Modelica for Embedded Systems

Hilding Elmqvist[1], Martin Otter[2], Dan Henriksson[1], Bernhard Thiele[2], Sven Erik Mattsson[1]
[1] Dassault Systèmes, Lund, Sweden (Dynasim)
[2]German Aerospace Centre (DLR), Institute for Robotics and Mechatronics, Germany
Hilding.Elmqvist@3ds.com, Martin.Otter@DLR.de, Dan.Henriksson@3ds.com,
Bernhard.Thiele@DLR.de, SvenErik.Mattsson@3ds.com

Abstract

New language elements are introduced in Modelica 3.1 to facilitate use Modelica models in embedded systems, e.g., as controllers. Models can be conveniently configured by marking the borders of the respective controller parts and by defining the mapping of the marked parts to target processors and target tasks.

This approach allows to define a "logical" model from which all different "real" controller configurations for Model-, Software-, Hardware-in-the-Loop (MiL, SiL, HiL), rapid prototyping, and production code for multi-processing/multi-tasking are automatically derived by setting configuration options. Furthermore, a new, free library - Modelica_EmbeddedSystems - is presented that provides a convenient user interface to the new language elements. In summary, the power of Modelica in the area of real-time control is improved significantly.

Keywords: Embedded systems, real-time control, multi-tasking, multi-core, multi-rate, model parallelization, Model-in-the-Loop, Software-in-the-Loop, Hardware-in-the-Loop, rapid prototyping.

1 Introduction

Modelica has been used in advanced controller applications for embedded systems for several years, especially when non-linear plant models are part of the control system *(Looye et. al. 2005)*, such as non-linear control systems for aircrafts *(Bauschat et. al. 2001),* for industrial robots *(Thümmel et. al. 2005),* or for power plants *(Franke et. al. 2008)*.

Within the ITEA2 EUROSYSLIB project, a major effort started at end of 2007 to enhance Modelica considerably in the area of model-based control. This effort is also carried on in the ITEA2 MODELISAR project that started during 2008.

Existing methods and tools have been analyzed and different designs have been performed. Typically, controller parts that are to be downloaded to target platforms are defined by the root of a hierarchical structure. However, this standard approach has several inherent limitations and therefore, a novel, new approach was developed where only the borders of control systems are marked and algorithms have been developed to deduce the controller code from this information. Furthermore, in principal any Modelica model is supported, and therefore a controller to be downloaded to a real-time target machine may contain non-linear differential-algebraic equation systems as needed for advanced control systems.

Modelica extensions have been designed and are included in version 3.1 of the Modelica Specification *(Modelica 2009)*. A free library "Modelica_EmbeddedSystems" has been developed as a convenient user interface to the new language elements. A prototype implementation in Dymola *(Dymola 2009)* was performed to validate the concept. Furthermore, device drivers for Windows game controllers, I/O boards using the Comedi-Interface *(Comedi 2009)* on real-time Linux and CAN-bus have been implemented by DLR. Device drivers for dSPACE (www.dspaceinc.com) hardware and the Lego Mindstorms NXT platform (mindstorms.lego.com) have been implemented by Dynasim, and the concept has been used in a student project *(Akesson et. al. 2009)*.

2 Logical and Technical System Architecture

A model of an embedded system is composed of subsystems which may have local controllers. The subsystems are coupled physically and through con-

trol systems (e.g. with buses). The notation from *(Schäuffele and Zurawka 2005)* is used:

Complex controllers, e.g., in vehicles, are first designed with an abstract view which is termed "logical system architecture". Here all the functional and logical behavior of the control system is defined. In a second step this architecture is mapped to a "technical system architecture" which is the concrete implementation of the control system in several tasks on several micro-controllers inter-connected by communication buses and other communication methods. For complex control systems, as in vehicles with over 60 ECUs interconnected via different bus systems, it must be possible to map from the logical to the technical architecture in a very flexible way. The new Modelica language extensions have been designed to fulfill this demanding requirement.

An already sufficiently complicated, but still rather simple, logical system architecture of a robot is shown in the left part of Figure 1. Every "axis" of the robot has a local control system in addition to the continuous motor and gearbox models. All local controllers are connected to a global controller (at the top of the figure) via a control bus. This system has eight coupled controllers that shall be downloaded to different processors (e.g., all axes controllers on two signal processors and the global controller on another processor and the processors communicate via buses). An example is shown in the right part of Figure 1 where this mapping of the logical to the technical system architecture is sketched. The new method has now the following important properties:

1. The user is not forced to manually assemble the parts belonging to the technical system architecture for download to the ECUs. Instead a Modelica translator performs this automatically from the logical system representation, given information about the mapping to the technical system architecture. Note, with standard methods and tools this is not possible, because the logical system architecture would be destroyed if the user is forced to move all controller parts under a hierarchical structure.
2. The mapping to the technical system architecture can be defined without modifying the logical system architecture. This is performed by a new model that inherits (extends) from the logical system model and where the mapping information is given as modifier, including the selection of hardware drivers. The latter are defined via replaceable external objects.

With these two features it is possible to conveniently configure different use cases, such as:

- Model-in-the-Loop (MiL) simulation
  (Plant: variable step size integrators. Controller: ideal, synchronous continuous or discrete controllers)
- Software-In-the-Loop (SiL) simulation
  (Plant: variable step size integrators. Controller: non-ideal, asynchronous controllers with modeled latencies).
- Rapid prototyping (real-time)

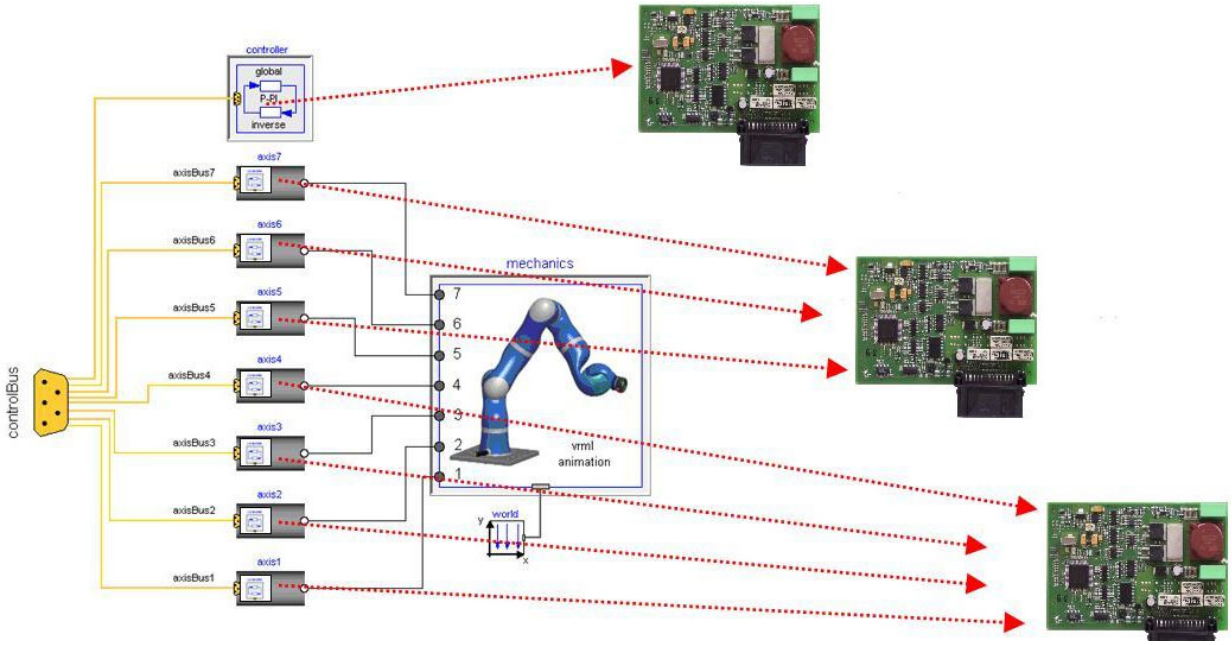


Figure 1: Mapping of logical to technical system architecture for a robot control system.
(Displayed ECUs adapted from http://commons.wikimedia.org/wiki/File:KeylessGoSiemensVDO.jpg)

(Plant: physical prototype. Controller: asynchronous controllers, channel assignment).

- Hardware-In-the-Loop (HIL) simulation (real-time)
  (Plant: fixed step size, multi-rate integrators. Controller: embedded in ECUs, multi-tasking, production code, fixed-point representation, channel assignment, bus communication).
- Production code (real-time)
  (Plant: Real product. Controller: embedded in ECUs, multi-tasking, production code, fixed-point representation, channel assignment, bus communication).

Dealing with embedded systems in Modelica according to the sketched concept above consists of the following parts:

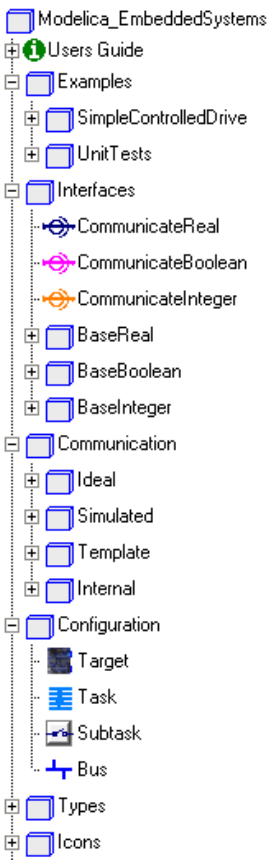
- New Modelica language elements are introduced. This is described in section 4. Basically, a new annotation "mapping" and some new built-in operators are provided. Furthermore, new algorithms are sketched to deduce the controller code from the logical system architecture.
- A new library "Modelica_EmbeddedSystems" is offered that provides a convenient interface to the new language elements. This library is discussed in the next section. It is provided freely and it is planned to be included in the Modelica Standard Library (note: a user may provide its own interface to the new language elements).
- Hardware drivers as Modelica external objects to access hardware from a Modelica model. A few hardware drivers that are available on every PC/notebook are provided in the Modelica_EmbeddedSystems library freely for Windows and for Linux. Other hardware drivers will be provided from third parties (e.g., commercially from tool vendors).

3 Modelica_EmbeddedSystems

The Modelica_Embedded-Systems library is available as an open source library from www.modelica.-org/libraries and will be the basis of embedded systems in Modelica. The current status of the library is shown in the screenshot to the right.

The `Examples` sub-library contains various use cases to demonstrate the usage of the library. Only some of the available examples are currently included in the library.

The `Interfaces` sub-library contains the basic components to define communication points (i.e., borders of controller parts) and to select the implementation of the actual communication in the embedded system. This can be external I/O, network communication or inter-task communication on the same ECU.



The `Communication` sub-library contains open source drivers for simulated communication (ideal and with simulated quantization effects taken into account), as well as a simple template for hardware drivers.

The `Configuration` sub-library contains templates to define the configuration of the embedded target systems (tasks, subtasks, sampling, target processor, etc.).

`Types` and `Icons` are utility sub-libraries.

The major goal of the library is to define the splitting of a model in tasks and subtasks and to associate device drivers with input and output signals of the respective parts. The following notation is used:

A "task" identifies a set of equations that are solved together as one entity, so equations are sorted and solved in a "synchronous" way as usual in Modelica. There are no equations that relate variables from different tasks because communication to and from tasks is performed by function calls of Modelica ExternalObjects[1]. Different tasks are executed asynchronously with possible synchronization via the ExternalObjects used for communication and possibly running on different cores or processors. Typically, a Modelica task is mapped to a task of the underlying operating system.

A "subtask" identifies a set of equations inside a task that are executed in the same way within the subtask with regards to sampling and integration method: If a subtask has continuous equations, all these equations are solved with the same integration method. Different subtasks can use different integration methods, e.g., fixed or variable step size methods of different orders. If a subtask is sampled, it is activated at the sampling instants and the equations of the subtask are integrated from the time instant of

[1] A Modelica ExternalObject defines a Modelica interface to C-functions that operate on the same memory and have constructor and destructor functions for this memory.

the last sample instant up to the current sample instant using the defined integration method. If a subtask is running on a real-time system, usually real-time integrators are utilized like explicit fixed-step solvers. The equations of several subtasks in the same task are automatically synchronized via equation sorting.

3.1 Communication Blocks

The usage of the Modelica_EmbeddedSystems library will be demonstrated by the very simple use case from Figure 2. The model consists of a reference controller ("ramp"), a feedback controller ("feedback" and "PI") and a plant ("torque", "load" and "speedSensor"). The task of the controller is to control the speed of the load inertia.

The model is split in different partitions by placing "communication blocks" in the signal paths. This is the "logical" model. At this stage it is only defined how the model is split into different parts, but it is not yet defined how to handle these parts (or more precisely, by default all communication blocks just pass their input signal to their output).

A "target" model is derived by inheriting from the model and by applying modifiers on the communication blocks. These modifiers usually reference a "configuration block" (in Figure 2 this is called "comedi") where all details are defined how to map this model to one or more target machines.

The "communication blocks" are the central part of the Modelica_EmbeddedSystems library and provide a graphical user interface between the user and the new Modelica 3.1 language elements. Clicking on one of the communication blocks in Dymola gives the menu shown in Figure 3.

The most important option is the first entry "communicationType". It defines the communication that shall take place between the input and the output of the communication block:

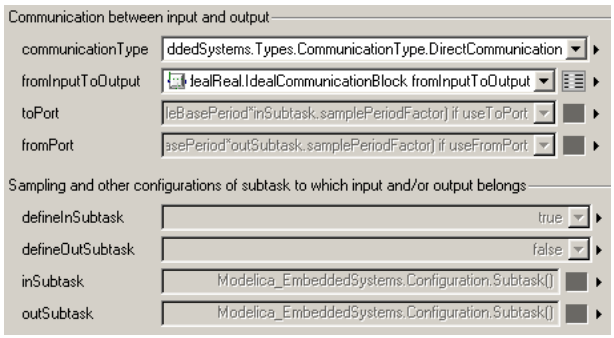


Figure 3: Menu of a communication block.

- Direct communication with Modelica equations (simplest case: y = u). This is mainly used to start and have a meaningful default, and/or to test some controller effects like noise or signal delays.
- Communication between two subtasks. This defines that the input and output signals are in different subtasks. All properties of these subtasks can be configured with the rest of the options, e.g., that the input subtask is periodically sampled with a defined sampling rate.
- Communication between two tasks. This defines that the input and output signals are running in different tasks on the same machine. All properties of the tasks can be configured with the rest of the options, e.g., in which way the communication between the tasks takes place (e.g., via shared memory).
- Communication to a port. This defines that the input signal is sent to an I/O board or to a bus (like the CAN bus). In this case, the communication block has no output signal. All properties of the I/O board can be configured with the rest of

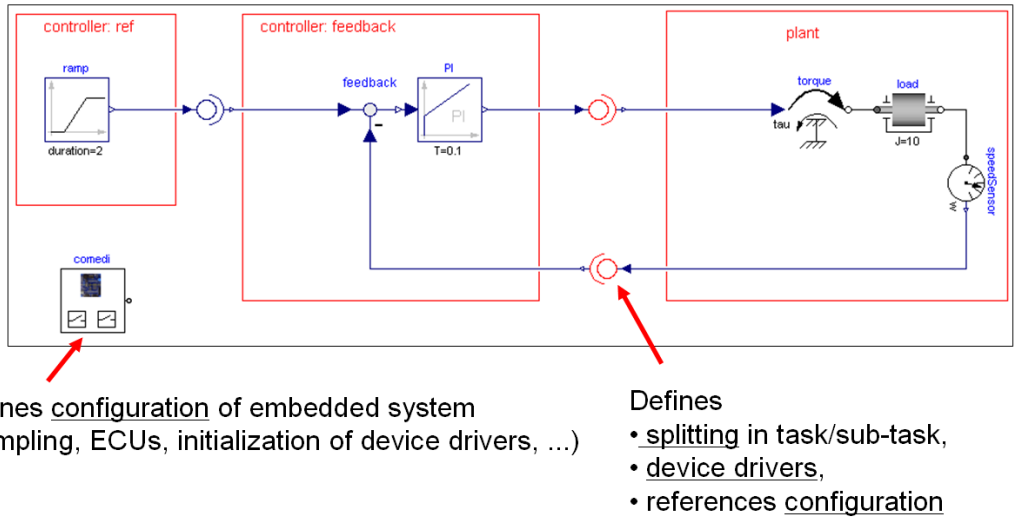


Figure 2: Simple drive train with two controller parts and three communication blocks.

the options, as well as the task/subtask properties of the equations that generate the signal to be sent to the I/O board.

- Communication from a port. This defines that the output signal is received from an I/O board or from a bus. In this case, the communication block has no input signal. All properties of the I/O board can be configured with the rest of the options, as well as the task/subtask properties of the equations that use the received signal.

Depending on the selected option, input fields of the parameter menu are enabled. These are mostly replaceable models using Modelica ExternalObjects. For example, when clicking on "toPort", all currently loaded device drivers to send a signal to an I/O board are listed. Selecting the desired device driver and then clicking on the "table" symbol to the right of the menu, opens the driver specific menu to configure this particular device.

Whenever a user introduces a device driver that is derived by inheritance from one of the partial models defined in Modelica_EmbeddedSystems.Interfaces (like Interfaces.BaseReal.PartialWriteRealToPort), this device driver is automatically included in the corresponding list of the communication block, due to the "choicesAllMatching" annotation defined in this block.

The important point is that all these hardware configuration settings can be made without copying the "logical model" and modifying it, but just inheriting from it and applying modifiers on the communication blocks.

When clicking on "inSubtask", the subtask/task/target properties of the model part can be defined that is connected to the input of the communication block (in a similar way, the properties of the output can be defined with "outSubtask"). In Figure 4 a typical screen shot is shown:

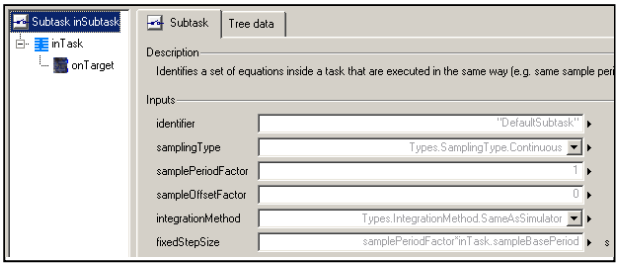


Figure 4: Defining subtask properties of the input signal of a communication block.

This is a hierarchical structure where all properties of a mapping annotation can be defined (for details see section 4). In Figure 4, the top-most hierarchical dialog level is shown to define the identifier, the sampling properties and the integration method of the subtask in which the input signal is running. The subtask identifier is a string that must be unique within a task. If the same subtask shall be referenced in different communication blocks, identical subtask identifiers must be given.

The second level of the dialog is shown in Figure 5, to define the task identifier, the task priority, the basic sample period (if the task is periodically sampled) and the core, if the task is running on a multi-core machine.

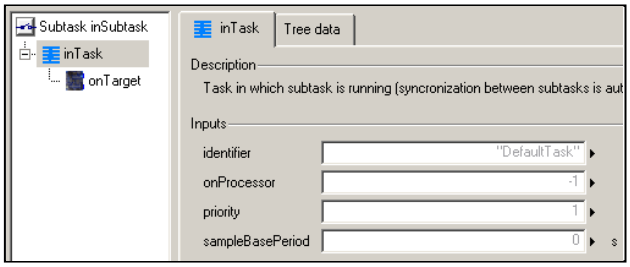


Figure 5: Defining task properties.

Finally, the third level of the dialog (shown in Figure 6) is used to define the identifier of the target on which the task is running and the "kind" of the target. The kind property will identify in a tool-specific way all properties of the target machine that need to be known for the code generation (e.g. if the target has or does not has a floating point unit).

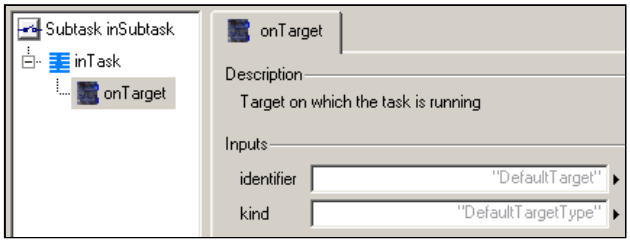


Figure 6: Defining target properties.

From the information provided in the communication blocks, it is possible to partition all equations of the flattened Modelica model in to the desired pieces. E.g., in the example above, two different C-codes are generated, one for the two controller parts and one for the plant and both are running in different tasks. The reference and the feedback controller are running with different sampling rates in the same task (sub-sampling).

The algorithm to determine the equations that belong to the different parts is sketched in section 4.4. In short, the BLT-algorithm (usually used by Modelica translators to determine the sorting of the equations) must be applied two or three times additionally on the model equations. So, a Modelica tool vendor has the basic algorithm already and must just apply it in some variants.

Note, subtask properties (like sampling period) may be defined at the input and/or at the output of a

subtask. If a subtask has several inputs and/or several outputs it is sufficient to define this information only at one location. For example in the use case of Figure 2, the properties of a subtask are defined at the output signal of the respective subtask.

3.2 Configuring Subtasks, Tasks, Targets and Devices.

All information to configure the subtasks, tasks and targets could be given in the hierarchical menus of the communication blocks shown in Figure 4, 5, and 6. However, this has a significant drawback: In the example above, the same target properties have to be defined three times (for all three different subtask definitions) and the task properties of the controllers have to be defined twice (for the "reference" and for the "feedbackController" subtask). Even more critical is the configuration of the device drivers. For example, an IO board is typically initialized once and then channel assignment takes place. It is difficult to define such initialization processes in the communication blocks.

In order to avoid redundant definitions and to have a simple way to initialize the device drivers, it is recommended to define all configuration options at one place once on the top level of the control system. An example is block "comedi" in the lower left part of Figure 2, where the control system is configured for real-time Linux using the comedi device drivers *(Comedi 2009)*.

This block consists of record instances to define subtask, task and target properties of all parts of the control system and to initialize the used device drivers. For example, Modelica_EmbeddedSystems.-Configuration.Subtask is defined as:

```
record Subtask
   parameter Task inTask = Task();
   parameter identifier  = "Default";
   ...
end Subtask;
```

The first parameter in the record is "`inTask`" which is an instance of record "`Task`". In order to automatically have a hierarchical menu built up, a default value of "`Task()`" is given, i.e., the record constructor of record "`Task`" is called. When using the Subtask record in a configuration block, the "task" properties are defined in an instance of record "Task" (called "controller" in Figure 7). In the subtask definitions, like "slowSampler" and "fastSampler" in Figure 7, parameter "inTask" is defined as the instance name of the record task ("controller" in Figure 7). By this technique, the configuration is defined in a non-redundant way.

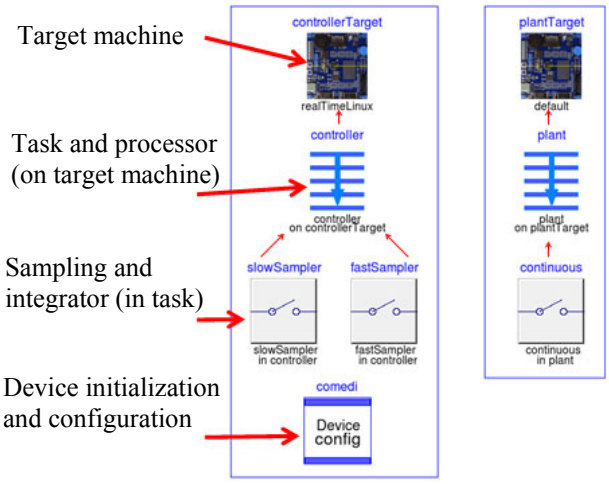


Figure 7: Configuration of use case for real-time Linux with the comedi device drivers.

Device drivers are record instances where device-specific configuration options are given and a final parameter is used for the device handle that is defined by a call to the constructor of the respective ExternalObject.

Finally, a hierarchical modifier is used to reference the configuration record instances at the appropriate places in the communication blocks. For example, parameter "inSubtask" in the communication block at the output of the feedback controller in Figure 2 is defined as "comedi.fastSampler". In Dymola this can be conveniently defined by just clicking on the small arrow at the right part of the input field of "inSubTask" and selecting "Insert Component Reference". Dymola presents a list in which the instances (like comedi.fastSampler) are listed that can be utilized in the input field, and the modeler has just to select the appropriate entry.

3.3 Mapping to Target Data Types

Modelica has the four basic data types `Real`, `Integer`, `Boolean` and `String` that are usually mapped by Modelica translators to the C-types "`double`", "`int`", "`int`" and "`char*`", respectively. In many cases a different mapping is desired if the target is an embedded micro-controller. The details how this is defined is not standardized in Modelica 3.1. Standardization will occur when more practical experience is gained. In the meantime, vendor-specific enhancements have to be used.

In the simplest case, the "target.kind" definition in the "Target" record defines the type of the target machine. A vendor may associate different data type mappings for different target kinds. For example, a Modelica "Real" type may then be automatically mapped to a C "float" type.

For cheap microprocessors that do not have a floating point unit, such an automatic type mapping is not sufficient. For this purpose, vendor-specific variable annotations are planned that can be changed via hierarchical modifiers. The purpose is to allow definitions of the following form:

```
block Controller
  Real x(min=20, max=80);
  ...
end Controller;

Controller myController(x annotation(
   mapping(__NameOfVendor(
                  targetType = uint16,
                  min=10, max=100))));
```

The interpretation is that "x" is mapped on the target machine to an unsigned 16-bit integer "xi" with range [0 .. 65535] so that x = 10.0 is mapped to xi = 0 and x = 100.0 is mapped to 65535:

```
xi = round( (x - 10)*65535 / (100-10) )
```

The "targetMapping" min/max values (10 .. 100) might be different from the variable min/max values (20 .. 80) in order to have a margin so that operations on "xi" do not immediately cause overflow.

It is of course not practical to define such a data type mapping on every variable in a controller manually. Here, special tool support is needed. Possible, tool-specific approaches might be:

- All variables from the model part that shall be downloaded are displayed in a hierarchical variable browser and the GUI supports a convenient way to define the mapping quickly for sets of variables (e.g., all selected variables, or all variables of the same type in a particular hierarchy).
- The data type mapping might be defined for a few variables only (e.g., for the input variables). Via the equation-based relationships between variables, this mapping is propagated along the equations. E.g., if "a = b + c" and a data type mapping is defined for "b", but not for "a" and "c", then "a" and "c" are mapped in the same way as "b". This approach is similar to the automatic "unit" propagation in Dymola.

This approach of data type mapping has the big advantage that every Modelica model can be utilized even on cheap microprocessors without any changes to the "logical" model (at least in principal). In contrast, the standard approach in many controller environments is much more restrictive: Every model has to be defined from the beginning in the desired data types of the expected target system. As a result, whenever a controller is designed, it must be re-implemented from scratch for a particular target system.

4 Modelica Language Extensions

In this section the Modelica language extensions and the needed algorithms are sketched to implement the approach discussed above. All the details can be found in the Modelica 3.1 Language Specification *(Modelica 2009, Chapter 16)*.

4.1 Defining Subtask Boundaries

Boundaries of subtasks are identified with the following built-in operator (which is part of the built-in package "Subtask"):

```
Subtask.decouple(v);  // same as v
```

A boundary between a subtask A and a subtask B is defined by using this operator in an equation of subtask A with a variable v which is computed in subtask B. The operator returns its argument. Typically, this operator is used as:

```
u = Subtask.decouple(y);
```

where y is an output of subtask B and u is an input of subtask A. The effect is that "u = y" and "u" and "y" are in different subtasks.

4.2 Defining Subtask, Task and Target

The "**mapping**" annotation defines properties of variables. This annotation can only be applied on a declaration of a variable that does not have a **constant** or **parameter** prefix. It is usually applied on input and output variables of a subtask or a task. Example:

```
parameter Modelica.SIunits.Time Ts;
RealInput u annotation(mapping(
 target (identifier = "cluster"),
 task   (identifier = "slowTask",
         sampleBasePeriod = Ts),
 subtask(identifier = "reference",
         samplingType =
         Subtask.SamplingType.Periodic,
         samplePeriodFactor = 4)));
```

The meaning is that variable "u" is in the subtask "reference" which is periodically sampled with a sample period of "4*Ts". Subtask "reference" is within task "slowTask" that has a base sampling period of Ts. Task "slowTask" shall be downloaded to the target machine with the name "cluster".

The mapping annotation is formally defined by the following hierarchical record definition (as with all annotations, also here vendor-specific extensions can be added):

```
record mapping
  Boolean apply = true;
  Target     target ;
  Task       task   ;
  Subtask    subtask;
end mapping;
```

```
record Target
  String identifier="DefaultTarget";
  String kind = "DefaultTargetType";
end Target;

record Task
  String   identifier = "DefaultTask";
  Integer  onProcessor      = -1;
  Integer  priority         =  1;
  Modelica.SIunits.Period
           sampleBasePeriod =  0;
end Task;

record Subtask
  String identifier= "DefaultSubtask";
  Subtask.SamplingType samplingType =
        Subtask.SamplingType.Continuous
  Integer samplePeriodFactor(min=1)= 1;
  Integer sampleOffsetFactor(min=0)= 0;
  IntegrationMethod integrationMethod
                   = "SameAsSimulator";
 Modelica.SIunits.Period fixedStepSize;
end Subtask;
```

All values supplied to these records can be parameter expressions. If

```
mapping(apply = false,
        target (..),
        task   (..),
        subtask(..));
```

the "target(..), task(..), subtask(..)" definitions are ignored. This is, e.g., used to conveniently define in a parameter menu whether the input and/or the output signal of a communication block defines target/task/subtask properties without complicated Modelica code.

The mapping annotation defines that the respective variable is computed in the task with the identification "task.identifier" and with the task priority "task.priority", on the target platform (e.g., computer, processor) with the identification "target.identifier" on processor "onProcessor".

The interpretation of task.identifier, task.onProcessor, task.priority, target.identifier and target.kind is tool-dependent. For example, target.kind may identify a multi-processor or multi-core target machine and task.onProcessor may identify the processor or core on this target. Alternatively, a tool may identify a particular processor or core with target.kind and may ignore task.onProcessor.

The respective task may have one or more subtasks. A task is active when any of its subtasks is active. A subtask is defined with the following properties:

- If samplingType = Subtask.SamplingType.-Continuous, the subtask is a continuous system that is always active.
- If samplingType = Subtask.SamplingType.-Periodic, the subtask is periodically sampled with a sample period of "samplePeriodFactor * task.sampleBasePeriod" and an offset of "sampleOffsetFactor*task.sampleBasePeriod". So sample period and sample offset are integer multiples of the task.sampleBasePeriod.
- The differential equations in a subtask are integrated according to the "integrationMethod" property. For fixed-step integration methods, a fixed integrator step size of fixedStepSize is used. A tool may adapt the selected fixed step size, e.g., by automatically restricting it to the time from the previous to the actual activation. Usually, fixedStepSize = samplePeriodFactor * task.sampleBasePeriod. In some applications, fixedStepSize might be smaller than one sample period, in order to have several integrator steps in one sample period since otherwise the fixed step size integration method might not be stable.

The mapping annotation influences the simulation result and therefore different simulation results might be obtained if this annotation is removed.

4.3 Inquiring Subtask Properties

In order for purely discrete models to be implemented, there are two operators to inquire properties of the subtask in which the model is running:

- **`Subtask.activated`**`()`:
  Returns true at the activation time instant of the subtask, where this operator is called. At all other time instants when the associated task is exeuted, including initialization, the operator returns false.
- **`Subtask.lastSampleInterval`**`()`:
  Returns the time instant from the activation time instant of a subtask to the previous activation time instant of the same subtask, where this operator is called.

If one of these operators is used, the corresponding subtask is not allowed to have subtask.sampleType = Subtask.SamplingType.Continuous.

In many standard cases these operators are not needed. Typically, a controller block, like a PI block, is implemented in its continuous form. When the subtask is periodically sampled, the Modelica translator automatically derives the discrete form, if an appropriate integration algorithm with fixed step-size is used. For linear control systems it is recommended to use the trapezoidal integration algorithm, since this gives the closest correspondence between the continuous and the sampled form and only small lin-

ear equation systems must be solved in the discrete form (since the trapezoidal method is an implicit integration algorithm).

In some cases, a controller has only a discrete representation. A typical example is a finite impulse response (FIR) filter. A mean value FIR filter, would be typically implemented in the following form:

```
block meanValueFilter
  import Modelica.Blocks.Interfaces;
  Interfaces RealInput  u;
  Interfaces.RealOutput y;
equation
  when {initial(), Subtask.activated()}
    then
    y = (u + pre(u)) / 2;
  end when;
initial equation
  pre(u) = u;  // steady state init
end meanValueFilter;
```

4.4 Partitioning a Model in to Parts

Via the **decouple**(..) operator and the **mapping** annotation, certain variables of a model are marked. In this section the algorithm is sketched how to derive all equations that belong to a particular subtask and task, respectively:

This requires the "Block Lower Triangular" (BLT) transformation to be applied several times. The BLT algorithm is, e.g., described in *(Pantelides 1988)*. This is the standard algorithm used in Modelica translators to sort the equations of a model and identify the algebraic loops. In *(Pantelides 1988)* it is shown that a differential-algebraic equation system does not have a unique solution[2], if all "**der**(v)" are replaced by "v" and a unique assignment for all unknown variables is not possible ("v" are treated as unknown variables in this case).

We will use a similar technique below. The motivation is that the "**der**(..)" and "**pre**(..)" operators act as "loop breakers" between equation systems. If "**der**(v1)" is replaced by "v1" and "**pre**(v2)" is replaced by "v2" and all "v1" and "v2" are treated as unknowns, then algebraic loops are present between all equations that need to be "treated" together.

Based on this observation we can now sketch the partitioning algorithm:

1. BLT to determine the (asynchronous) tasks:

If tasks are present, there are function calls to receive signals from another task or from external inputs and to send signals to another task or external outputs. From a Modelica point of view, there is no coupling between variables of different tasks (due to the function calls) and therefore the equations are naturally "cut" in to partitions. These partitions are determined by replacing all **pre**(v1) with v1 and all **der**(v2) by v2 and by performing a BLT transformation.

All BLT blocks that have variables with the same task.identifier annotation belong to the same task. If a BLT block B references one or more variables that are assigned in a BLT block A, that belongs to a task task.identifier, then all equations of B belong to task task.identifier. If a BLT block C references variables that are assigned in B, then all equations of C belong to task task.identifier, and so on. If a BLT block, directly or indirectly, references variables that are assigned in two different tasks, this is an error (wrong mapping annotations). All remaining BLT blocks that do not belong to any task are collected together to a "continuous" default task. This default task is usually running on the host machine or might also be deactivated (not running).

2. BLT to determine the (synchronous) subtasks:

This is achieved by inspecting all equations of every task. For every task, the **decouple**(v) operators are conceptually replaced by zero, so that "v" is no longer part of the equation where **decouple**(v) appeared. As a result, subtasks are decoupled. BLT is performed on all equations of a task by replacing all **pre**(v1) with v1 and all **der**(v2) by v2.

If one or more variables of a BLT-block A have a subtask annotation, the equations belong to this subtask S. If a BLT block B references one or more variables that are assigned in A, all equations of B belong also to S. If a BLT block C references variables that are assigned in B, all equations of C belong also to S, and so on. All remaining BLT blocks that do not belong to any subtask, are collected together to a "continuous" default subtask. It is an error, if a BLT block, directly or indirectly, references variables that are assigned in two different subtasks. If different subtasks have identical samplingType, samplePeriodFactor and sampleOffsetFactor, the subtasks can be merged (the subtasks are sampled at the same time instants but different integration methods are used for the subtasks).

3. BLT to determine the sorted equations in a task:

Standard BLT is performed on the equations of a task (identified in step 1) to determine the execution order of all equations. In this phase, every "**decouple**(v)" operator is replaced by "v". If sampled subtasks are present, the corresponding equations

[2] or more precisely, the system has an infinite index. If on the other hand all variables "v" have a unique assignment, then and only then, the "Pantelides" algorithm to determine the equations to be differentiated will converge.

(identified in step 2) must be guarded by if-clauses and must be only evaluated if the corresponding sampling event occurs. As a result the sorted (synchronous) equations of a task are obtained.

Note, due to the equation sorting, it is guaranteed that a variable reading from an input communication channel is only used after it is read and that a variable is first computed before writing it to an output communication channel.

The description above was made for clarity. It is, however, not the most efficient implementation. For example, it is possible to combine step 1 and 2, by just performing the BLT transformation according to step 2, i.e., in total only two and not three BLTs are needed . The task/subtask annotations are then used in a corresponding way to determine the tasks and subtasks.

5 Conclusions

In this article, a powerful extension to Modelica has been described that is used to define a "logical" model in Modelica and from this *same* model derive various "real" representations as needed for, e.g., Model-in-the-Loop, Software-in-the-Loop, or Hardware-in-the-Loop Simulation, as well as rapid prototyping, or generation of production code. With respect to standard approaches in state-of-the-art control design environments, no copying of model parts takes places and the data type on the target machine is defined as a mapping rule of the "logical" model.

The language elements are not complete and some features are missing. After getting more experience with this new way of describing control systems, more features will be standardized and missing features will be added. Especially, it is planned to include triggered subtasks in the next version.

6 Acknowledgements

Partial financial support of DLR by BMBF (BMBF Förderkennzeichen: 01IS07022F) for this work within the ITEA2 project EUROSYSLIB is highly appreciated (www.itea2.org/public/project_leaflets/EUROSYSLIB_profile_oct-07.pdf).

Dynasim thanks the Swedish funding agency VINNOVA (2008-02291) for partial funding for this work within the ITEA2 project MODELISAR.

Furthermore, we would like to thank our Modelica Association colleagues Ramine Nikoukhah (INRIA), Torsten Blochwitz and Gerd Kurzbach (ITI GmbH) for fruitful discussions.

References

Akesson J., Nordström U., Elmqvist H. (2009): **Dymola and Modelica_EmbeddedSystems in Teaching – Experiences from a Project Course**. In: F. Casella (editor): Proc. of the. 7th Int. Modelica Conference, Como. www.modelica.org/events/modelica2009

Bauschat, M., Mönnich, W., Willemsen, D., and Looye, G. (2001): **Flight testing Robust Autoland Control Laws**. In Proceedings of the AIAA Guidance, Navigation and Control Conference, Montreal CA.

Comedi (2009). Linux Control and Measurement Device Interface. www.comedi.org.

Dymola (2009). **Dymola Version 7.3**. Dassault Systèmes, Lund, Sweden (Dynasim). Homepage: www.dymola.com.

Franke R., Babji B.S., Antoine M., Isaksson A. (2008): Model-based online applications in the ABB Dynamic Optimization framework. In: B. Bachmann (editor): Proc. of the 6th Int. Modelica Conference, Bielefeld. www.modelica.org/events/-modelica2008/Proceedings/sessions/session3b1.pdf

Looye G., Thümmel M., Kurze M., Otter M., Bals J. (2005): **Nonlinear Inverse Models for Control**. In: G. Schmitz (editor): Proc. of the 4th Int. Modelica Conference, Hamburg. www.modelica.org/events/Conference2005/online_proceedings/Session3/Session3c3.pdf

Modelica (2009). **Modelica Language Specification 3.1**. www.modelica.org/documents/ModelicaSpec31.pdf

Pantelides C. (1988): **The consistent initialization of differential-algebraic systems**. SIAM Journal of Scientific and Statistical Computing, pp. 213-231.

Schäuffele J. and T. Zurawka (2005): **Automotive Software Engineering – Principles, Processes, Methods and Tools**. SAE International. ISBN-10 0-7680-1490-5.

Thümmel M., Otter M., Bals J. (2005): **Vibration Control of Elastic Joint Robots by Inverse Dynamics Models**. H. Ulbrich, W. Günthner (editors): IUTAM Symposium on Vibration Control of Nonlinear Mechanisms and Structures, München, ISBN 978-1-4020-4160-0, pp. 343-353.