions of control systems take) can be specified in a hierarchical manner: If execution duration information is not available, the user can specify a global execution delay for each sampling time at which a control system is executed. This global information can be overwritten for each controller component with local execution delay information, or even with the request to the model management engine to measure the actual execution delay during execution.

The configuration specification contains three major sections:

```
<?xml version='1.0' encoding='us-ascii'?>
<!--DYMASOS MS Framework Configuration File, V7, - 24.02.201 -->
<ConfigFile MSFWVersion="0.1">

        <ComponentDefinitions>
            • • •
        </ComponentDefinitions>

        <SamplingTimes>
            • • •
        </SamplingTimes>

        <Structure>
            • • •
        </Structure>

</ConfigFile>
```

2.2.1 Component Definitions

The section *ComponentDefinitions* contains the definitions of all components in the CPSoS model. It supports the following component types:

- *ValModel*: A validation model component. The following types of components are supported:
    - White-box *Modelica* models
    - Black-box model as a "Functional Mock-up Unit" (FMU), implemented according to the FMI co-simulation standard
- *Controller*: A component that contains a control algorithm. The following types of components are supported:
    - White-box *Modelica* algorithms

- External functions that can be implemented in a variety of tools and programming languages (e.g. in *Matlab, C, C++, C#, Fortran*, …)

These component specifications contain a reference to the component file within the simulation model source structure (see above), all required parameterizations, and instantiations of the generic interfaces that are used to interconnect the model and algorithm components. Note that we have improved and simplified the interface definitions considerably over the versions described in [1], and we now define only four generic interfaces, see Figure 6.

Most of these interface definitions are optional for each component, and that every component can define several interfaces of a type, which makes it possible to connect a single validation model to several different control algorithms, and vice versa. In addition, the distinction between “global” and “local” control algorithms has been removed so that the user can define arbitrarily complex and versatile automation architectures.

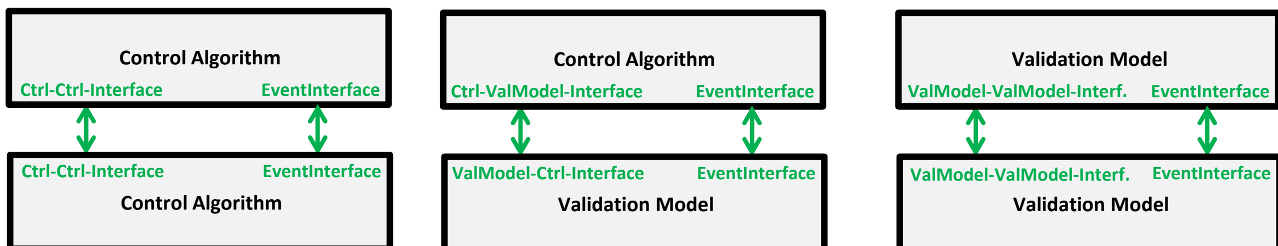

Figure 6: Revised interface definitions of the modelling and simulation framework.

The data entities that are defined within these interfaces can be specified in all *Modelica* types, including multi-dimensional types (e.g. vectors, matrices) and *Modelica* connector instantiations. The following interfaces can be instantiated in the components:

2.2.1.1 Generic Interface between Controllers

This interface with the XML type *Ctrl-Ctrl-Interface* can only be instantiated within *Controller* components. It supports the definition of continuous-valued and discrete-valued inputs and outputs of a *Controller* component that it can send to/receive from other controller components, as well as how these inputs and outputs are mapped to the input and output data structures that are provided to (or received from) the underlying external functions that implement the control algorithm.

2.2.1.2 Generic Interfaces between Controllers and Validation Models

These interfaces with the XML types *Ctrl-ValModel-Interface* (if instantiated in a *Controller* component) and *ValModel-Ctrl-Interface* (if instantiated in a *ValModel* component) define the data entities that can be exchanged between controllers and validation models.

The *ValModel-Ctrl-Interface* contains the definitions of continuous-valued and discrete-valued inputs and outputs of a *ValModel* component that it can send to/receive from controller components. Here, we have removed the separation between sampling-time-based variables and “truly” continuous variables, although both types of connections are still implicitly supported. The parameterization of this interface also defines how input and output variables are mapped to the variables of the underlying model component.

The *Ctrl-ValModel-Interface* is the counterpart to the *ValModel-Ctrl-Interface* interface on the side of *Controller* components and defines how the defined inputs and outputs are mapped to the input and output data structures that are provided to (or received from) the underlying external functions that implement the control algorithm.

2.2.1.3 Generic Interface between Validation Models

This interface with the XML type *ValModel-ValModel-Interface* can only be instantiated within *ValModel* components. It supports the definition of continuous-valued and discrete-valued inputs and outputs of a

ValModel component that it can send to/receive from other *ValModel* components. In addition, nondirectional input-output variables are supported within the *InOut* section that enable equation-based connections between validation models, i.e. connections in which the computation causality is not pre-defined.

2.2.1.4 Event Interfaces

This interface with the XML type *EventInterface* can be defined in both, *Controller* and *ValModel* components (although the *Controller* component version is slightly different to the *ValModel* version). It defines how to specify discrete input and output events.

Input events that are defined within a *ValModel* component map incoming discrete events into a rising (or falling) edge in a Boolean variable in the underlying model. Output event definitions contain Boolean guard definitions that can be defined as (1) a *Modelica* expression over the variables of the underlying model or (2) as a change in value of a Boolean variable in the underlying model.

Within a *Controller* component, input events define how to map incoming events into data structures that are passed to the control algorithm function before execution, while output events are mapped from a data structure that is returned from the control algorithm function after execution. In addition, a special final output event must always be defined. This event must be triggered by a *Controller* component to indicate to the model management engine that it has finished its iteration with other *Controller* components.

2.2.2 Sampling Times and Execution Delays

The section *SamplingTimes* contains definitions of the sampling times at which *Controller* components (or discrete-time *ValModel* components) can be executed. For each *SamplingTimes* element, a time value (in seconds) must be defined. In addition, a time offset value can be added which indicates how long after the start of the simulation the first sampling time instant will be triggered. Furthermore, a global execution delay value (in seconds) can be defined for each sampling time, which is an estimate of the execution duration of all *Controller* components that are executed at this sampling time. Note that this value can be overwritten, see section 2.2.3.

To ensure a deterministic, unambiguous execution of control architectures in which different controllers are executed iteratively, only a single *Controller* component is allowed to *subscribe* to a sampling time definition. This *Controller* component will then be executed first after a corresponding sampling time instant has been reached and can subsequently trigger executions of other *Controller* components via event communication. This restriction does not apply for discrete-time *ValModel* components, of which arbitrarily many can subscribe to a sampling time.

2.2.3 Structural Model Definition

The section *Structure* contains definitions of the interconnections between all of the *ValModel* and *Controller* components. Connections between variables of models and controllers are defined using the XML type *Connection*, and subscriptions to events or sampling times are defined using the XML type *Subscription*.

2.2.3.1 Connections between Validation Models

Variable connections between validation models are defined in the sub-section *ValModel-ValModel*. Two types of connections are supported:

- **Connections between directional input and output variables:** Here, connections are defined between input variables of one *ValModel* component and output variables of another *ValModel* component, in the form:

  ```
  <Connection> {ValModel1}.{inputvar1}, {ValModel2}.{outputvar1} </Connection>
  ```

- **Connections between nondirectional input-output variables:** Connections between nondirectional variables can involve more than two variables and may additionally indicate the type of the connection. The definition

  ```
  <Connection Type="flow"> {ValModel1}.{inoutvar}, {ValModel2}.{inoutvar},
  {ValModel3}.{inoutvar}  </Connection>
  ```

  describes a *flow* (sum-to-zero) *connection* between the three variables in which the overall sum of flows over the connection must be zero, and

  ```
  <Connection Type="potential"> {ValModel1}.{inoutvar}, {ValModel2}.{inoutvar},
  {ValModel3}.{inoutvar}  </Connection>
  ```

  represents a *potential connection* that enforces that all connected variables are of equal value.

2.2.3.2 Connections between Validation Models and Controllers

Variable connections between validation models and controllers are defined in the sub-section *ValModel-Ctrl* and only support connections between directional variables (see previous section).

2.2.3.3 Event and Sampling Time Subscriptions

The execution of control systems is controlled using discrete events. *Controller* components can subscribe to arbitrary output events of other *Controller* components and *ValModel* components, while *ValModel* components can only subscribe to other *ValModel* events. This restriction is introduced since control algorithms are, by definition, only executed at fixed time instants during which the simulation is halted and *ValModel* components cannot receive any events.

If an event is triggered, all subscribed components receive this event and can then request data from other components, use this data during their own execution, and trigger the execution of other *Controller* components. Thus, the user can straightforwardly implement arbitrary communication structures and iteration sequences between all *Controller* components. In subscription definitions, *Controller* components can additionally provide a value for their execution delay in the case that they are triggered via this event. This value then overwrites the global value that was defined in the *SamplingTimes* section, if any. Note that if a single Controller component overwrites the *ExecutionDelay* value, all connected controllers must provide *ExecutionDelay* values as well since otherwise, the model management engine cannot compute the overall delay that arises during the execution of the complete interconnected control system.

3 A Survey of Optimization Problem Formulations

One results of our literature survey of distributed coordination and optimization methodologies (see section 3.3.1 of [1]) is that many of these methodologies are based on specialized optimization problem formulations. Thus, as a basis for the development of a general optimization problem representation that covers not only the methodologies that are presented in [1], but also a more general class of optimization problems, we have extended our survey with selected optimization frameworks and tools that are known for dealing with very general optimization problems, see e.g. [3], with the goal to extract the most useful concepts from other languages. The following sections provide a summary of this survey.

3.1 *JModelica*

The main goal of *JModelica* [4], which was initiated by Lund University in 2007, is to increase design productivity by providing “high-level support for optimization in the *Modelica* language”. Although *Modelica* is a very rich language in terms of modeling, it lacks syntax elements for the formulation of optimization problems. To this end, the *Optimica* extension was designed for the high-level formulation of dynamic optimization problems based on *Modelica* models [4]. It adds syntax elements for representing objective functions, constraints, initial guesses, etc. The *JModelica* compiler is able to parse and flatten *Modelica* code as well as the *Optimica* extension, and to solve dynamic optimization problems specified in *Optimica*.

The *Optimica* extension includes several elements for the formulation of optimization problems. An optimization problem is defined in the `optimization` class which is a specialized *Modelica* class. However, the `optimization` class does not include all the features of a specialized *Modelica* class, e.g. it cannot be instantiated. One major restriction of this class is that it can only represent optimization problems that are solved offline and not during simulation, as would e.g. be necessary for the solution of model-predicive control (MPC) problems. The `optimization` class contains specialized attributes for the specification of objective functions, both inequality and equality constraints, and the start and final time of the optimization. In addition, it introduces new attributes for the built-in type *Real*: `initialGuess` and `free`.

JModelica uses the Python language for scripting. For this purpose, the compiler provides a number of Python packages to interact with *Modelica* models and optimization algorithms. *JModelica* does not provide *Modelica*-based interfaces to optimization algorithms yet.

The *JModelica* package *Assimulo* provides interfaces to DAE and ODE solvers (including the *SUNDIALS* solver suite). The *PyFMI* package provides import and export of Functional Mock-up Unit (*FMU*) co-simulation components. The *PyModelica* package interacts with *Modelica* and *Optimica* models by compiling the models into *FMUs* and *JMUs*, where the *JMU* format is a *JModelica*-specific format which is designed to follow the *FMU* file format [5]. The *PyJMI* package contains drivers for optimization algorithms. *JMI* is the *JModelica Model Interface* which is a *C* interface for the efficient evaluation of model equations, the cost function, the constraints, and derivatives, and is intended to offer a convenient interface for integration of numerical algorithms.

JModelica features four different algorithms for solving dynamic optimization problems. Three different algorithms are based on direct collocation which rely on the NLP solver *IPOPT* [6]. The default algorithm is coded in the *C* language and relies on the solver *CppAD* for computing derivatives. Two other algorithms are implemented in *Python* and rely on the dynamic optimization package *CasADi* [7]. A derivative-free algorithm is also supported for model calibration based on measurement data. For more information on the *JModelica* framework, please refer to [5].

3.1.1 *Formulation of an Optimization Problem in JModelica*

In order to illustrate the optimization problem formulation in *JModelica*, an optimization problem as well as its *Optimica* formulation is presented in the following. The example is extracted from the *JModelica* user manual [5].

An optimization problem with model constraints, final-time constraints, and bounds on the control inputs is given as:

$$\min_{u(t)} t_f$$

subject to the Van der Pol dynamics:

$$\dot{x}_1 = \left(1 - x_2^2\right) x_1 - x_2 + u, \quad x_1(0) = 0$$
$$\dot{x}_2 = x_1, \quad x_2(0) = 1$$

and the constraints:

$$x_1(t_f) = 0, \quad x_2(t_f) = 0$$

$$-1 \leq u(t) \leq 1$$

This problem can be formulated in *JModelica* as follows.

```
optimization VDP_Opt_Min_Time (objective = finalTime, startTime = 0,
finalTime(free= true,min= 0.2, initialGuess = 1))
      // the states
      Real x1(start = 0, fixed= true);
      Real x2(start = 0, fixed= true);

      // the control signal
      input Real u(free= true, min =-1, max= 1)

      equation
            //dynamic equations
            der(x1) = (1-x2^2)*x1 – x2 + u;
            der(x2) = x1;

      constraint
            // terminal constraints
            x1(finalTime)= 0;
            x2(finalTime)= 0;
end VDP_Opt_Min_Time;
```

The constructor of the `optimization` class, which includes the objective function formulation, the start time, and the final time, has to be placed directly after the class definition. The model equations have to be defined or instantiated within the `optimization` class. Since a minimum-time problem is being formulated in this example, the `finalTime` attribute of the objective function is set to `free`. The `initialGuess` is set to a fixed value. In difficult cases in which a constant value for the initial guess is not sufficiently expressive, the `initialGuess` can also be set to simulation profiles which are generated using initial guesses for input variables.

The *JModelica* compiler introduces an easy-to-use optimization extension which is based on *Modelica* models. However, solver interfaces were not integrated into the *Optimica* compiler due to extensive coding effort, as stated in [4]. The lack of a *Modelica*-based interface to optimization variables and instantiation of the model in the optimization problem are the two main drawbacks that our own general optimization problem formulation language aims to overcome, see section 4.

3.2 The Matlab Optimization Toolbox

The *Matlab Optimization Toolbox* offers solvers for wide variety of optimization problems. As an example, Rosenbrock's function is used [8]:

$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$
$$s.t.$$
$$x_1^2 + x_2^2 \leq 1$$

The objective function can be formulated as a *Matlab* function with optimization variables as inputs[1]. Equality and inequality constraints have to be formulated in the form $c(x) \leq 0$ or $ceq(x) = 0$. After reformulation, all the constraints have to be saved in a *Matlab* function, as shown in the following. If one of the equality or inequality constraints does not exist, an empty array has to be added as the constraint to the function.

$$function\ [c, ceq] = const(x)$$
$$c = x_1^2 + x_2^2 - 1;$$
$$ceq = [];$$

In order to execute the optimization, first a structure which represents the optimization options has to be created. In this structure, the user can specify details such as the desired solver, the algorithm, tolerances on the constraint violation, termination tolerance on the optimization variables and the objective function, and many other options. The user can also provide functions that compute the gradient, Jacobian and Hessian matrices/vectors of the objective function as well as the gradient of the constraints with respect to the optimization variables in the corresponding M-files and set the related flags in the options structure to 'on'. An exemplary function call that generated the options structure is (here, the solver *fmincon* is specified):

```
options = optimoptions(@fmincon,'Algorithm','interior-point',
'TolX', '1e-10', 'TolFun', '1e-12',
'DerivativeCheck','on','GradObj','on','GradConstr','on');
```

In the option structure, the first term represents the variable and the second term represents the value.

e.g. `Variable = 'DerivativeCheck'`

`Value = 'on'`

After setting the options, the solver is called including the `options` structure and all the information on the initial conditions as well as the linear and nonlinear equality and inequality constraints:

[1] Note that the details of the supported algorithms are not discussed since the problem formulation is the main concern here.

```
[X,FVAL,EXITFLAG,…]=FMINCON(@ObjFun,X0,A,B,Aeq,Beq,LB,UB,NONLCON,options)
```

For more details on the problem formulation and algorithms, please refer to [8].

3.2.1 Dynamic Optimization in Matlab

In [9], the method of orthogonal collocation over finite elements is developed and implemented in *Matlab* which extends the capabilities of the *Matlab Optimization Toolbox*, based specifically constrained nonlinear minimization routine *fmincon*. The function *dynopt* is the main function of the collection of functions in this framework. *dynopt* has a number of predefined inputs which have to be specified in the following order: the number of collocation points, the vector of initial lengths of *control vector parameterization* intervals, the final time, a matrix of control variable initial values, state variable lower and upper bounds [lbx ubx], control variable lower and upper bounds [lbu ubu], time interval bounds [lbdt, ubdt], the objective function which has to be implemented in an M-file, the function that computes the nonlinear equality and inequality constraints as an M-file, the function that models the targeted dynamic system, and a structure contaning optimization options which has the same structure as the one defined in the *Matlab Optmization Toolbox*.

The outputs of the *dynopt* function are the solution, the value of the objective function, an exit flag which specifies whether the algorithms has converged or exceeded the maximum number of function evaluations or iterations, an output structure which returns the information about the solution, such as the number of iterations, the number of function evaluations, the used algorithm, the step size, etc. Also, some information about the values of Lagrange multipliers and the gradient of the objective function is returned. In the following, the maximum number of outputs of the *dynopt* function as well as the inputs is shown:

```
[x,fval,exitflag,output,lambda] =
dynopt(ncol,delta_t,tf,uinit,bdx,bdu,bdt,objfun,confun,process,options)
```

If the final-time value is not specified, `tf` should be replaced with an empty vector: [].

The user can optionally include information about the gradient, Jacobian and Hessian to the M-files which include the mathematical formulation of the objective function, the cost function, and the process model. The corresponding flag has to be set to `'on'` in the `options` structure. For instance, when the user includes the gradient information into the objective function, the following has to be done:

```
options = optimset(options,'GradObj','on')
```

The information on the gradient of the objective function with respect to states, inputs, and time must be added to the M-file in the following order:

```
function [f,Dft,Dfx,Dfu] = objfungrad (t,x,u)

f = [...];      % compute the function value at t, x, u
Dft = [...];    % compute the gradient evaluate at t
Dfx = [...];    % compute the gradient evaluate at x
Dfu = [...];    % compute the gradient evaluate at u
```

The constraints are represented in the same way as in the *Matlab Optimization Toolbox*. The information on the gradient of the constraints can be optionally added to the M-file that includes the constraints.

The process model must be formulated using the following template:

```
function sys = process(t,x,flag,u)

switch flag,
    case 0
        sys = [...];    % right hand sizes of the ODE
    case 1
        sys = [...];    % jacobian with respect to x
    case 2
        sys = [...];    % jacobian with respect to u
    case 3
        sys = [...];    % jacobian with respect to t
    case 4
        sys = [...];    % initial values of state variables
    otherwise
        error(['unhandled flag = ',num2str(flag)]);
end
```

The corresponding M-files for the objective function, the constraints, and the process model take as inputs a time point *t* and a scalar/vector of state and input variables and return the function/constraint/model values.

3.3 gPROMS / gOPT

The *gOPT* extension of the process simulation tool *gPROMS* supports general dynamic and steady-state optimization, see e.g. [3]. It comes with two standard mathematical solvers for solving dynamic optimization problems. The first (default) solver implements a *control vector parameterization* (CVP) algorithm based on a single-shooting approach with both discrete and continuous decision variables (*mixed-integer optimization*). The second solver is an implementation of control vector parameterization based on a multiple-shooting approach.

The dynamic optimization facilities in *gOPT* support time-invariant, piecewise constant, and piecewise linear control inputs. These are by far the most commonly encountered problems in practical applications. However, if necessary, it is relatively straightforward to introduce several other types of control signals [10].

The optimization problem can be formulated using the *gPROMS* language or the *gPROMS* GUI. The following sections briefly outline how an optimization problem is formulated using the latter approach.

3.3.1 Definition of Cost Function, Process Model, and Control Vector Values

The process model that serves as a basis for the optimization has to be separately defined in the *gPROMS* process modeling environment and can be referred to in the optimization problem. The formulation of the objective function is not restrictive and could be either one of the differential variables, algebraic variables, or a separate term that is defined in the process model. The maximization or minimization of the objective function can be specified in the same window.

If dynamic optimization is chosen as the type of the optimization problem, the initial values on the number of time intervals and the initial guesses have to be specified in the time horizon section.

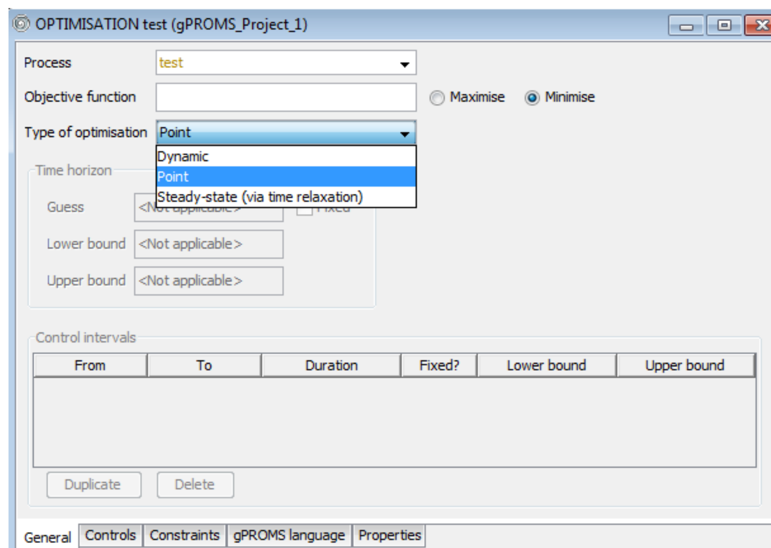

Figure 7: Definition of the process model, the cost function, the type of the optimization problem, and the discretization values of the control vector parameterization.

3.3.2 Definition of Variables

The optimization variables can be defined as shown in the following figure:

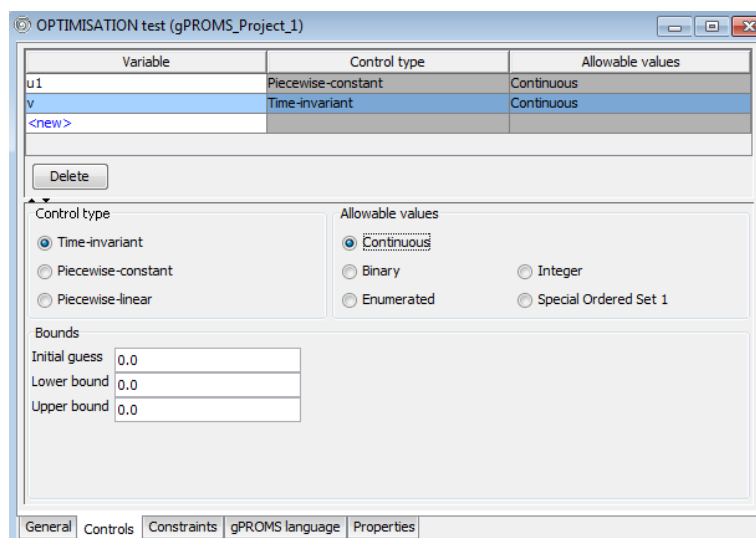

Figure 8: Deffintion of optimization variables using the gPROMS GUI.

The control types, values, initial guesses, and bounds of the variables are specified in the same window.

3.3.3 Definition of Constraints

Equality path constraints have to be added to the process model by converting one of the control variables u(t) into an algebraic variable y.

Both end-point and interior-point constraints are special cases of point constraints. However, for convenience, *gPROMS* treats them separately. It also treats any constraints that have to be satisfied at the initial time as interior-point constraints.

Figure 9: Definition of the constraints.

4 Definition of the General Optimization Problem Formulation Language

The main goal of the optimization problem language that we have defined for the DYMASOS simulation and validation framework is to decouple the optimization problem formulation from optimizing controllers that are connected to the framework, as this makes it straightforward to plug in new control algorithms without any customizations.

The optimization problem formulation language was developed as a general formalism that is based on *Modelica* and that is integrated into the model generator configuration specification that was described in section 2. It is based on our analysis of optimization languages and tools that is described in section 3, as well as on previous experience. Since these languages are comprehensive and allow for the definition of very general optimization problems, we are very confident that most of the optimization problems that arise in the practice of distributed management can be represented in our language, see e.g. our survey in [1], and we will, if necessary, adapt this language based on our experiences with the industrial DYMASOS case studies.

An optimization problem formulation can be directly integrated into the XML configuration file that was described in section 2. To this end, a dedicated, optional section is defined for each *Controller* component definition:

```
...
<Optimization>
        <![CDATA[
                optimization
                        ...
                end optimization
        ]]>
</Optimization>
...
```

Note that the optimization language is defined in a text-based format (instead of as an XML tree) since this approach makes it much easier for users to write and/or modify optimization problem formulations.

Our language supports a large variety of different problem formulations for steady-state and dynamic mixed-integer and hybrid optimization, with components such as:

- Linear and nonlinear discontinuous cost functions with Mayer and Lagrange terms,
- Linear and nonlinear discontinuous equality and inequality constraints,
- Both path and interior-point constraints for dynamic optimization, where constraints can be freely assigned to different time points (e.g. in a control vector parameterization approach), and
- Support for the symbolic definition of gradients, Jacobians, and Hessians.

In addition, numerous optimization parameters can be defined, such as (global or local) absolute and relative tolerances, variable scaling, maximum number of algorithm iterations, and user-defined values for specific optimization problem formulations.

The modeling and validation framework will provide control algorithms with two ways to access the optimization problem formulation:

- **Black-box access:** A dedicated interface to the model management engine (see [1]) will allow algorithms to obtain the values for the cost function, constraints, and derivatives at each stage during their execution. These values will be computed based on optimization variable values (or trajectories of such variables) provided by the algorithms.

- **White-box access:** If an algorithm requires symbolic access to the optimization problem entities (e.g. equations or (in)equalities), for example for automatic derivative generation or symbolic reformulation, it can access a text-based description of the optimization problem formulation.

During execution of an optimization, all variables that are defined in an optimization problem definition are **inputs** to the model management engine, i.e. their values must be provided by the control algorithm. The only **outputs** of the model management engine towards control algorithms are (a) numerical cost function values, (b) numerical values indicating the fulfilment of constraints, (c) the optimization options struct, and (d) numerical values of the symbolically defined gradients, Jacobians, and Hessians.

The black-box access mode is illustrated in the conceptual UML sequence diagram shown in Figure 10. Via the optimization interface, the model management engine initializes a control algorithm by sending the options structure and a callback function pointer that the control algorithm can use to request data. At each time that the algorithm needs access to any of the outputs described above, it requests these outputs via the callback function pointer and provides values for all optimization variables. The model management engine then computes and returns the requested values. Once the algorithm is finished with the current iteration, it returns execution to the model management engine.

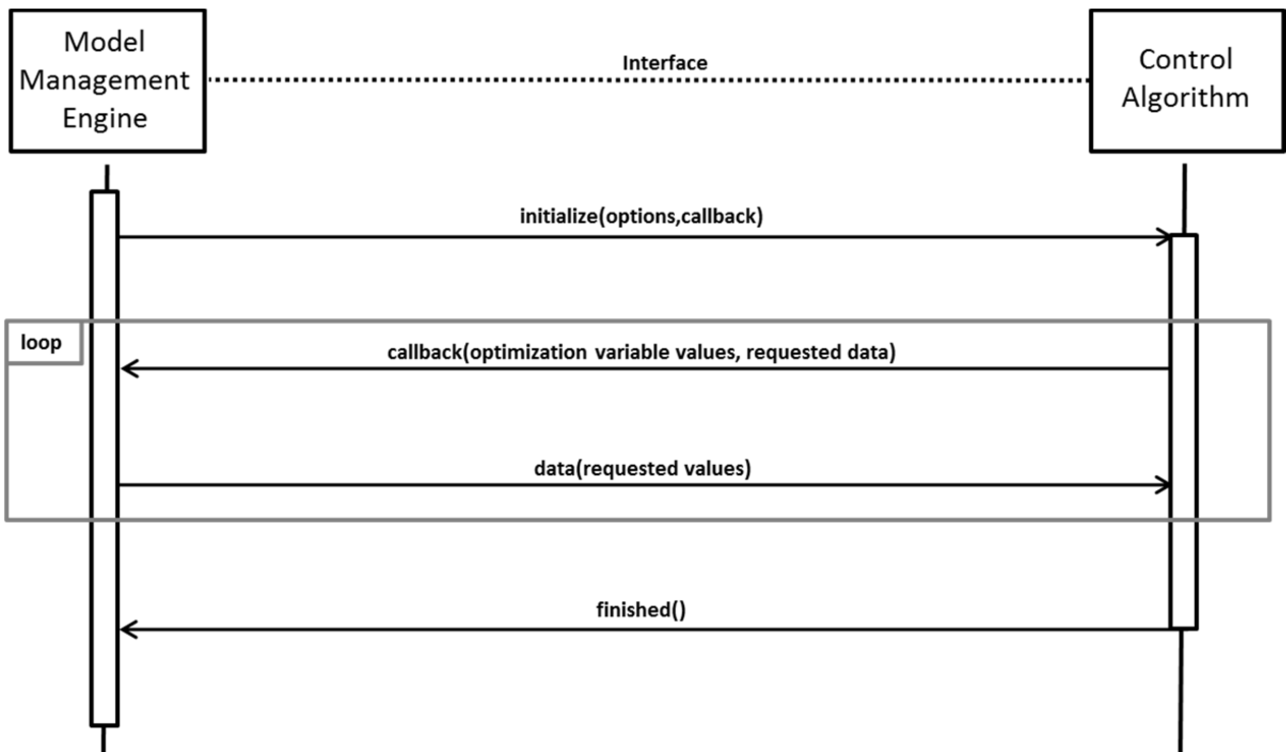

Figure 10: Conceptual UML sequence diagram illustrating the black-box access mode.

As in *JModelica*, an optimization problem is defined using the keyword `optimization.` Unlike the formulations that were discussed in the previous section, this class is defined independently of the process model and does not contain an instantiation of the design model. All the required information from the models is either provided internally by the control algorithm or is determined via the standard interfaces between the optimization framework and the design model(s).

A complete definition of the optimization problem language is provided in appendix B. In the following, the components of the language are described.

4.1 Variable Definitions

The user is free to either define optimization variables in one single definition or to use multiple definitions. All variables must be defined using *Modelica*-style definitions and can include all numerical variable types that *Modelica* supports (such as scalars, vectors, or matrices)[2]:

```
// Variable definitions
{Modelica type} optVars[…,…](lb = {...}, ub = {...});
{Modelica type} X[…,…](lb = {...}, ub = {...});
{Modelica type} Y[…,…](lb = {...}, ub = {...});
. . .
```

Here, predefined key words are given in **bold** text. Optionally, the user can define lower bounds (**lb**) and/or upper bounds (**ub**) for the variables.

For dynamic optimization formulations, the user can optionally specify trajectories of time instances that cover the time horizon of the optimization problem. For these instances, the integral part and time-instant-bound parts of the cost function can be evaluated, and interior-point constraints can be defined (see below).

```
// Time trajectory definitions for control vector parameterization
CVPTime[1](start = {0,1,3,5,...,50}, fixed = true);
CVPTime[2](start = {2,7,...}, lb = {...}, ub = {...}, fixed = false);
. . .
```

The **start** values represent the values of the time instances. If the flag **fixed** is equal to **true**, these values are fixed and cannot change during optimization. If the flag **fixed** is equal to **false**, these values are not fixed and can change during optimization (i.e. they are considered as optimization variables). Several **CVPTime** definitions can exist, thus fixed and non-fixed time instances can be mixed. In addition, the user can specify lower bounds (**lb**) and/or upper bounds (**ub**) for the time instants.

In addition, the user can define variable trajectories that correspond to the time instants defined above. A trajectory of values that corresponds to the values of a variable (e.g. a state variable or a controller output) can be defined as follows:

```
{Modelica type} timeVars1(CVPTime[1]);
{Modelica type} timeVars2(CVPTime[2]);
. . .
```

The **start time** of the optimization problem formulation is the smallest time instant that is contained in any of the **CVPTime** definitions, and the **final time** is the largest time instant that is contained in any of the **CVPTime** definitions. In the example above, the start time is equal to 0, the end time is equal to 50, and both time instants are fixed (i.e. they cannot be changed by an optimization algorithm).

Additional (constant) parameters, which might be used internally in the optimization problem formulation, can be added as well:

```
// Additional parameters
parameter {Modelica type} p1 = value1;
```

[2] Note that we do not enforce these definitions to represent model state variables, model input variables, or other optimization variables. The user is free to define these variables as he/she sees fit.

```
parameter {Modelica type} p2 = value2;
. . .
```

4.2 Definition of the Cost Function

The cost function definition of the language supports Mayer terms and, in the case of dynamic optimization, in addition a Lagrange (integral) term can be added. In addition, an initial guess for the cost function value can be provided:

```
costfun // costfun = MayerTerm + ∫_{t_0}^{t_f} LagrangeTerm dt
    initialGuess = ...;
    MayerTerm = {ModelicaExpr};
    LagrangeTerm = {ModelicaExpr}; // optional
```

Both terms must be defined as *Modelica*-style expressions and may include discontinuities (e.g. using the *if* keyword of *Modelica*). The terms can be based on any of the variables defined before, as well as time instants (such as the final time or interior time instants). During evaluation of the cost function, the model management engine will integrate the Lagrange term numerically over the optimization time range while the Mayer term will be computed directly. The addition of the values of both terms constitutes the cost function value for the provided values of the optimization variables.

In addition, the user can symbolically define the gradient vector of the cost function (i.e. the vector of first partial derivatives), as well as the Hessian matrix (i.e. the matrix of second partial derivatives). Alternatively, the user will be able to provide a definition for a *Modelica* external function call that returns the gradient and/or the Hessian. The user can in addition choose to define several different elements of the gradient and/or the Hessian in several different definitions:

```
    Gradient(optvars[0:7])  = [{ModelicaExpr};…]; // vector
    Gradient(optvars[8:10]) = [{ModelicaExpr};…]; // vector
    . . .

    Hessian(optvars[0:7])  = [{ModelicaExpr},{ModelicaExpr}; ...];
                                                         // matrix
    Hessian(optvars[8:10]) = [{ModelicaExpr},{ModelicaExpr}; ...];
                                                         // matrix
    . . .
end costfun;
```

4.3 Equality and Inequality Constraints

The constraints section is used to define equality andinequality *path constraints* (in the case of dynamic ptimization):

```
constraints
```

Note that while we do not enforce a separation between equality and inequality constraints, it is good practice by the user to separate the definition of these types of constraints since optimization algorithms might depend on this separation. The following types of constraints can be defined.

4.3.1 Nonlinear Equality and Inequality Constraints

The keyword **nl** must be used to create a vector of nonlinear constraints. There are several ways to define these constraints. The constraint **nl**[1] defines a nonlinear constraint as a (discontinuous) Boolean *Modelica* expression (that is valid along the entire time horizon in the case of dynamic optimization):

```
nl[1]: {BooleanModelicaExpr};
. . .
```

Two examples are:

```
nl[1]: optVars[1]*X[2] == 0; // nonlinear equality constraint

nl[1]: optVars[1]*X[2] <= 0; // nonlinear inequality constraint
```

In the case of a dynamic optimization problem, a time duration can additionally be specified during which the constraint must hold:

```
nl[2](start_time,end_time): {BooleanModelicaExpr};

. . .
```

Two examples are:

```
nl[2](0,35): optVars[1]*X[2] == 0; // nonlinear equality constraint
                           that is only enforced between 0 and 35 seconds.

nl[2](CVP_time[1][1],CVP_time[1][3]): optVars[1]*X[2] > 0;
                  // nonlinear inequality constraint that is only enforced
                  between two of the CVP time instants, as defined above.
```

4.3.2 Linear Equality and Inequality Constraints

Linear constraints in the form $Ax == b$ can be defined by the specification of the matrices **A** and **b**, the vector **x**, and the type of equality/inequality to be used. These constraints are defined by assigning the correct values to the corresponding keywords as in the following examples:

```
// The following specification defines Ax == b
lin_A[1]=[...]; // matrix
lin_b[1]=[...]; // vector
lin_x[1]=[X[1]; Y[2]; …]; // vector
lin_type[1]=eq; // lin_type can take the following values:
                           eq: equal
                           le: less than or equal
                           ge: greater than or equal
                           gt: greater than
                           lt: less than


// The following specification defines Ax < b over a reduced time
interval
lin_A[2]=[...]; // matrix
lin_b[2]=[...]; // vector
lin_x[2]=[Y[1]; X[2]; …]; // vector
lin_type[2]=lt;

lin_interval[2]=[start_time,end_time]
```

Additionally, the Jacobians (for several constraints) or gradients (for a single constraint) with respect to any set of optimization variables can be defined symbolically:

```
Jacobian(constraints,variables) = [{ModelicaExpr},…]; // matrix

Gradient(constraint,variables) = [{ModelicaExpr};…]; // vector
```

Two examples are:

```
    Jacobian(nl[1:3],optvars[1:8]) = [{ModelicaExpr},…]; // matrix
    Gradient(nl[1], optvars[2:3]) = [{ModelicaExpr};…]; // vector
end constraints
```

4.4 Interior-point Constraints

Interior-point constraints (including initial-time and final-time constraints) can be defined in the section

ip_constraints

Interior-point constraints must only hold true for specific time instants that must be specified during the constraint definition. The following types of constraints can be defined.

4.4.1 Nonlinear Interior-point Constraints

The keyword **ipnl** must be used to create a vector of nonlinear interior-point constraints. There are several ways to define these constraints. The constraint **ipnl**[1] defines a constraint as a Boolean *Modelica* expression that is valid at all of the specified time points:

```
ipnl[1](timeinst1,timeinst2,…): {BooleanModelicaExpr};
```

An example is:

```
ipnl[1](3,5,CVP_time[1][1],CVP_time[3][1]): optVars[1]*X[2] <= 0;
```

If an interior-point constraint should hold at all time instants in a vector, e.g. a CVP time instant vector (see above), this can be abbreviated:

```
ipnl[1](CVPTime[1]): optVars[1]*X[2] >= 0; // Must hold at all time
                                              instants defined in CVPTime[1]
```

4.4.2 Linear Interior-point Constraints

Linear interior-point constraints can be defined similary as in section 4.3.2 using the keywords shown in the following, with the addition that the time instants must be provided:

```
iplin_A[1]=[...]; // matrix
iplin_b[1]=[...]; // vector
iplin_x[1]=[X[1]; Y[2]; …]; // vector
iplin_type[1]=eq; // lin_type can take the following values:
                        eq: equal
                        le: less than or equal
                        ge: greater than or equal
                        gt: greater than
                        lt: less than
iplin_ti[1]=[timeinst1,timeinst2,…];
```

4.4.3 Initial-time and Final-time Constraints

Although initial-time and final-time constraints can be defined using the concepts described in the previous two sections, it is often easier to explicitly define these constraints. Nonlinear and linear initial-time constraints can be defined as:

```
init_nl[1]: {BooleanModelicaExpr};

init_lin_A[1]=[...]; // matrix
init_lin_b[1]=[...]; // vector
init_lin_x[1]=[X[1]; Y[2]; …]; // vector
init_lin_type[1]=eq; // lin_type can take the following values:
                        eq: equal
                        le: less than or equal
                        ge: greater than or equal
```

```
                        gt: greater than
                        lt: less than
```

Nonlinear and linear final-time constraints can be defined as:

```
      final_nl[1]: {BooleanModelicaExpr};

      final _lin_A[1]=[...]; // matrix
      final _lin_b[1]=[...]; // vector
      final _lin_x[1]=[X[1]; Y[2]; …]; // vector
      final _lin_type[1]=eq; // lin_type can take the following values:
                        eq: equal
                        le: less than or equal
                        ge: greater than or equal
                        gt: greater than
                        lt: less than

      end ip_constraints
end optimization
```

4.5 Optimization Options

All the relevant options for solving the optimization problem can be provided in the section **options**.

```
options
      max_iter = ...;  // maximum number of algorithm iterations

      abstol_constraints = ...; // absolute tolerance on the constraint
                              violation
      abstol_optvars = ...; // termination tolerance on the optimization
                              variables

      reltol_constraints = ...; // relative tolerance on the constraint
                              violation

      reltol_optvars = ...; // termination tolerance on the optimization
                              variables

      max_objfunc_eval = ...; // maximum number of obj function evaluation

      scaling_obj = ...; // scaling factor of the objective function

      scaling_optvars = {..., ... , ..., } //scaling factors for
                                          different optimization
                                          variables

      algorithm = ... // (ex.'interior-point' 'trust-region' '
                                                quasi-newton' ,...)
      collocation_method = ...

      ncol = ... // number of collocation points. Points at which a
            polynomial of a certain degree satisfies the initial condition and
            the differential equation of the PDE/ODE model at which the
            optimization is done

      fe_t = []; //vector of size (ni×1) - of initial length of the
            finite elements. The number of intervals that the ODE model is
            divided into and a number of collocation points are considered per
            element. The length of the elements can be also considered as
            optimization variables.

      // User-defined options

      user[1] = ...

      user[2] = ...
```

```
      user[3] = ...
      ...
end options
```

Other information about the optimization problem formulation can be generated automatically from the remainder of the problem definition, such as *problem type* (steady-state or dynamic) or *problem class* (LP, NLP, MINLP, QP, ...).

5 Conclusions and Future Work

In this report, two new advances in the development of the simulation and validation framework are presented, a concrete XML-based language for the specification of CPSoS simulation models, and a new language for the specification of general optimization problems.

Over the next months, we will integrate the configuration specification with the information platform developed by partner RWTH and will aim to devise a standardized version of the configuration specification and the general optimization problem formulation language that is based on established formalisms for automation systems exchange, such as *AutomationML* or *CAEX*.

Subsequently, we will and will implement both of these results into the simulation and validation framework prototype and will validate them using the industrial DYMASOS case studies.

Bibliography

[1] D. Kampert, S. Nazari and C. Sonntag, “DYMASOS Engineering Concept Specification,” Deliverable D4.1 for the DYMASOS Project, 2014.

[2] S. Nazari, C. Sonntag, G. Stojanovski and S. Engell, “A Modelling, Simulation, and Validation Framework for the Distributed Management of Large-scale Processing Systems,” in *Proc. 12th International Symposium on Process Systems Engineering and 25th European Symposium on Computer Aided Process Engineering (PSE/ESCAPE)*, 2015.

[3] C. Sonntag, “Modeling, Simulation, and Optimization Environments,” in *Handbook of Hybrid Systems Control - Theory, Tools, Applications*, Cambridge University Press, 2009, pp. 325-360.

[4] J. Akesson, “Languages and Tools for Optimization of Large-Scale Systems,” 2007.

[5] “JModelica.org User Guide, Version 1.15,” Modelon AB, 2014.

[6] S. Vigerske, A. Waechter, Y. Kawajir and C. Laird, “Introduction to IPOPT: a tutorial for downloading, installing and using ipopt.,” Carnegie Mellon University , 2015 .

[7] J. Andersson, J. Åkesson and M. Diehl, “Dynamic optimization with CasADi,” in *Proceedings of the 51st IEEE Conference on Decision and Control (CDC)*, 2012.

[8] “MATLAB Optimization Toolbox. User's Guide,” MathWorks , 2014.

[9] M. Cizniar, “Dynamic optimization of processes,” Diploma work, Slovak Technical University in Bratislava, Faculty of Chemical and Food Technology, Department of information Engineering and Process Control, 2005.

[10] “gPROMS Optimization Guide, Release v3.4.0.,” Process Systems Enterprise Limited , 2011.

A Example of a XML Configuration File

The following XML text is an example of a configuration file for the DYMASOS simulation and validation framework. A more formal definition of the corresponding XML language will be developed during the implementation of the prototype that processes this definition.

```
<?xml version='1.0' encoding='us-ascii'?>
<!--DYMASOS MS Framework Configuration File, V7, - 24.02.201 -->
<ConfigFile MSFWVersion="0.1">

	<!-- *** Global definitions *** -->

	<!-- The section "SamplingTimes" contains definitions of the sampling times at which
	the control system can be executed. A single "Controller" component can subscribe to
	each sampling time. This "Controller" component will then be the entry point (i.e. will
	be executed first by the model management engine -->
	<SamplingTimes>
		<Element>
			<!-- ID of this sampling time, mandatory (must be unique) -->
			<ID>GlobalST</ID>
			<!-- Sampling time value, mandatory (here: 15 minutes) -->
			<Value>900</Value>
			<!-- Sampling time offset (occurrence of first sampling time instant
			after start of simulation, optional: default value is "0" -->
			<Offset>30</Offset>
			<!-- Assumed execution delay for the execution of the control system at
			this sampling time, optional: default value is "0" (i.e. instantaneous
			execution is assumed by default).
			Note: This execution delay is only used if the controller components do
			not provide execution delays! -->
			<ExecutionDelay>5</ExecutionDelay>
		</Element>
		<Element>
			<!-- ID of this sampling time, mandatory (must be unique) -->
			<ID>LocalST</ID>
			<!-- Sampling time value, mandatory (here: 15 minutes) -->
			<Value>30</Value>
			<!-- Sampling time offset (occurrence of first sampling time instant
			after start of simulation, optional: default value is "0" -->
			<Offset>0</Offset>
			<!-- Assumed execution delay for the execution of the control system at
			this sampling time, optional: default value is "0" (i.e. instantaneous
			execution is assumed by default).
			Note: This execution delay is only used if the controller components do
			not provide execution delays! -->
			<ExecutionDelay>0.05</ExecutionDelay>
		</Element>
		...
	</SamplingTimes>

	<!-- The section "ComponentDefinitions" contains the definitions of all components in
	this SoS model. It supports the following component types:
		"ValModel": Validation model component. The following types of components are
		supported:
			*.mo: White-box Modelica model
			*.fmu: Black-box model as a "functional mock-up unit" (FMU), implemented
			according to the FMI co-simulation standard
		"Controller": Component of the automation/management system
			*.mo: Modelica algorithm
			*.dll/*.c/Fortran: External function (e.g. exported from Matlab)
	A component of type "ValModel" can have the following interface definitions:
		"ValModel-Ctrl-Interface": Interface to the management/automation system,
			optional.
		"ValModel-ValModel-Interface": Interface to other validation model components,
			includes undirected variables for equation-based modeling, optional.
		"EventInterface": Definition of discrete events the model sends and receives,
			optional.
	A component of type "Controller" can have the following interface definitions:
		"Ctrl-ValModel-Interface": Interface to a validation model, optional. Note that
			there can be more than 1 instance of this interface
		"Ctrl-Ctrl-Interface": Interface to other controller components, optional.
		"EventInterface": Definition of discrete events the model sends and receives,
```

```
                optional.
        -->
        <ComponentDefinitions>
                <!-- Definition of a "ValModel" component. -->
                <ValModel>
                        <!-- ID of component that can be referenced within the complete file
                        (must be unique), mandatory -->
                        <ID>PowerPlant</ID>
                        <!— Location of the model file (in this case: white-box Modelica model),
                        mandatory -->
                        <Location> /PowerPlant.mo </Location>
                        <ValModel-Ctrl-Interface>
                                <!-- Inputs (control actions if connected to a
                                controller) of the ValModel component via
                                this interface, optional -->
                                <Inputs>
                                        <!-- Continuous-valued inputs -->
                                        <CI>
                                                <!-- Each input definition is enclosed by
                                                the "Element" tag, mandatory -->
                                                <Element>
                                                        <!-- ID of the output that can be referenced
                                                        within the complete file (can be different to
                                                        the model variable), mandatory -->
                                                        <ID> u </ID>
                                                        <!-- Name of input in this model component,
                                                        optional: default value is ID == ModelRef -->
                                                        <ModelRef> u </ModelRef>
                                                        <!-- Type of this input, mandatory. Note that
                                                                (1) This might also be a Modelica
                                                                        type, as defined in the
                                                                        Modelica standard library, and
                                                                (2) This may also refer to a Modelica
                                                                        connector (which can be
                                                                        included in the SoS model
                                                                        definition, e.g. by another
                                                                        ValModel component)-->
                                                        <Type> Real[] </Type>
                                                        <!-- Dimensionality of the input in the form
                                                        "[dim1 (e.g. rows), dim2 (e.g. columns),
                                                        ...]", optional: default value is "1" -->
                                                        <Dimensionality> [2,2] </Dimensionality>
                                                        <!-- Indicates if this input must be
                                                        connected (i.e. if there is no default value
                                                        used in the model if this input is not
                                                        connected), optional: default value is "Yes"
                                                        -->
                                                        <Mandatory>Yes</Mandatory>
                                                </Element>
                                                • • •
                                        </CI>
                                        <!-- Discrete-valued inputs -->
                                        <DI>
                                                • • •
                                        </DI>
                                </Inputs>
                                <!-- Outputs (i.e. measurements if connected to a controller) of
                                the ValModel component via this interface, mandatory -->
                                <Outputs>
                                        <!-- Continuous-valued outputs -->
                                        <CO>
                                                <!-- Each output definition is enclosed by the
                                                "Element" tag, mandatory -->
                                                <Element>
                                                        <!-- ID of the output that can be referenced
                                                        within the complete file (can be different to
                                                        the model variable), mandatory -->
                                                        <ID> x </ID>
                                                        <!-- Name of input in this model component,
                                                        optional: default value is ID == ModelRef -->
                                                        <ModelRef> x </ModelRef>
                                                        <!-- Type of this output, mandatory. Note
                                                        that
                                                                (1) This might also be a Modelica
                                                                        type, as defined in the
                                                                        Modelica standard library, and
```

```
                              (2) This may also refer to a Modelica
                                    connector (which can be
                                    included in the SoS model
                                    definition, e.g. by another
                                    ValModel component)-->
                        <Type> Real[] </Type>
                        <!-- Dimensionality of the output in the form
                        "[dim1 (e.g. rows), dim2 (e.g. columns),
                        ...]", optional: default value is "1" -->
                        <Dimensionality> 4 </Dimensionality>
                  </Element>
                  ...
            </CO>
            <!-- Discrete-valued outputs -->
            <DO>
                  ...
            </DO>
      </Outputs>
</ValModel-Ctrl-Interface>

<ValModel-ValModel-Interface>
      <!—Inputs of the ValModel component via this interface, optional -
      ->
      <Inputs>
            <!-- Continuous-valued inputs -->
            <CI>
                  <!-- Each input definition is enclosed by the
                  "Element" tag, mandatory -->
                  <Element>
                        <!-- ID of the input that can be referenced
                        within the complete file (can be different to
                        the model variable), mandatory -->
                        <ID> m_s_5 </ID>
                        <!-- Name of input in this model component,
                        optional: default value is ID == ModelRef -->
                        <ModelRef> m_s_5 </ModelRef>
                        <!-- Type of this input, mandatory. Note that
                              (1) This might also be a Modelica
                                    type, as defined in the
                                    Modelica standard library, and
                              (2) This may also refer to a Modelica
                                    connector (which can be
                                    included in the SoS model
                                    definition, e.g. by another
                                    ValModel component)-->
                        <Type> Real </Type>
                        <!-- Dimensionality of the input in the form
                        "[dim1 (e.g. rows), dim2 (e.g. columns),
                        ...]", optional: default value is "1" -->
                        <Dimensionality> 1 </Dimensionality>
                        <!-- Indicates if this input must be
                        connected (i.e. if there is no default value
                        used in the model if this input is not
                        connected), optional: default value is "Yes"
                        -->
                        <Mandatory>Yes</Mandatory>
                  </Element>
                  ...
            </CI>
            <!-- Discrete-valued inputs -->
            <DI>
                  ...
            </DI>
      </Inputs>
      <!-- Outputs of the ValModel component via this interface,
      optional -->
      <Outputs>
            <!-- Continuous-valued outputs -->
            <CO>
                  <!-- Each output definition is enclosed by the
                  "Element" tag, mandatory -->
                  <Element>
                        <!-- ID of the output that can be referenced
                        within the complete file (can be different to
                        the model variable), mandatory -->
```

```
                        <ID> x </ID>
                        <!-- Name of output in this model component,
                        optional: default value is ID == ModelRef -->
                        <ModelRef> m_s_30 </ModelRef>
                        <!-- Type of this output, mandatory. Note
                        that
                                (1) This might also be a Modelica
                                    type, as defined in the
                                    Modelica standard library, and
                                (2) This may also refer to a Modelica
                                    connector (which can be
                                    included in the SoS model
                                    definition, e.g. by another
                                    ValModel component)-->
                        <Type> Real </Type>
                        <!-- Dimensionality of the output in the form
                        "[dim1 (e.g. rows), dim2 (e.g. columns),
                        ...]", optional: default value is "1" -->
                        <Dimensionality> 1 </Dimensionality>
                </Element>
                • • •
        </CO>
        <!-- Discrete-valued outputs -->
        <DO>
                • • •
        </DO>
</Outputs>
<!-- Input/output variables for equation-based acausal
connections, optional -->
<InOut>
        <!-- Continuous-valued input/output variables -->
        <CIO>
                <!-- Each output definition is enclosed by the
                "Element" tag, mandatory -->
                <Element>
                        <!-- ID of the input/output that can be
                        referenced within the complete file (can be
                        different to the model variable), mandatory -
                        ->
                        <ID> m_s_40 </ID>
                        <!-- Name of input/output in this model
                        component, optional: default value is ID ==
                        ModelRef -->
                        <ModelRef> m_s_40 </ModelRef>
                        <!-- Type of this input/output, mandatory.
                        Note that
                                (1) This might also be a Modelica
                                    type, as defined in the
                                    Modelica standard library, and
                                (2) This may also refer to a Modelica
                                    connector (which can be
                                    included in the SoS model
                                    definition, e.g. by another
                                    ValModel component)-->
                        <Type> Real </Type>
                        <!-- Dimensionality of the input/output in
                        the form "[dim1 (e.g. rows), dim2 (e.g.
                        columns), ...]", optional: default value is
                        "1" -->
                        <Dimensionality> 1 </Dimensionality>
                        <!-- Indicates if this input/output must be
                        connected (i.e. if there is no default value
                        used in the model if this input is not
                        connected), optional: default value is "Yes"
                        -->
                        <Mandatory>Yes</Mandatory>
                </Element>
                • • •

        </CIO>
        <!-- Discrete-valued input/output variables -->
        <DIO>
                • • •
        </DIO>
</InOut>
```

```
		</ValModel-ValModel-Interface>

		<EventInterface>
			<!-- Discrete events that the model accepts, optional. The event
			is propagated to the model via an edge of a Boolean input
			variable, i.e. a Boolean input variable is set to "true"/"false"
			(depending on "Trigger"). -->
			<InputEvents>
				<!-- Each event definition is enclosed by the "Element"
				tag, mandatory -->
				<Element>
					<!-- ID of the event that can be referenced in the
					complete file, mandatory -->
					<ID>Event_in</ID>
					<!—Boolean variable in the linked model that must be
					set when this event occurs, mandatory -->
					<Flag>EventFlag_in</Flag>
					<!-- Indicates how the event is propagated. If
					"High", the "Flag" variable is set to "true" at
					event occurrence, if "Low", the "Flag" variable is
					set to "false" at event occurrence, optional:
					default value is "High". -->
					<Trigger>High</Trigger>
					<!-- Dimensionality of the event input in the form
					"[dim1 (e.g. rows), dim2 (e.g. columns), ...]",
					optional: default value is "1" -->
					<Dimensionality> 1 </Dimensionality> <!--
				</Element>
				...
			</InputEvents>
			<!-- Discrete events that the model generates, optional. -->
			<OutputEvents>
				<!-- Each event definition is enclosed by the "Element"
				tag, mandatory -->
				<Element>
					<!-- ID of the event that can be referenced in the
					complete file, mandatory -->
					<ID>Event_out</ID>
					<!-- Guard expression that is used to trigger this
					event, mandatory. This may be:
						(1) Any Boolean Modelica expression over the
						    model outputs or
						(2) A Boolean model variable that generates
						    the event
					Note: There must be at least one "Guard" element,
					but there can be arbitrarily many. -->
					<Guard><![CDATA[P_el < 5]]></Guard>
					<!-- Indicates how the event is propagated. If
					"High", the "Flag" variable is set to "true" at
					event occurrence, if "Low", the "Flag" variable is
					set to "false" at event occurrence, optional:
					default value is "High". -->
					<Trigger>High</Trigger>
					<!-- Dimensionality of the event output in the form
					"[dim1 (e.g. rows), dim2 (e.g. columns), ...]",
					optional: default value is "1" -->
					<Dimensionality> 1 </Dimensionality> <!--
				</Element>
				...
			</OutputEvents>
		</EventInterface>
	</ValModel>

	<!-- Definition of a "Controller" component. Note that "Controller" components
	can communicate with other Controller components by event subscription. -->
	<Controller>
		<!-- ID of component that can be referenced within the complete file
		(must be unique), mandatory -->
		<ID>LC_PowerPlant</ID>
		<!-- Location of the file implementing the control algorithm (in this
		case: Matlab-based DLL implementation), mandatory -->
		<Location> /powerplant.dll </Location>
		<!-- External optimization problem definition, optional -->
		<Optimization>
			<![CDATA[
				optimization
```

```
                ...
            end optimization
        ]]>
</Optimization>
<!-- Definition of the functions within the controller executable that
implement the DYMASOS Controller interface, optional -->
<Parameters>
        • • •
</Parameters>

<Ctrl-ValModel-Interface>
        <!-- Inputs (measurements from a connected validation model) of
        the Controller component via this interface, optional -->
        <Inputs>
                <!-- Continuous-valued inputs -->
                <CI>
                        <!-- Each output definition is enclosed by the
                        "Element" tag, mandatory -->
                        <Element>
                                <!-- ID of the input that can be referenced
                                within the complete file (can be different to
                                the name of variable in the controller
                                implementation), mandatory -->
                                <ID> ZVar </ID>
                                <!-- Index of the input vector where this
                                input should be stored when invoking the
                                controller, mandatory -->
                                <Index> 1 </Index>
                                <!-- Type of this input, mandatory. Note that
                                        (1) This might also be a Modelica
                                                type, as defined in the
                                                Modelica standard library, and
                                        (2) This may also refer to a Modelica
                                                connector (which can be
                                                included in the SoS model
                                                definition, e.g. by another
                                                ValModel component)-->
                                <Type> Real[] </Type>
                                <!-- Dimensionality of the input in the form
                                "[dim1 (e.g. rows), dim2 (e.g. columns),
                                ...]", optional: default value is "1" -->
                                <Dimensionality> [2,1] </Dimensionality>
                                <!-- Indicates if this input must be
                                connected (i.e. if there is no default value
                                used in the controller if this input is not
                                connected), optional: default value is "Yes"
                                -->
                                <Mandatory>Yes</Mandatory>
                        </Element>
                        • • •
                </CI>
                <!-- Discrete-valued inputs -->
                <DI>
                        • • •
                </DI>
        </Inputs>
        <!-- Outputs (i.e. measurements if connected to a validation
        model) of the "Controller" component via this interface, mandatory
        -->
        <Outputs>
                <!-- Continuous-valued outputs -->
                <CO>
                        <!-- Each output definition is enclosed by the
                        "Element" tag, mandatory -->
                        <Element>
                                <!-- ID of the output that can be referenced
                                within the complete file (can be different to
                                the name of variable in the controller
                                implementation), mandatory -->
                                <ID> SX </ID>
                                <!-- Index of this output in the output
                                vector that is returned by the controller
                                implementation after execution, mandatory -->
                                <Index> 1 </Index>
```

```
                    <!-- Type of this output, mandatory. Note
                    that
                           (1) This might also be a Modelica
                                  type, as defined in the
                                  Modelica standard library, and
                           (2) This may also refer to a Modelica
                                  connector (which can be
                                  included in the SoS model
                                  definition, e.g. by another
                                  ValModel component)-->
                    <Type> Real[] </Type>
                    <!-- Dimensionality of the input in the form
                    "[dim1 (e.g. rows), dim2 (e.g. columns),
                    ...]", optional: default value is "1" -->
                    <Dimensionality> [4,1] </Dimensionality>
             </Element>
             • • •
      </CO>
      <!-- Discrete-valued outputs -->
      <DO>
             • • •
      </DO>
  </Outputs>
</Ctrl-ValModel-Interface>

<Ctrl-Ctrl-Interface>
  <!-- Inputs (from a connected controller) of the "Controller"
  component via this interface, optional -->
  <Inputs>
      <!-- Continuous-valued inputs -->
      <CI>
             <!-- Each output definition is enclosed by the
             "Element" tag, mandatory -->
             <Element>
                    <!-- ID of the input that can be referenced
                    within the complete file (can be different to
                    the name of variable in the controller
                    implementation), mandatory -->
                    <ID> prices </ID>
                    <!-- Index of the Ctrl-Ctrl input vector
                    where this input should be stored when
                    invoking the controller -->
                    <Index> 1 </Index>
                    <!-- Type of this input, mandatory. Note that
                           (1) This might also be a Modelica
                                  type, as defined in the
                                  Modelica standard library, and
                           (2) This may also refer to a Modelica
                                  connector (which can be
                                  included in the SoS model
                                  definition, e.g. by another
                                  ValModel component)-->
                    <Type> Real[] </Type>
                    <!-- Dimensionality of the input in the form
                    "[dim1 (e.g. rows), dim2 (e.g. columns),
                    ...]", optional: default value is "1" -->
                    <Dimensionality> [2,1] </Dimensionality>
                    <!-- Indicates if this input must be
                    connected (i.e. if there is no default value
                    used in the controller if this input is not
                    connected), optional: default value is "Yes"
                    -->
                    <Mandatory>Yes</Mandatory>
             </Element>
             • • •
      </CI>
      <!-- Discrete-valued inputs -->
      <DI>
             • • •
      </DI>
  </Inputs>
  <!-- Outputs (sent to another controller) of the "Controller"
  component via this interface, mandatory -->
  <Outputs>
      <!-- Continuous-valued outputs -->
```

```
        <CO>
            <!-- Each output definition is enclosed by the
            "Element" tag, mandatory -->
            <Element>
                <!-- ID of the output that can be referenced
                within the complete file (can be different to
                the name of variable in the controller
                implementation), mandatory -->
                <ID> SXB </ID>
                <!-- Index of this output in the Ctrl-Ctrl
                output vector that is returned by the
                controller after execution -->
                <Index> 3 </Index>
                <!-- Type of this output, mandatory. Note
                that
                    (1) This might also be a Modelica
                        type, as defined in the
                        Modelica standard library, and
                    (2) This may also refer to a Modelica
                        connector (which can be
                        included in the SoS model
                        definition, e.g. by another
                        ValModel component)-->
                <Type> Real[] </Type>
                <!-- Dimensionality of the output in the form
                "[dim1 (e.g. rows), dim2 (e.g. columns),
                ...]", optional: default value is "1" -->
                <Dimensionality> [4,1] </Dimensionality>
            </Element>
            ...
        </CO>
        <!-- Discrete-valued outputs -->
        <DO>
            ...
        </DO>
    </Outputs>
</Ctrl-Ctrl-Interface>

<EventInterface>
    <!-- Discrete events that the controller accepts, mandatory. An
    event that is received triggers an execution of the controller by
    the model management engine. Note that:
         (1) There must be at least one input event defined,
         because otherwise the controller cannot be executed.
         (2) The controller can subscribe to both, events by
         validation models and events by other controllers. -->
    <InputEvents>
        <!-- Each event definition is enclosed by the "Element"
        tag, mandatory -->
        <Element>
            <!-- ID of the event that can be referenced within
            the complete file, mandatory -->
            <ID>Event_in</ID>
            <!-- Index of this event in the input event vector
            that is sent to the controller before execution,
            mandatory. -->
            <Index>3</Index>
            <!-- Indicates how the event is propagated. If
            "High", the corresponding flag in the input event
            vector is set to "true" at event occurrence, if
            "Low", the corresponding flag in the input event
            vector is set to "false" at event occurrence,
            optional: default value is "High". -->
            <Trigger>High</Trigger>
            <!-- Dimensionality of the event in the form "[dim1
            (e.g. rows), dim2 (e.g. columns), ...]", optional:
            default value is "1" -->
            <Dimensionality> 1 </Dimensionality>
        </Element>
        ...
    </InputEvents>
    <!-- Discrete events that the controller generates, optional.
    Note: These events can only be connected to other controllers, NOT
    to validation models. -->
    <OutputEvents>
```

```
                                    <!-- Each event definition is enclosed by the "Element"
                                    tag, mandatory -->
                                    <Element>
                                           <!-- ID of the event that can be referenced within
                                           the complete file, mandatory -->
                                           <ID>Event_out</ID>
                                           <!-- Index of this event in the output event vector
                                           that is provided by the controller that indicates
                                           tjat this event must be sent, mandatory. -->
                                           <Index>1</Index>
                                           <Dimensionality> 1 </Dimensionality>
                                    </Element>
                                    ...
                                    <!-- Predefined output event with which the controller
                                    indicates that it has finished execution in the current
                                    iteration, mandatory. -->
                                    <FinalEventIndex>2</FinalEventIndex>
                             </OutputEvents>
                      </EventInterface>
               </Controller>
        </ComponentDefinitions>

        <!—Definition of interconnections of validation models and controllers -->
        <Structure>
               <!-- Variable connections (validation model <-> validation model), including
               between input/output variables, optional -->
               <ValModel-ValModel>
                      <!-- Connections are defined using the "Connection" tag. In variable
                      connections between input/output variables in validation models, the
                      property "Type" indicates the connection type in Modelica style (either
                      flow connection or potential connection) -->
                      <Connection Type="flow"> PowerPlant.m_s_5, Tank_5bar.inflow </Connection>
                      <Connection Type="potential"> PowerPlant.m_s_30, Tank_30bar.inFlow,
                      Tank_5bar.outFlow </Connection>
                      <!-- Example: a list of variables are be connected: -->
                      <Connection Type="flow"> {x_1[1], x_1[3], x_1[2], x_1[4]}, {x_2[4],
                      x_2[1], x_2[3], x_3[1]} </Connection>
                      ...
               </ValModel-ValModel>

               <!-- Variable connections (validation model <-> controller), directional
               connections only -->
               <ValModel-Ctrl>
                      <!-- One of the two variables within a directional connection must be an
                      input variable, the other one must be an output variable. -->
                      <Connection> PowerPlant.m_s_5, GC.inflow </Connection>
                      ...
               </ValModel-Ctrl>
               <!-- Event and sampling time subscriptions, optional.
               The following example tells the model management engine to execute the control
               algorithm "GC" at the "GlobalST" sampling time (i.e. GC will be the first
               controller executed at this sampling time) -->
               <Subscription>
                      <!-- Source event of the subscription definition, mandatory (in this
                      case: a global sampling time) -->
                      <Source>GlobalST</Source>
                      <!-- Target event of the subscription definition, mandatory (in this
                      case: the input event "Event_In" of the control algorithm "GC" -->
                      <Target>GC.Event_In</Target>
                      <!-- The tag "ExecutionDelay" can be used to override the global
                      execution delay definition and tells the framework that it should measure
                      the exeuction time during controller execution, optional.
                      NOTE: If one subscription overrides the global exeuction delay, all
                      subscriptions by controllers must override the global execution delay. --
                      >
                      <ExecutionDelay>Measured</ExecutionDelay>
               </Subscription>
               <Subscription>
                      <Source>GC.Event_Out</Source>
                      <Target>LC_PowerPlant.Event_In</Target>
                      <!-- Data entities to be sent via this subscription (subsets of the
                      corresponding Ctr-Ctrl-Interfaces), mandatory -->
                      <Data> ... </Data>
```

```
                        <!-- The tag "ExecutionDelay" can be used to override the global
                        execution delay definition and tells the framework that it should measure
                        the exeuction time during controller execution, optional.
                        NOTE: If one subscription overrides the global exeuction delay, all
                        subscriptions by controllers must override the global execution delay. --
                        >
                        <ExecutionDelay>Measured</ExecutionDelay>
                </Subscription>
                <!-- Event subscriptions between validation models -->
                <Subscription>
                        <Source>ValModel1.Event_Out</Source>
                        <Target>ValModel2.Event_In</Target>
                </Subscription>
                ...
        </Structure>
</ConfigFile>
```