Towards a Standard-Conform, Platform-Generic and Feature-Rich Modelica Device Drivers Library

Bernhard Thiele[1] Thomas Beutlich[2] Volker Waurich[3] Martin Sjölund[1] Tobias Bellmann[4]

[1]PELAB, Linköping University, Sweden, {bernhard.thiele,martin.sjolund}@liu.se
[2]ESI ITI GmbH, Germany, thomas.beutlich@esi-group.com
[3]Chair of Construction Machinery, TU Dresden, Germany, volker.waurich@tu-dresden.de
[4]Institute of System Dynamics and Control, DLR, Germany, tobias.bellmann@dlr.de

Abstract

There are many cases where simulation applications need to interact with their environment. Typical examples are Human-in-the-Loop (HITL) simulators (including flight, driving, and marine training simulators), Hardware-in-the-Loop (HIL) simulators, but also offline process simulators which cannot operate in a completely self-contained manner and therefore need to be coupled to external applications. Embedded control applications are another related area requiring interaction between applications and their environment. The *Modelica_DeviceDrivers* library, which had its first release as open-source library in 2012, tries to cater to such use cases. This paper describes the library for the first time and reports about the numerous challenges that the project experienced to meet its goal of supporting several platforms and tools within a standard-conform, platform-generic, feature-rich, and easy-to-use Modelica library. Furthermore, the paper gives an insight into the inner mechanics of the library's communication and serialization functionalities, the various supported hardware interfaces and the possibilities to generate code for embedded systems.

Keywords: human-in-the-loop, hardware-in-the-loop, real-time simulation, embedded control application, Modelica external C

1 Introduction

The most common usage of Modelica models is for offline simulation experiments. However, in many cases simulations need to interact with their environment or other software components. Typical examples are Human-in-the-Loop (HITL) simulators (including flight, driving, and marine training simulators), Hardware-in-the-Loop (HIL) simulators, but also offline process simulators which cannot operate in a completely self-contained manner and therefore need to be coupled to external applications. Furthermore, Modelica can be used for developing (model-based) control applications that also require interaction with their environment.

There are different approaches for enabling the above-mentioned applications in the context of Modelica. Several development environments offer tool chains for real-time simulation and/or model-based development of embedded control applications. Some of these environments can be coupled with Modelica tools, by wrapping code that is generated from Modelica tools into respective third-party tool-internal representations which can be connected to hardware devices in the respective development environment. For example, such customized solutions are available in Dymola[1] via its DymolaBlock interface to the MATLAB/Simulink[2] tool chain, OpenModelica[3] via customized tool chains (Worschech and Mikelsons, 2012), or SimulationX[4] via Code Export for Simulink/Simulink Coder[2] or HIL environments like dSPACE DS1006[5], NI VeriStand[6] or ETAS LABCAR[7] (Blochwitz and Beutlich, 2009). Furthermore, it may also be possible for a Modelica tool to generate Functional Mock-up Units[8] (FMUs) which can be imported into compatible simulator environments (*e.g.,* the dSPACE SCALEXIO[5] HIL simulator).

Instead of embedding the (FMI-) compiled Modelica model into a simulator environment, the *Modelica_DeviceDrivers* (MDD) library uses a different approach. The MDD library provides access to external devices by utilizing Modelica's external function interface for interfacing to the C API of various device drivers directly from Modelica models (see Section 2).

Historically, the origins of the MDD library can be traced back to the *ExternalDevices* library (Bellmann, 2009), an internal DLR[9] Modelica library developed for the interactive simulation and visualization of Modelica models. The *ExternalDevices* library already supported UDP and shared memory communication as well as several

[1]Dassault Systèmes, https://www.3ds.com
[2]The MathWorks, https://mathworks.com
[3]Open Source Modelica Consortium (OSMC), https://www.openmodelica.org
[4]SimulationX by ESI, https://www.simulationx.com
[5]dSPACE, https://www.dspace.com
[6]National Instruments, http://www.ni.com
[7]ETAS, http://www.etas.com
[8]FMI development group, https://www.fmi-standard.org
[9]Deutsches Zentrum für Luft- und Raumfahrt (DLR), German Aerospace Center, http://dlr.de

input devices (keyboard, 3Dconnexion SpaceMouse[10], and game controller). Additionally, it featured a model-integrated real-time visualization system, the foundation of the later *DLR Visualization* library (Hellerer et al., 2014).

However, the *ExternalDevices* library only supported Microsoft Windows and was developed and tested using only the Dymola tool, which caused unintentional incompatibilities with other Modelica tools. In the further course of development, it was decided to split the *ExternalDevices* library into the commercial *DLR Visualization* library and an open-source cross-platform hardware interface library, the *Modelica_DeviceDrivers* library. The library is available from its GitHub project site at `https://github.com/modelica/Modelica_DeviceDrivers/`. This paper is based on MDD v1.5.0.

2 Modelica_DeviceDrivers

The MDD library allows Modelica models to access hardware devices by using the Modelica external C interface calling the appropriate C driver functions provided by the underlying operating system (see Section 2.1).

The library is organized in several layers as indicated in Figure 1. It provides two high-level drag & drop block interfaces.

1. The `Blocks` components are compatible to Modelica v3.2, using **when** `sample()` for periodically calling Modelica functions from the Function Layer.

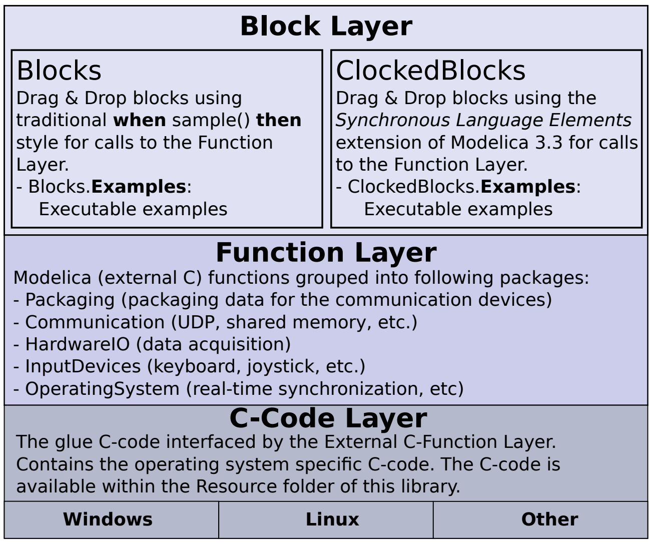
2. The `ClockedBlocks` components use the synchronous language elements extension introduced in Modelica v3.3 and are compatible with the *Modelica_Synchronous* library (Otter et al., 2012). Due to this support, the MDD library formally depends on the *Modelica_Synchronous* library, but in practice the *Modelica_Synchronous* library (and tool support for the synchronous language elements extension) is only required for this `ClockedBlocks` interface.

2.1 Cross-Platform Support

As of MDD v1.5.0, Windows and Linux are supported as main platforms, but prototypical work also targets popular embedded systems boards directly (see Section 4.2).

When accessing hardware devices, a Modelica model or application calls Modelica functions from the Function Layer (see Figure 1). These Modelica functions provide a generic interface to the underlying C Code Layer, which is accessed by Modelica's external function interface. The platform differentiation is handled in the C Code Layer which uses preprocessor directives for conditional inclusion/exclusion of platform-specific code (**`#if`**, **`#else`**, **`#endif`**, *etc.*) similar to the code fragment below.

[10]3Dconnexion, `https://3dconnexion.com`


Block Layer

Blocks
Drag & Drop blocks using traditional **when** sample() **then** style for calls to the Function Layer.
- Blocks.**Examples**: Executable examples

ClockedBlocks
Drag & Drop blocks using the *Synchronous Language Elements* extension of Modelica 3.3 for calls to the Function Layer.
- ClockedBlocks.**Examples**: Executable examples

Function Layer
Modelica (external C) functions grouped into following packages:
- Packaging (packaging data for the communication devices)
- Communication (UDP, shared memory, etc.)
- HardwareIO (data acquisition)
- InputDevices (keyboard, joystick, etc.)
- OperatingSystem (real-time synchronization, etc)

C-Code Layer
The glue C-code interfaced by the External C-Function Layer. Contains the operating system specific C-code. The C-code is available within the Resource folder of this library.

Figure 1. MDD layered architecture.

```
#if defined(_MSC_VER) || defined(__CYGWIN__
    ) || defined(__MINGW32__)
#include <windows.h>
/* Windows specific code goes here */
#elif defined(__linux__)
#include <unistd.h>
/* Linux specific code goes here */
#else
#error "Modelica_DeviceDrivers: Unsupported
     compiler or platform"
#endif
```

2.2 Extended Tool Support

Back in 2009, the library was developed using the Dymola tool. With MDD v1.4.0, considerable development efforts have been spent on the Modelica compliance of the library in order to better support SimulationX and OpenModelica.

Since OpenModelica v1.11.0 Beta 1 the MDD `SerialPackager` blocks as well as the `Communication` blocks are finally supported by OpenModelica. For achieving this, it was necessary to change parts of the MDD library (under the constraint of maintaining backwards compatibility), and at the same time, to extend the abilities of respective tools (partly by providing support for non-standard Modelica constructs). This is discussed in more detail in Section 3.2.3.

2.3 Library Structure

Figure 2 shows a screenshot of the package browser view with loaded MDD library. The first two sub-packages `Blocks` and `ClockedBlocks` provide the drag & drop blocks which correspond to the Block Layer of Figure 1. The remaining sub-packages (except `Utilities` and `EmbeddedTargets`) provide the Function Layer. Both layers use sub-packages for subdividing the provided functionality into different groups. Package `EmbeddedTargets` contains highly target specific function and blocks for supporting restricted embedded systems like the Arduino microcontroller (see Section 4).

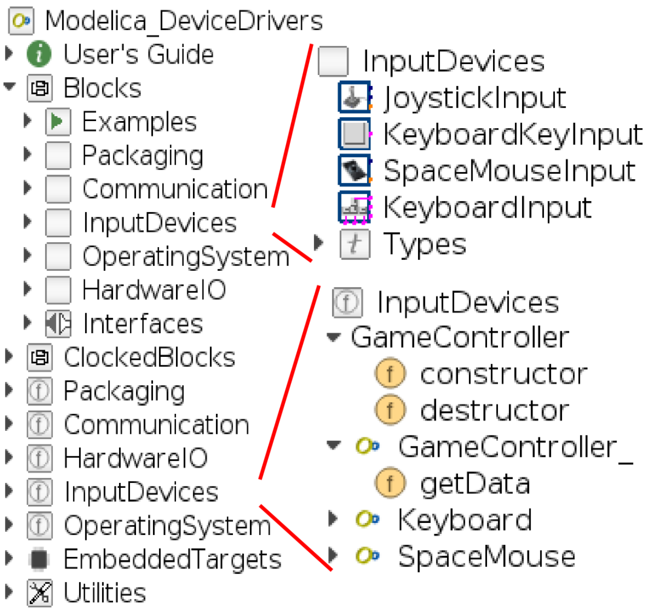


Figure 2. MDD library structure.

Furthermore, Figure 2 gives an indication about the relation between the Block Layer and the Function Layer. Typically, a device driver block will instantiate the corresponding external object from the Function Layer. Modelica's external objects allow external functions to access the internal memory (created by the `constructor`) between calls to external functions, *i.e.*, the device handlers are maintained in memory in order to access them in subsequent simulation phases. Modelica also guarantees that both the `constructor` and `destructor` functions of an external object are called exactly once, enabling a reliable *one-time* initialization and termination of hardware devices, usually during the initialization and termination phase of the Modelica simulation model, respectively. For example, the `JoystickInput` block creates an instance of the external object `GameController`. The package `GameController_` collects functions that can operate on external objects of type `GameController`. This package provides the function `getData`, which takes a `GameController` object as argument and returns the values of the axes and buttons of its associated hardware device.

A good way of learning how to use the Block Layer interface of the library is by exploring the `Examples` package. Care has been taken to provide self-explanatory usage examples for the provided device driver blocks.

2.4 Interfaces

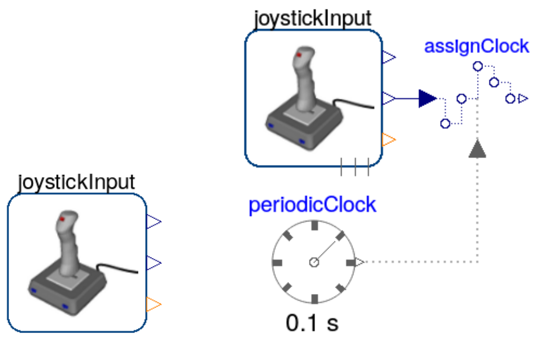
MDD library functionality can be accessed by drag & drop of blocks from the `Blocks` and `ClockedBlocks` subpackages, or by direct calls to the underlying functions.

An example, which directly uses the Function Layer for accessing a game controller, is given below:

```
model GameControllerExample
  import
    Modelica_DeviceDrivers.InputDevices.*;
  parameter Integer id = 0 "0 = first
      attached game controller";
  GameController gc = GameController(id);
  discrete Real axesRaw[6];
  Integer buttons[32], pOV;
equation
  when sample(0, 0.1) then
    (axesRaw, buttons, pOV) =
        GameController_.getData(gc);
  end when;
end GameControllerExample;
```

The code above creates an external object named `gc`. The constructor for this object takes the argument `id`. This argument allows specifying which controller to use if several game controllers are attached to the system. The function `getData` is called periodically within a **when**-clause. It takes the external object `gc` as argument and returns vectors which contain the values read from the associated game controller. The vector is pre-dimensioned, so that it can attune to controllers featuring as much as six axes, 32 buttons and a POV (point of view) switch. The actually available data depends on the connected game controller hardware. Tests with the actual hardware are needed for determining which vector entry corresponds to which physical axis or button.

Figure 3 shows how game controllers can be accessed by simply dragging & dropping a `JoystickInput` block into the diagram view of a Modelica tool. While Figure 3a uses the block found in the `Blocks` package, Figure 3b uses the corresponding clocked variant from `ClockedBlocks`. The additional blocks `periodicClock` and `assignClock` are from the *Modelica_Synchronous* library. They associate a periodic clock to the variables and equations within the `JoystickInput` block. As a result, the underlying `getData` function will be called whenever the associated clock ticks (*i.e.*, every 0.1 s in the presented example).



(a) Using `Blocks` **(b)** Using `ClockedBlocks`

Figure 3. Accessing game controller devices by using the `JoystickInput` block from the `Blocks`, or the `ClockedBlocks` package.

The example models can be simulated, but real-time synchronization is required to slow the simulation speed down, in order to synchronize the real-time inputs with the simulation progress. The MDD library provides convenient support for (soft) real-time synchronization[11]. However, a user should consider that Modelica tools might provide better (tool-specific) options for real-time synchronization.

2.5 Features

The MDD library has grown to support a respectable amount of hardware devices and associated features that will be briefly presented in this section.

2.5.1 Input Devices

Standard input devices such as keyboard and game controllers are ubiquitously available on the market, enabling the user to build up interactive simulations quickly. MDD provides blocks for using the generic keyboard and game controller interface of Windows or Linux (see Figure 4).

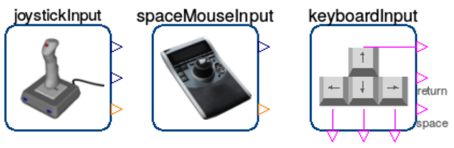


Figure 4. Supported input devices from the `Blocks` package.

In addtion, more specialized hardware like the 3Dconnexion SpaceMouse is supported for both platforms. Often, these blocks will be used for interactive desktop simulations, but they can also become part of more involved (cost-efficient) HITL simulation scenarios.

2.5.2 Communication

The most comprehensive and complex part of the library is related to implementing support for communication devices in Modelica and external C code.

Supported Devices Figure 5 gives an overview over the supported devices.

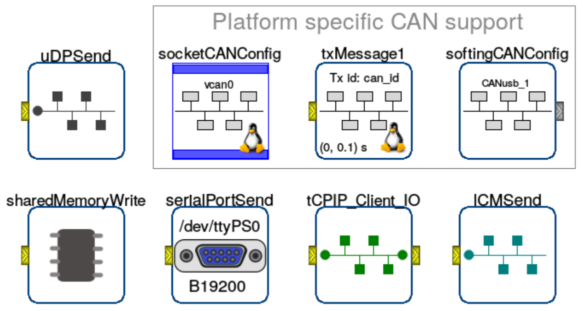


Figure 5. Supported communication devices from the `Blocks` package.

Cross-platform support for UDP and shared memory was already available in the first released version of MDD. Support for serial port communication is available since MDD v1.3 (Linux) and v1.4.0 (Windows). A client block for TCP/IP socket communication was added in v1.4.0 (Windows) and v1.5.0 (Linux). Furthermore, support for sending and receiving of Lightweight Communications and Marshalling (LCM) datagrams[12] was added in v1.5.0. LCM is a set of libraries and tools for message passing and data marshalling[13], which is particularly targeted at low-latency real-time applications for robotic systems (Huang et al., 2010).

Basic support for the Controller Area Network bus (CAN bus) is available by two different block sets. The first is based on the CAN Layer2 API from Softing[14] and restricted to the Windows platform. The second uses the SocketCAN interface provided by the Linux kernel.

Packaging Concept Communication devices like UDP or shared memory use a common packaging concept in order to send or receive data. Therefore, the same packager can be used with different communication devices. Figure 6 shows an example in which a package consisting of three variables of type `Real` followed by a variable of type `Integer` is either transmitted using shared memory or UDP blocks. Switching between the two communication devices is achieved by simply replacing the corresponding device block.

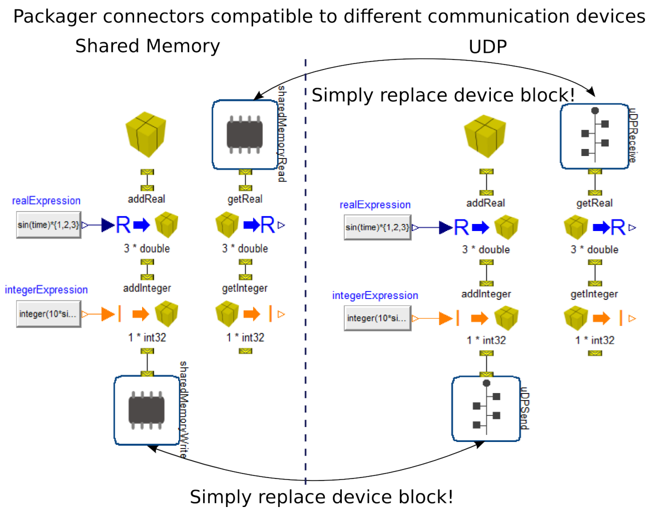


Figure 6. Simple switching of communication devices due to common packaging concept in order to send or receive data.

The packages are constructed by using blocks from the `Packaging` sub-package (see Figure 2). In the initial design of MDD, it was expected that different packaging concepts would be supported which share a common connector interface. However, as of MDD v1.5.0 the `SerialPackager` is the only available packager. It allows periodically adding or retrieving fixed size vectors to or from a package, respectively. Figure 7 shows the available blocks for serializing Modelica variables of the

[11] See documentation to block `SynchronizeRealtime`.

[12] LCM project, `https://lcm-proj.github.io`

[13] As of MDD v1.5.0, only the communications aspect of LCM is considered.

[14] Softing, `http://industrial.softing.com`

predefined types `Boolean`, `Integer`, `Real` and `String` into a "package". The type `Real` can be packed either as double-precision or (using a C static cast) as single-precision floating-point number.

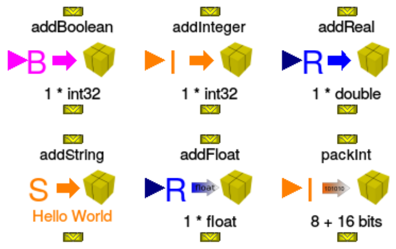


Figure 7. `SerialPackager` blocks for adding variables to a package.

At the C side, a package is a C byte array in which C variables with respectively indicated types are simply successively appended in a data-flow prescribed order. For example, in Figure 6 the resulting byte array starts with three **`double`** values (3×8 bytes) followed by one `int32` value (4 bytes), resulting in a byte array of size 28. For the sake of providing an illustrative example at the C language level the following C code snippet constructs a structurally equal package named `data` (the example shall shed light on the concept, it does not advocate a coding style using magic numbers for array offsets):

```
double v1[3] = {1.1, 2.2, 3.3};
int v2 = 4;
unsigned char* data = (unsigned char*)
    calloc(28, sizeof(unsigned char));
memcpy(&data[0], &v1[0], sizeof(v1));
memcpy(&data[24], &v2, sizeof(v2));
```

Figure 7 shows the blocks for adding variables to a package, symmetrically, blocks are available for retrieving variables from a package. Using these blocks is deemed to be rather intuitive with the notable exception of the `packInt` block. This block allows packing unsigned integer values at the bit level. The number of bits used for encoding is set by a parameter `width`, therefore the maximum value of the integer signal that can be encoded is $2^{\text{width}} - 1$. A parameter `bitOffset` allows to specify the bit at which the encoding starts relative to the preceding block. Since MDD v1.3 most blocks support specifying the byte ordering (big-endian or little-endian format).

It is simple to use the `SerialPackager` blocks for deserializing data which has been serialized by it (see Figure 6). In practice, however, communication typically needs to be established with a remote station that is unrelated to the Modelica model. As long as this remote station periodically sends or receives structurally static, fixed sized packages, it is usually quite convenient to establish a communication using the MDD blocks. If the remote station uses a more dynamic protocol, it becomes more difficult. In some cases using the Function Layer directly (instead of the Block Layer) can provide additional flexibility for coping with more dynamic protocols. However, the main use-case for the `SerialPackager` concept is periodically sent, structurally static data. These restrictions may be relieved in future versions of the MDD library by providing alternative, well-established "Packagers" that offer support for more flexible means of packaging data, *e.g.*, the data marshalling of the LCM library or the efficient binary serialization format of the MessagePack library[15].

Finally, it turned out that the `SerialPackager` blocks were a major hurdle for extending the number of Modelica tools which support MDD (see Section 3.2).

2.5.3 Hardware I/O

Package `HardwareIO` (see Figure 2) is intended for data acquisition hardware like digital-analog converter (DAC), analog-digital converter (ADC) and other interface hardware. As of MDD v1.5.0, it contains only one subpackage, which provides support for the Linux control and measurement device interface "Comedi". The Comedi project develops open-source drivers, tools, and libraries for data acquisition[16]. The project provides a common interface for accessing supported data acquisition hardware (see the website for supported hardware). The MDD library implements an interface to the Comedi user-space library.

Figure 8 shows an example model, which uses the available blocks. Configuration of the device is performed in the Modelica record named `comedi`. The record contains an external object `dh` of type `ComediConfig` which contains the Comedi device handle and is passed through a parameter to the other blocks (`comedi.dh`). Using external objects in records is not standard-compliant to Modelica v3.3 revision 1 (Modelica Association, 2014), which is further discussed in Section 3.3.

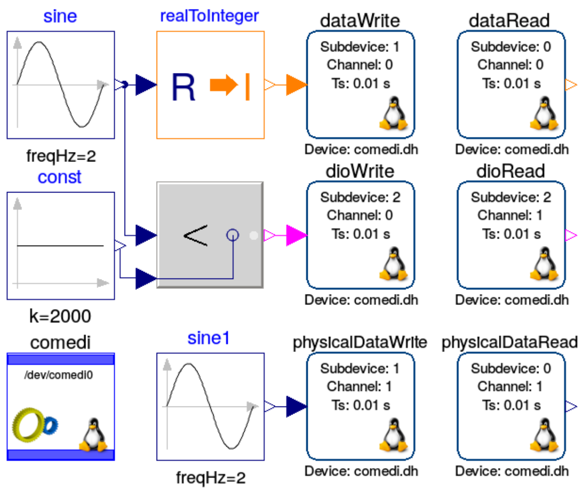


Figure 8. Accessing data acquisition hardware via the Linux control and measurement device interface "Comedi".

Writing or reading raw integer values to DAC or from ADC channels is provided by the blocks `DataWrite`

[15]MessagePack project, `https://msgpack.org`
[16]Comedi project, `http://comedi.org`

or `DataRead`, respectively. These blocks have each a variant which works with physical values, instead of the raw integer values (`PhysicalDataWrite` and `PhysicalDataRead`). Blocks `DIOWrite` and `DIORead` support digital input and output (DIO) channels.

2.5.4 Embedded Targets

MDD v1.5.0 introduced the new top-level package `EmbeddedTargets`. The package is intended for platform-specific targets, such as microcontrollers, that cannot so easily share code with other devices due to memory or hardware limitations. There exists first prototypical support for the Atmel[17] AVR family of microcontrollers. A prototype application is described in Section 4.2.

3 Modelica Standard-Compliance

Using a Modelica library-based approach for accessing hardware devices from a simulation started as an experiment, which relied on the Dymola tool and its support for interfacing external C code. However, when trying to extend the number of Modelica tools supporting the MDD library, it became apparent that quite a few constructs that were useful and appreciated by the initial authors of the library were not supported by other tools and were partly problematic in respect of compliance to the Modelica standard.

On one hand, this section reports on important development efforts (starting with MDD v1.4.0) that have been spent on the Modelica compliance of the library for better supporting SimulationX and OpenModelica, and on the other hand it addresses open issues which may be of interest for future improvements to the Modelica standard, or which may require possibly non-backwards compatible revisions of the MDD library for achieving full Modelica compliance.

3.1 Modelica's External Function Interface

As the Modelica standard specification on the external function interface improved over the years, standard-conform libraries with external C code dependencies could be created in a more satisfying way. For example, Modelica v3.2 standardized the search directory structure for the external C header files and libraries (Modelica Association, 2010, p. 153). Having a standardized directory structure facilitated creating cross-platform libraries with external C library dependencies. For example, the Modelica code snippet below declares an include dependency to the header file `MDDKeyboard.h` and linker dependencies to the libraries `X11` and `User32`:

```
function getKey
  input Integer keyCode "Key code";
  output Integer keyState "Key state";
  external "C" MDD_keyboardGetKey(keyCode,
     keyState) annotation(
    Include = "#include \"MDDKeyboard.h\"",
    Library = {"X11", "User32"});
  annotation(__ModelicaAssociation_Impure=
     true);
end getKey;
```

A Modelica tool will map this information to compiler- and linker-dependent directives and thereby select the libraries that fit best for the respective platform.

3.1.1 Linking Platform-Dependent System Libraries

Having platform-specific system libraries like `X11` (for Linux only) and `User32` (for Windows only) in one generic `Library` annotation, proofed to be a significant development difficulty. As a remedy, dummy libraries of the Linux system libraries are provided in the Windows-specific library directories `win32` and `win64`, and vice versa. Furthermore, the linking to system libraries on Windows was simplified by the introduction of compiler-specific pragmas, *e.g.*, in `MDDKeyboard.h`

```
#pragma comment( lib, "User32.lib" )
```

understood by the Visual Studio compilers only. However, for GCC (including the MinGW and CygWin build environments) the issue remains unresolved[18].

3.1.2 Impure Functions

The above example function `getKey` features an additional (vendor-neutral) annotation which declares the function as “impure”. The intended meaning is that a tool may not expect that the function returns the same output for the same input, which is the typical case for MDD functions that read values from external devices. Indeed, Modelica v3.3 introduced the dedicated keyword “impure” to cater for such cases. However, since not all Modelica tools support this keyword, yet, the MDD library uses the `Impure` annotation which is understood by Dymola, OpenModelica and SimulationX.

3.1.3 Modelica Standard Improvements

Future releases of MDD may benefit from improvements on the external function interface, which are expected in the (future) Modelica v3.4 standard:

- Compiler-specific sub-directories for the platform-specific library directories, *e.g.*, if Visual Studio 2015 is used as a Windows 64-bit compiler a Modelica tool may first search directory `win64/vs2015` for dependent libraries[19].
- The `IncludeDirectory` annotation accepts *multiple* directories enabling a more convenient way to specify several external C header file dependencies distributed over different include directories[20].

[17]Atmel, http://atmel.com

[18]Modelica Issue Tracker, https://trac.modelica.org/Modelica/ticket/1668

[19]Modelica Issue Tracker, https://trac.modelica.org/Modelica/ticket/1316

[20]Modelica Issue Tracker, https://trac.modelica.org/Modelica/ticket/2103

However, a generalized build process[18] of the external code still misses the definition and (future) standardization of build features such as compilation of several C source modules, compiler flags (`CFLAGS`) or preprocessor defines (`CPPFLAGS`)[21].

3.2 The Serial Packager

The `SerialPackager` blocks are the core elements of the block-based communication support provided by the MDD library (see Section 2.5.2). They use a rather intricate approach for propagating a "package" between connected blocks.

3.2.1 Connector Definition

The definition of the `SerialPackager` input connector is given below.

```
connector PackageIn "Packager input
    connector"
  input SerialPackager pkg;
  input Boolean trigger;
  input Real dummy;
  output Boolean backwardTrigger;
  output Integer userPkgBitSize;
  output Integer autoPkgBitSize;
end PackageIn;
```

The definition of the output connector is similar, but with reversed input and output causalities. Most notably connector `PackageIn` contains an element `pkg`, which is an external object of type `SerialPackager`. This external object is passed between connected blocks (see Figure 6). Within an "add" or "get" block the passed in external object is used as an argument to external functions which first add or retrieve data from the package and then pass it on to the next block.

Due to the design of the `SerialPackager` connector sharing *both* input and output variables it is impossible to have more than one **`connect`** equation per connector. However, Modelica offers no option to tell a user already at modeling time about this maximal allowed connector cardinality.

3.2.2 Basic Concept

The following *simplified* Modelica code snippet illustrates the basic idea for adding the (`Integer`) value of an input variable `u` to a package:

```
block AddInteger
  PackageIn pkgIn "Input connector";
  PackageOut pkgOut "Output connector";
  IntegerInput u "Integer input connector";
equation
  when initial() then
    pkgIn.autoPkgBitSize =
        pkgOut.autoPkgBitSize + 32 /* bit
        size of int32 */;
  end when;
  when pkgIn.trigger then
    pkgOut.dummy = addInteger(pkgOut.pkg,
        u, pkgIn.dummy);
  end when;
  pkgOut.pkg = pkgIn.pkg;
  pkgOut.trigger = pkgIn.trigger;
  pkgOut.backwardTrigger =
      pkgIn.backwardTrigger;
  pkgOut.userPkgBitSize =
      pkgIn.userPkgBitSize;
end AddInteger;
```

The instantaneous equation invoking the `addInteger` function is activated by the event `trigger` which is propagated through the connected packager blocks. The `dummy` variables are used to establish data-flow dependencies which ensure that the "`addValue`" functions of connected blocks are invoked in the correct order. The `backwardTrigger` event allows propagating a triggering event in the inverse connector direction. Its supporting logic is omitted here for brevity. A Modelica standard-conform alternative is provided by the variable `userPkgBitSize` that allows propagating a user defined package size, *i.e.*, it is possible for a user to customize the package size of the external data buffer of the communication device block (see Section 2.5.2). However, in the default setting the necessary package size is deduced automatically with the help of the `autoPkgBitSize` variable. This approach is described in Section 3.2.4.

3.2.3 External Object Aliasing

A problem with the Block Layer of the `SerialPackager` is that the `pkg` objects within the connectors are not explicitly created by calling an external object constructor function as required in Modelica v3.3 (Modelica Association, 2014, p. 165). Instead, they rely on aliasing through (connect) equations to access an external object which has been created at another place. In Figure 6 the `pkg` object for the "add" blocks is created in the "Packager" block at the top of the figure, while the `pkg` object for the "get" blocks is created in the device block for reading from shared memory (or UDP, respectively). While the concept of external object aliases does not exist in Modelica v3.3, equating two external objects may be interpreted as an assignment to an external object, which is forbidden. The authors hope that future versions of the Modelica standard will consider use-cases that the Modelica tools Dymola, OpenModelica and SimulationX already support[22].

A Modelica standard-conform implementation that avoids the aliasing is to only rely on the Function Layer provided by package `SerialPackager_`.

3.2.4 Automatic Buffer Size

The actual creation of the `SerialPackager` object is performed in the "Packager" block, or, respectively, in the reading device block (see above). The following *simplified* code illustrates the basic concept.

[21] Modelica Issue Tracker, https://trac.modelica.org/Modelica/ticket/2145

[22] Modelica Issue Tracker, https://trac.modelica.org/Modelica/ticket/1669

```
block Packager
  PackageOut pkgOut(
    pkg = SerialPackager(bufferSize),
    dummy(start=0, fixed=true));
  Integer bufferSize;
equation
  when initial() then
    bufferSize =
      if pkgOut.userPkgBitSize > 0 then
      pkgOut.userPkgBitSize else
      pkgOut.autoPkgBitSize;
  end when;
end Packager;
```

The difficulty here is that the `bufferSize` which is needed as an argument for the external object constructor `SerialPackager(bufferSize)` needs to be computed by solving the initial system of equations. This is not supported by all Modelica tools and its Modelica compliance was discussed at the Modelica Issue Tracker with a majority opting to clarify the specification in order to forbid it[23], but on the other hand it was also discussed how the Modelica standard could be extended to allow it[24].

In the initial version of the MDD library the external object was actually created within a **when**-clause, which was clearly illegal in Modelica v3.3. As part of improving the Modelica compliance of the library, the creation of the object was moved into the component declaration.

3.3 External Objects in Records

The `SocketCAN` and the `Comedi` blocks use a Modelica record as means for specifying general settings for a hardware device. The idea is that the settings are specified once when creating an instance of the record and this instance is passed as parameter to blocks using this device. For example, the `Comedi` configuration record (stripped from some elements for brevity) is defined as

```
record ComediConfig
  parameter String deviceName =
    "/dev/comedi0" "Name of Comedi device";
  final parameter Comedi dh =
    Comedi(deviceName) "Handle to comedi
        device";
end record;
```

where `dh` is an external object. It is convenient to collect configuration information in a record, since this allows passing a complete set of related configuration settings at once. The problem here is that passing an external object as part of a record can be interpreted as the record returning the object and assigning it to another external object (which is forbidden in Modelica v3.3 but supported by Dymola). However, similarly to the external object aliasing described in Section 3.2.3 it seems highly desirable to consider use-cases as described above in some way, in future versions of the Modelica standard.

[23]Modelica Issue Tracker, `https://trac.modelica.org/Modelica/ticket/1907`

[24]Modelica Issue Tracker, `https://trac.modelica.org/Modelica/ticket/2037`

3.4 Fixed Attribute of Strings

According to Modelica v3.3 the predefined type `String` was designed without the `fixed` attribute (as opposed to other predefined types `Boolean` or `Integer`). However, such a `fixed` attribute is particularly relevant for the `GetString` block of the `SerialPackager` when retrieving sampled String data from a package. This issue was resolved by (future) Modelica v3.4 such that future Modelica tools supporting Modelica v3.4 will no longer raise a warning on the `GetString` block[25].

4 Applications

This section describes several applications that were implemented with the help of the MDD library.

4.1 Arduino

The Arduino[26] is an open-source electronics platform that features easy configurations to read the sensors, process the data and send it to other devices via a serial connection. Therefore, the Arduino can be utilized to provide sensor data in a real-time Modelica model by means of the MDD serial port implementation, as depicted in Figure 9. With the help of potentiometers or other deflection sensors, customized control devices can be built.

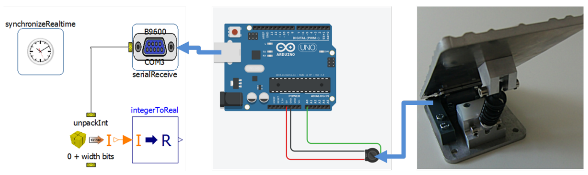


Figure 9. Setup to read potentiometer deflection during real-time simulation with MDD serial port model[27].

As an exemplary application, self-built pedals for a driving simulator can be equipped with a sensor in order to measure the displacement. The pedal itself is a steel sheet, mounted on a revolute joint and a shock spring. The measured deflection is transferred via a serial connection to a `Blocks.Communication.SerialPortReceive` in order to drive a virtual vehicle. Therefore, expensive or unavailable input devices can be substituted by custom constructions. By using a Bluetooth module with Serial Port Profile (SPP) a wireless connection between Arduino is handled in the same way as a serial port over USB connection. No further modifications are necessary to implement a wireless control device.

4.2 Embedded Control

The `EmbeddedTargets` package (see Section 2.5.4) contains blocks and functions to directly control I/O or clocks

[25]Modelica Issue Tracker, `https://trac.modelica.org/Modelica/ticket/1797`

[26]Arduino, `https://arduino.cc`

[27]Autodesk screen shots reprinted courtesy of Autodesk, Inc.

in the AVR ATmega microcontroller family[28]. The advantage of including this code into the MDD library is that it makes it simple to write a model for a microcontroller that works the same way in any Modelica tool since all the OS support, real-time code, *etc.*, is abstracted away. Provided that the Modelica tool produces minimal C-code (uses minimal features outside of standard C: for example, no linear solver included if the system has no linear systems, no OS or I/O functions, no threading models, *etc.*), and the model itself does not use C-code that the embedded target cannot support (such as file I/O), the code generator would work on pretty much any embedded target supporting C.

The Modelica code itself tries to avoid the Integer constants from the data sheets. Instead, enumerations such as `prescaler`=1/128 or `clock`=2B are passed from Modelica and the C code for the AVR target depends on function inlining in order to remove dead code. For example, the constructor for the clock takes an enumeration that specifies the clock, which should be manipulated, and after function inlining, the C code for other clocks is removed. The blocks in the MDD library try to take user-friendly constants such as `frequency`=100 Hz or `period`=0.1 s for real-time synchronization; the Modelica code then has logic to find good clock prescalers to create a matching frequency. The code does not use parameters since they cannot be guaranteed to be evaluated in Modelica, and the C-code depends on the C-compiler (AVR GCC) being able to inline and eliminate dead code from C-code such as the constructor. An example of this is the timer external object in the microcontroller, which becomes one or two bitset instructions when the function is called with a constant input:

```
function constructor "Initialize timer"
  input Types.TimerSelect timerSelect;
  input Types.TimerPrescaler clockSelect;
  input Boolean clearTimerOnMatch;
  output Timer timer;
  external "C" timer = MDD_avr_timer_init(
      timerSelect, clockSelect,
      clearTimerOnMatch)
    annotation(Include = "#include \"
       MDDAVRTimer.h\"");
end constructor;
```

```
static inline void* MDD_avr_timer_init(int
    timerSelect, int clockSelect, int
    clearTimerOnMatch)
{
  static const uint8_t
    clockSelectTable0[7] = {...},
    clockSelectTable1[7] = {...},
    clockSelectTable2[7] = {...};
  switch (timerSelect) {
#if defined(TCCR0)
  case 1: /* Timer 0 */
    TCCR0 |= ...;
    break;
#elif defined(TCCR0B)
  case 1: /* Timer 0 */
    TCCR0B |= clockSelectTable0[clockSelect
        -1];
    TCCR0A |= ...;
    break;
#endif
  case 2: /* Timer 1 */
    ...
  case 3: /* Timer 2 */
    ...
  default:
    exit(1);
  }
  return (void*)timerSelect;
}
```

[28]As of MDD v1.5.0, only ATmega16 and ATmega328P (=Arduino Uno) are supported. The code can easily be extended, but requires checking the data sheets in order to write to the correct bits.

One of the AVR examples included in MDD is the single board heating system (SBHS[29]), shown in Figure 10.

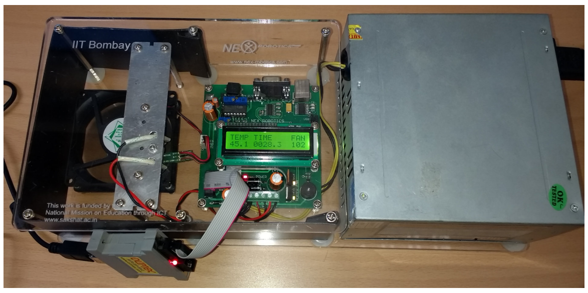


Figure 10. The single board heater system running a real-time control algorithm using firmware based on MDD code. There is a programmer attached to the board to upload new firmware, but the code runs without any computer connected to the SBHS.

The SBHS consists of a heater assembly, fan, temperature sensor, AVR ATmega16 microcontroller and associated circuitry. It was developed by IIT Bombay and is used for teaching and learning control systems (Arora et al., 2010). The MDD SBHS example uses pulse width modulation (PWM) blocks to control the heater and fan, and an analog-to-digital converter (ADC) block to read the temperature. It combines these elements with a PID controller with the goal to control the fan such that the temperature settles at a setpoint of 45 °C while a constant voltage feeds the heater assembly.

4.3 DLR Demonstrators

At the DLR Institute of System Dynamics and Control, several simulator systems utilize the MDD library for inter-system communication and querying of input devices.

The DLR Robotic Motion Simulator (Bellmann et al., 2011) is a 7-axis driving and flight simulator based on an industrial robot arm (see Figure 11). The main use of this motion simulator is the evaluation of input devices such as side-sticks, steering wheels, pedals, *etc.*, as well as the test and validation of control algorithms in terms of stability and real-time capability. The control architecture of the simulator uses blocks from the MDD library in several ways:

[29]SBHS, `http://sbhs.fossee.in/`

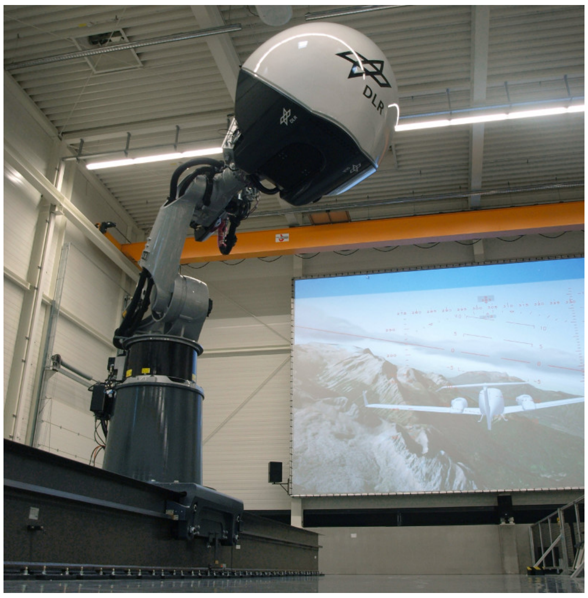


Figure 11. The DLR Robotic Motion Simulator.

- Input devices such as force-feedback steering wheels are connected via CAN bus and integrated in the software framework via the CAN blocks; the same applies for a force-feedback side-stick.
- Other, consumer based input devices such as pedals or Airbus styled flight controls are connected via the `JoystickInput` block.
- The control architecture for the robot consists of two Modelica simulations on two different computers: First, the real-time path planning running on a real-time Linux system controlling the movements of the robot, and second, the control panel running on a standard Windows system. The control panel is used to change parameters such as washout filter modes (the washout filter maps the movement of road vehicles / airplanes to the workspace of the simulator) and gives an overview on the actual robot's position and telemetry. All real-time critical communication (*e.g.*, the simulated road vehicle / airplane forces and angular velocities inputs for the real-time path-planning, or the control panel I/O) are communicated via the UDP blocks and the serial packaging system.

Figure 12 shows the inside of the simulator cabin. The instrumentation package can be adapted for different simulation types or for testing different input concepts. An on-board computer is used to query input devices, to display information on control screens, and to project the pilot's outside view visualization on the embracing concave dome shell. These tasks are performed using Modelica models, where the `SynchronizeRealtime` block is used for real-time synchronization. In addition, communication with the other simulation components is performed partly via the UDP blocks.

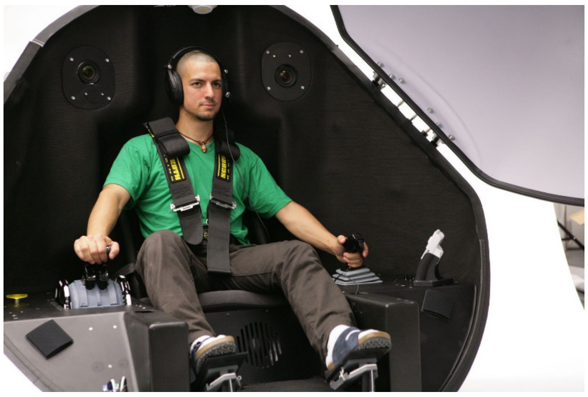


Figure 12. View into the simulator cabin of the DLR motion simulator. The instrumentation package is replaceable, so that the simulator cabin can be easily adapted for different simulation types, *e.g.*, for driving or flight simulation.

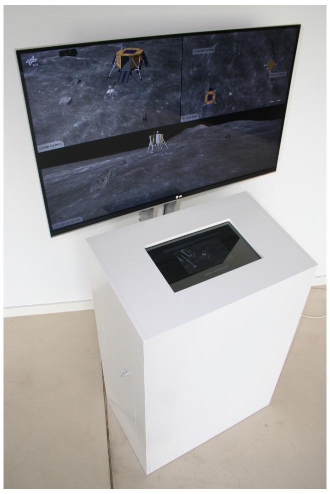


Figure 13. DLR ROBEX technology demonstrator.

Figure 13 shows the ROBEX demonstrator which was developed as a technology demonstrator for a science exhibition. This demonstrator allows the user to command a rover on a scientific lunar mission. The mission's goal is to pick up a sensor package from a nearby lander and to place it on a marked position on the lunar surface. The user controls the rover via an Android App, which runs on a tablet computer in front of the simulator screen. On the screen, the visualization of the rover is displayed. The underlying Modelica simulation performs the multi-body simulation of the rover and utilizes the *DLR Visualization* library to display the rover and the scenery. It uses the UDP blocks to communicate with the tablet computer and the `SynchronizeRealtime` block to adjust the simulation speed.

In very similar ways, the library is also used in several other simulator and demonstrator systems, *e.g.*, a drilling rig training simulator, several desktop flight simulators, or a rover software-in-the-loop development environment.

5 Outlook

The *Modelica_DeviceDrivers* library is a tried and tested library, which can support a wide range of application scenarios. During its development, valuable experience on interfacing Modelica with external C code has been gained. Thus, the source code can also serve as an example for anybody who is interested in applications, which require a more complex integration of Modelica code with external C code.

Considerable development efforts have been spent on improving the Modelica compliance of the library. Still, there are open issues and one may see the library as a testbed, which stresses Modelica's external function interface to the limit. On one hand, experiences gained thereby can provide inputs for further enhancements to the Modelica standard specification, on the other hand, further efforts in the library development can improve the level of standard-compliance. However, since backwards compatibility is a strong objective in the library development, non-backwards compatible changes for the sake of better standard-compliance will not be introduced lightly.

Naturally, there is a large pool of conceivable feature extensions to the library, due to the myriad number of available external devices and communication protocols. A frequent request is to extend the communication abilities beyond the capabilities of the available `SerialPackager`. There exists a huge choice of data serialization formats that could be utilized for this purpose (*e.g.*, LCM or MessagePack). Particularly, with regard to the Internet of Things (IoT) technology becoming more important, improving communication capabilities is a worthy goal. Similarly, supporting embedded systems beyond the prototypical work is very attractive in that perspective.

Acknowledgements

This work has been supported by Vinnova in the ITEA3 OPENCPS projects, and in the RTISIM project. Support from the Swedish Government has been received from the ELLIIT project, as well as from the European Union in the H2020 INTO-CPS project. The Open Source Modelica Consortium supports the OpenModelica development.

Finally, the authors would like to thank everybody who has contributed to the library, either by providing feedback and suggestions, or by direct contributions to the implementation of the library, particularly, Miguel Neves, Dominik Sommer, Rangarajan Varadan, and Dietmar Winkler.

References

Inderpreet Arora, Kannan M. Moudgalya, and Sachitanand Malewar. A low cost, open source, single board heater system. In *4th IEEE International Conference on E-Learning in Industrial Electronics (ICELIE)*, November 2010. doi:10.1109/ICELIE.2010.5669868.

Tobias Bellmann. Interactive Simulations and advanced Visualization with Modelica. In Francesco Casella, editor, 7th *Int. Modelica Conference*, Como, Italy, September 2009. doi:10.3384/ecp09430056.

Tobias Bellmann, Johann Heindl, Matthias Hellerer, Richard Kuchar, Karan Sharma, and Gerd Hirzinger. The DLR Robot Motion Simulator Part I: Design and Setup. In *2011 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4694–4701. IEEE, May 2011. doi:10.1109/ICRA.2011.5979913.

Torsten Blochwitz and Thomas Beutlich. Real-Time Simulation of Modelica-based Models. In Francesco Casella, editor, 7th *Int. Modelica Conference*, Como, Italy, September 2009. doi:10.3384/ecp09430119.

Matthias Hellerer, Tobias Bellmann, and Florian Schlegel. The DLR Visualization Library - Recent development and applications. In Hubertus Tummescheit and Karl-Erik Årzén, editors, 10th *Int. Modelica Conference*, Lund, Sweden, March 2014. doi:10.3384/ecp14096899.

Albert S. Huang, Edwin Olson, and David C. Moore. LCM: Lightweight Communications and Marshalling. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2010)*, pages 4057–4062, October 2010. doi:10.1109/IROS.2010.5649358.

Modelica Association. Modelica—A Unified Object-Oriented Language for Physical Systems Modeling v3.2. Standard Specification, March 2010. available at http://www.modelica.org/.

Modelica Association. Modelica—A Unified Object-Oriented Language for Systems Modeling v3.3 Revision 1. Standard Specification, July 2014. Available at http://www.modelica.org/.

Martin Otter, Bernhard Thiele, and Hilding Elmqvist. A Library for Synchronous Control Systems in Modelica. In Martin Otter and Dirk Zimmer, editors, 9th *Int. Modelica Conference*, Munich, Germany, September 2012. doi:10.3384/ecp1207627.

Niklas Worschech and Lars Mikelsons. A Toolchain for Real-Time Simulation using the OpenModelica Compiler. In Martin Otter and Dirk Zimmer, editors, 9th *Int. Modelica Conference*, Munich, Germany, September 2012. doi:10.3384/ecp12076839.