Generating FMUs for the Feature-Based Language Bloqqi

Niklas Fors[1] Joel Petersson[2] Maria Henningsson[2]

[1]Department of Computer Science, Lund University, Sweden, niklas.fors@cs.lth.se
[2]Modelon AB, Sweden, {joel.petersson, maria.henningsson}@modelon.com

Abstract

In this paper, we describe how we generate Functional Mock-up Units (FMUs) for the automation block language Bloqqi. This allows Bloqqi control programs to be tested with simulations of the physical processes they control. The physical process can be specified in any tool that supports the Functional Mockup-Interface (FMI) standard. For example, we have successfully run Bloqqi programs together with Modelica models exported as FMUs. Bloqqi programs execute at discrete times, and we describe how this is handled in the implementation of the `DoStep` function, specified in the standard.
Keywords: FMI; Bloqqi; code generation

1 Introduction

Automation control systems are usually programmed using block diagrams. These diagrams contain blocks and connections that describe the data-flow between the blocks. Examples of languages include Function Block Diagrams from the standard IEC 61131, Bloqqi (Fors and Hedin, 2016) and ControlBuilder from the company ABB. Before these programs are deployed in a plant, they are tested, often with test cases written in the language itself. For more complex dynamic processes, there are other tools more suitable for describing the dynamics of the processes, for example, the modeling language Modelica (Modelica, 2018). Thus, it would be useful to write the control program as a block diagram and specify the model of the physical process in a modeling language and test them together.

The *functional mock-up interface* (FMI) (Blochwitz et al., 2012) is a standard that allows dynamic models described by different tools to be used together. For example, one part of a composed model can be exported by one tool and another part can be exported by another tool. Many Modelica tools allow models to be exported as *functional mockup units* (FMUs).

In this paper, we describe how programs in the block language Bloqqi can be exported as (Co-Simulation source) FMUs. Bloqqi is an object-oriented language for automation control systems that supports the specialization mechanisms *connection interception* and *block redeclaration*. The language also has support for feature mechanisms, making it possible to describe variants in libraries that the user can select from. The Bloqqi tools are open source and covered by the Modified BSD License.

The contribution of this paper is partly a description of how Bloqqi programs are executed by describing how they are translated to C code. The C code can be generated without any external dependencies, making it easy to integrate the code into existing systems and run it on embedded systems. The paper also contributes with a description of how the C code is adapted for the FMI standard, and how the `DoStep` function defined in the standard is implemented. A Bloqqi FMU is a bit different than a FMU for a continuous-time physical system, since a Bloqqi FMU only executes at discrete time steps according to the sampling period. We also give some examples of Bloqqi programs exported as FMUs and simulated together with models specified in Modelica.

This paper begins with background on FMI and SSP in Section 2. SSP is a standard for specifying the composition of FMUs. Then, the Bloqqi language is introduced in Section 3, and Section 4 describes how Bloqqi diagrams are translated to C programs. These C programs are then adapted for FMI, which is described in Section 5. Examples of how we have used FMUs are described in Section 6. The paper ends with a discussion of related work in Section 7 and conclusions in Section 8.

2 Background

2.1 FMI

The Functional Mock-up Interface (FMI) is a tool-independent standard with support for both model exchange and co-simulation of dynamic models (Blochwitz et al., 2012). Version 1.0 of the standard was released in 2010, followed by version 2.0 in 2014. Using the FMI standard, models can be shared across all the 100+ tools that are currently supporting the standard. FMI uses a combination of XML-files and compiled C-code to create a Functional Mock-up Unit (FMU). An FMU is a zip-file with two major parts: a model description in XML-format, and a number of compiled binaries. Each FMU

can contain multiple binaries to support different platforms. The standard defines two kinds of FMUs: Model Exchange FMUs and Co-Simulation FMUs. A Model Exchange FMU requires an external solver for the FMU to be simulated. A Co-Simulation FMU on the other hand has a solver embedded.

2.2 SSP

In order to create simulation models for complex systems it is often convenient to separate the system into its components, and perform the necessary simulations in a tool suitable for the domain. However, with increasingly complex systems and large dependencies between components, full system simulations are essential. System Structure and Parameterization (SSP) (Köhler et al., 2016) is a new open standard (under development by the Modelica Association) defining a standardized format for the connection structure of a network of FMUs. It also defines a standardized way to store and apply parameters to such a structure. Utilization of the FMI and SSP standard will allow for components to be developed in the tool best suited for the domain, while still allowing for system simulations including all system components.

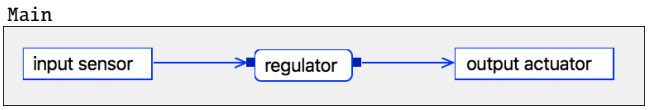
3 Bloqqi

Bloqqi is a data-flow language (Fors and Hedin, 2016; Fors, 2016) for programming the control subsystem part of automation systems. The language is a prototype language used for experimenting with language mechanisms for reuse. It has been developed in collaboration with ABB Automation Systems and specifically with the department responsible for development of tools for distributed control systems. Existing tools are based on the IEC 61131 standard for automation languages, which contains a family of five programming languages. The Bloqqi language is inspired by the Function Block Diagram language defined in this standard.

In Bloqqi, programs are specified as diagrams, where each diagram consists of blocks and connections between them that describe the data-flow. The blocks can be instances of other diagrams (which may be user-defined), making it possible to create hierarchical programs.

Bloqqi programs are executed periodically. For example, a program may run ten times per second. In each period, input values are read that are used to compute control signals. The control signals are then the output values of the program. Typically, input values are read from sensors and output values are sent to actuators that control the physical process.

This is illustrated in the diagram in Figure 1. Input values are blocks prefixed with the `input` keyword and the prefix `output` is used for output values. The block `regulator` is a normal block with an input port and an output port and is defined by another diagram. As can be seen in the figure, Bloqqi has both a visual and a texutal syntax. The textual syntax is used as the serialization format when the diagrams are stored.




```
diagramtype Main {
  input sensor: Int;
  output actuator: Int;
  regulator: Regulator;
  connect(sensor, regulator.in);
  connect(regulator.out, actuator);
}
```

Figure 1. A simple program in Bloqqi with an input value, a block and an output value. Both the visual and textual syntax are shown.

3.1 Diagram Inheritance

The Bloqqi language is similar to Modelica in that it supports diagram inheritance. A diagram S can extend another diagram T, making all blocks, connections, and parameters reusable in the subtype S. The subtype can also introduce new blocks, connections and parameters. There are two specialization mechanisms in Bloqqi: *connection interception* (Fors and Hedin, 2014) and *block redeclaration*. Connection interception allows the subtype to replace a connection defined in a supertype to instead go via a block added in the subtype. The connections in Bloqqi are directed, in contrast to Modelica where connections are undirected. Block redeclaration allows the subtype to specialize the type of a block that is defined in a supertype, similar to redeclare in Modelica.

3.1.1 Inheritance Example

Inheritance and the interception mechanism are illustrated in Figure 2. The diagram `P` describes a proportional regulator (`P`) with three input parameters: reference value (`r`), mesaured value (`y`) and the proportional coefficient (`kP`). The output parameter `u` is computed by taking the difference between the reference value and the measured value (the error), and then multiplying that with the coefficient. An instance of diagram `P` would then show the input parameters as input ports and the output parameter as output port.

The P-regulator is then extended with an integral part to incorporate the history of the error as well, as seen for diagram `PI`. The grey parts with dashed lines of the diagram are inherited from the supertype `P` and the blue parts with solid lines are declared locally in `PI`. The history of the error is stored in the block `acc` that simply accumulates the error value over time. The accumulated value is then multiplied with the integral coeffi-

DOI 10.3384/ecp18148245

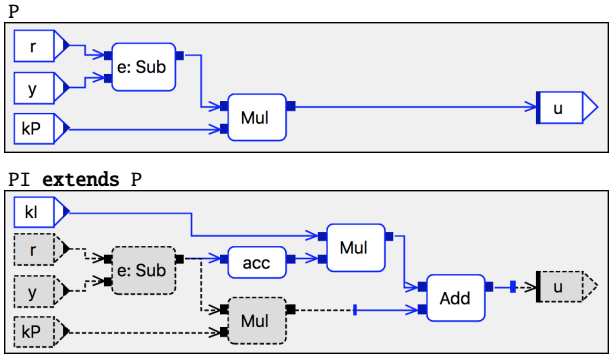


Figure 2. P-regulator (P) that is extended with an integral part in diagram PI. The connection to the output parameter u defined in P is intercepted in the subtype PI to add the integral part.

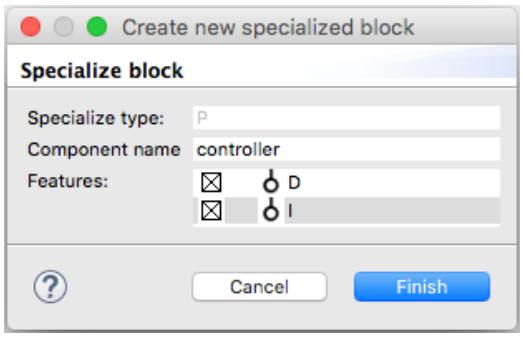


Figure 3. Feature-wizard for diagram P when the integral and derivate part are specified as features using the feature-based mechanisms in Bloqqi.

cient, which is added to the control signal. Here, the addition block intercepts the connection defined in the supertype P to go via locally declared addition block.

The accumulator block in the diagram PI contains a state to store the value to be used in the next execution period. The Bloqqi language has support for states in the form of variables, which are stored between the periods. When the variable is read, the value from the previous period is used, and when the variable is written to, a new value is stored to be used in the next period. Thus, the accumulator block contains a variable to accumulate the error value.

Currently, the Bloqqi language does not allow data-flow cycles. Instead, the user needs to break the cycle by introducing a variable.

3.2 Features

Bloqqi has also feature-based language mechanisms (Fors and Hedin, 2016). These allow the library developer to add optional features to diagrams, which can be selected by the library user, when the diagrams are instantiated. For example, consider the P-regulator in Figure 2; the integral part and the derivative part (not shown) can be seen as optional features to the diagram P. Speciyfing this using the feature-based mechanisms in Bloqqi would show an automatically generated wizard when the diagram P is instantiated as a block, as can be seen in Figure 3. The user can then select what features the block contains (in this case, both the features are selected).

4 Code Generation

Bloqqi programs are executed by first compiling them to C code. Running a Bloqqi program one period amounts to calling a C function. The function needs to know the input values and the state variables from the previous period, and store the new state variables to the next period and the output values. All the values are stored in a C struct and passed around as parameters. For example, the following C code will run the program in Figure 1 one period with the input value 10 for the `sensor` and printing the output value `actuator`.

```
Main_VARS v;
v.input.sensor = 10;
bloqqi(&v);
print(v.output.actuator);
```

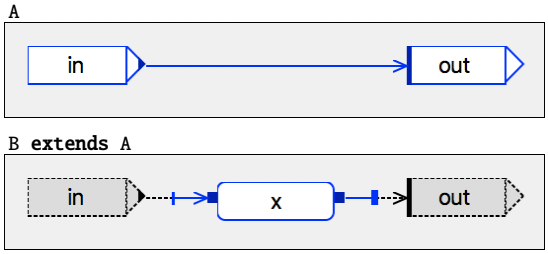
The Bloqqi compiler can generate a *driver function* that calls the `bloqqi` function a fixed number of times per second.

Each diagram is translated to a C function and each block is translated to a function call. The connections determine how the data flows between the function calls. Input parameters are mapped to function parameters and output parameters are mapped to the return value. Since diagrams can have several output parameters and C functions can only have one return value, the output parameters are wrapped in a struct and a value of this struct is returned. For example, the code implementing the diagram `Main` in Figure 1 is as follows:

```
void Main(
    Main_INPUT* _input,
    Main_OUTPUT* _output) {
  Regulator_RES regulator
    = Regulator(_input->sensor);
  _output->actuator = regulator.out;
}
```

The function has two parameters: a struct representing input values and another struct representing output values. The block `regulator` is translated to a function call and the value of the output port is stored in the struct `Regulator_RES`. Here, the only argument to the `Regulator` call is the value of the input port. If this block would have contained input values and output values, these would also be passed as arguments.

The entry point of a Bloqqi program is the diagram called `Main` with no parameters. The generated function `bloqqi` will just call the generated function `Main`.




```
diagramtype A(in: Int => out: Int) {
  connect(in, out);
}
diagramtype B extends A {
  x: X;
  intercept out with x.in, x.out;
}
```

Figure 4. Diagram `B` extends diagram `A` and intercepts the connection declared in `A`.

```
diagramtype B(in: Int => out: Int) {
  x: X;
  connect(in, x.in);
  connect(x.out, out);
}
```

Figure 5. Inheritance removed for diagram `B` defined in Figure 4 in the flattening process before C code is generated.

4.1 Inheritance Flattening

Before the C code is generated, the inheritance is removed by the source-to-source transformation *inheritance flattening*. This means that the inheritance is removed by copying all declarations in the supertype to the subtype. This transformation is possible since all block types are known statically.

For example, consider the two diagrams in Figure 4. Here, diagram `B` extends diagram `A` and intercepts the connection, from the input parameter (`in`) to the output parameter (`out`), and adds an extra block `x` in-between. The flattening transformation will remove the inheritance and transform the diagram `B` to a diagram without inheritance, as shown in Figure 5. We can see that the flattened diagram has the parameters declared in diagram `A`, the block `x` and the corresponding connections. The code generation will use this flat diagram when generating code.

4.2 Features

The feature mechanisms are based on inheritance and anonymous diagrams. The latter allows the block type to be an anonymous subtype of a diagram, which is illustrated in the following block declaration:

```
b: Block { ... };
```

The type of block `b` is an anonymous subtype of diagram `Block` and may have the same content as a normal subtype. Before code generation, these anonymous subtypes are given a unique name and moved to the same scope level as all other diagrams.

4.3 Simple Integration

The C code generated by the Bloqqi compiler is simple to integrate into an existing system. The generated code does not depend on any external library, only timing functionality from the C POSIX library when the driver function is generated. The Bloqqi compiler can also generate code without a driver function with minimal dependencies. In that case, the only dependency is to the header file `stdbool.h` in the standard library. This allows the code to run on any ordinary operating system. The generated C code is compatible with the C99 standard.

4.4 Embedded Systems

We have successfully run Bloqqi programs on an Arduino Uno, which is a single-board with a microcontroller, and on a Raspberry PI, which is a single-board computer running Linux. The generated driver function was used on the Raspberry PI and an adapted driver function was written specifically for Arduino, since Arduino does not have any operating system and thus not support the C POSIX Library. The adapted driver function called the generated main function of the Bloqqi program and added a delay between the periods using the Arduino library.

4.5 Implementation

There are two tools for the Bloqqi language: a graphical editor and a compiler. Both these tools use the textual syntax as the serialization format. The editor visualizes the program and allows the user to change it, and the compiler generates C code for the program.

The semantical analysis for both the editor and compiler is specified using the metacompiler JastAdd (Ekman and Hedin, 2007), which supports the semantic formalism reference attribute grammar (Hedin, 2000) (RAGs). RAGs make it easy to reuse the semantic specification between the tools. The analysis includes inheritance flattening, which is briefly described in Section 4.1, that is used both in the editor and the compiler. When a diagram is opened in the editor, the editor shows the flattened diagram, which includes blocks, connections, etc., defined in the supertype (as seen in Figure 4). Thus, the shown diagram is computed based on the semantics of the language.

DOI
10.3384/ecp18148245

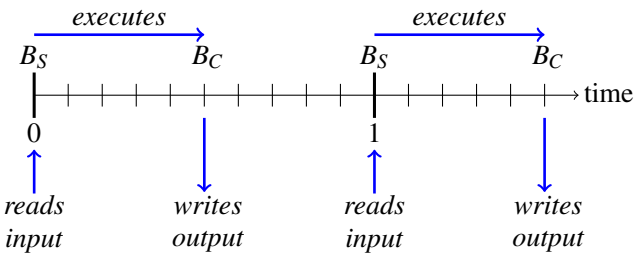


Figure 6. Simulation of a Bloqqi FMU with sampling period of 1 second and execution time of 0.5 seconds. The Bloqqi program starts executing at B_S and completes at B_C. The inputs are read at B_S and the outputs are set at B_C. The time on the axis is the simulated time.

5 FMU Generation

A Bloqqi program can be exported as a Co-Simulation FMU, which makes it easy to use Bloqqi programs together with models specified with other tools that support the FMI standard. The input and output values of the Bloqqi program are mapped to corresponding inputs and outputs of the FMU. The FMU executes periodically and the period can be set by the parameter `sampling-period`. It is also possible to simulate an estimated execution time of the Bloqqi program, which is set using the parameter `execution-time` and it will delay the output values with `execution-time` seconds. This is illustrated in Figure 6. The FMU can be generated as a *source FMU*, which includes the generated C code in the FMU.

The FMU generation for Bloqqi[1] is implemented using a port by Christopher Brooks[2] of FMU SDK[3] by QTronic which runs under Linux and MacOS. The Bloqqi compiler generates C code, as described in Section 4, and adds wrapper code for the FMU SDK and our own `DoStep` function. The Bloqqi compiler also generates an XML file describing the structure of the FMU.

5.1 Master Algorithm

When simulating a system of Co-Simulation FMUs, a *master algorithm* is responsible for exchanging data (inputs and outputs) between the FMUs at discrete *communication points*. The FMUs are then solved independently from each other between the communication points with their respective solver. The master algorithm uses functions, defined in the standard, for getting and setting inputs and outputs of the FMUs. Each FMU also defines the function `DoStep` that simulates one *communication step* with a given *communication step size* (which might vary between the calls). This function is called by the master algorithm. The following code illustrates one simple master algorithm with a fixed communication step size (`hc`) that simulates between the times `tStart` and `tEnd`:

```
curTc = tStart
while (curTc < tEnd) {
  read outputs from all FMU:s
  set inputs to all FMU:s
  call DoStep(curTc, hc) on all FMU:s
  curTc += hc
}
```

When using this algorithm together with a Bloqqi FMU, it is important that the communication points reflect the discrete times at when the Bloqqi program starts and completes execution. The master algorithm should exchange data every time the Bloqqi program starts an execution and when it completes an execution. Otherwise, the Bloqqi FMU will use old input values and delay its output values.

For the example shown in Figure 6, where the sampling period is 1 second and the execution time is 0.5 seconds, a master algorithm with the communication step size of 0.5 seconds will work fine without introducing any delays.

5.2 Implementation of DoStep Function

As described earlier, we have used FMU SDK but implemented our own `DoStep` function, with the goal of not introducing any unnecessary input/output delays. Our implementation internally keeps track of the next time the Bloqqi FMU should execute (according to the parameter `sampling-period`). If the Bloqqi FMU has started to execute, but has not yet completed, the function also keeps track of when it will complete. The function uses two variables to represent these times (`starts` and `completes`).

The function `DoStep` is called with the current communication point (`curTc`) and a communication step size (`hc`) by the master algorithm, and it will check if any of the time variables are before `curTc+hc`. An implementation sketch of `DoStep` is shown in Figure 7. As can be seen, if the Bloqqi program should start executing, the function will get the FMU inputs and set them to the inputs of the Bloqqi program, and then run the Bloqqi program one period. After that, the times for when the execution completes and when the next execution starts are calculated. It might happen that the execution completes during the same invocation of `DoStep`, and this is handled afterwards with the call to `set_outputs`, which sets the outputs of the FMU as the outputs of the Bloqqi program.

The `DoStep` function will only start executing if the simulated time has *passed* the value of `starts`. This is illustrated in Figure 8. Here, the second invocation of `DoStep` is between the times t_1 and t_2, and the invocation will not start the Bloqqi execution since the simulated time has not passed t_2. However, the third invocation will start the Bloqqi execution since it passes the time t_2.

[1] `https://bitbucket.org/bloqqi/bloqqi-fmi`
[2] `https://github.com/cxbrooks/fmusdk2`
[3] `https://qtronic.de/en/fmusdk.html`

```
// Internal state for Bloqqi program
Main_VARS v;

// When the next execution starts
starts = 0.0

// When a started execution completes.
// The value ∞ is used when there is
// no active execution.
completes = ∞

void DoStep(
    curTc, // current communication point
    hc     // communication step size
    ) {

  // ... error checking ...

  nextTc = curTc + hc

  // If an execution started in a previous
  // call to DoStep and completes during
  // this call.
  if (completes <= nextTc) {
    set_outputs(v.output)
    completes = ∞
  }

  if (starts < nextTc) {
    // Start a new execution
    v.input = get_inputs()
    bloqqi(&v)
    completes = starts + EXECUTION_TIME
    starts    = starts + SAMPLING_PERIOD

    // If the execution completes during
    // the same call to DoStep as when
    // it was started.
    if (completes <= nextTc) {
      set_outputs(v.output)
      completes = ∞
    }
  }
}
```

Figure 7. Implementation sketch for `DoStep`

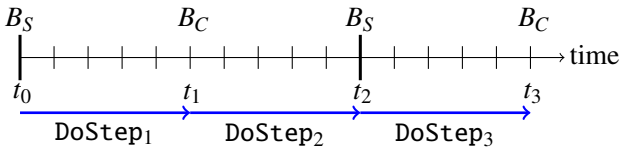


Figure 8. The execution starts when the simulated time has *passed* the value of the variable `starts`. This means that the first and third invocation of `DoStep` will start an execution, but not the second invocation. The second invocation is between the times t_1 and t_2, but it has not passed t_2, which is required for it to start executing.

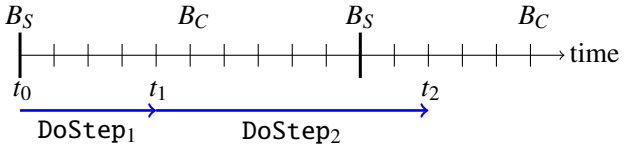


Figure 9. When the master algorithm makes the second call to `DoStep` in this example, the Bloqqi FMU should first write outputs from the previous execution before starting a new execution. This is handled by checking the variable `completes` before starting a new execution in `DoStep`.

We think that this behaviour makes it easier to use Bloqqi FMUs together with standard master algorithms, and where the Bloqqi FMU gets fresh input values when the execution starts. In the implementation of `DoStep` (Figure 7), this behaviour is captured by using the less than operator (<). On the other hand, when the output values are set, it is not needed to pass the time for when the execution completes, which is captured by using the less than or equal operator (<=). This will make the outputs visible when the execution is completed. For example, in Figure 8, the output values will be set by the first and third invocation of `DoStep`, making the outputs available at the times t_1 and t_3.

5.2.1 Handling Asynchronous Periods

It might happen that a call to `DoStep` starts an execution, but does not complete it, and that the execution is completed in a subsequent call to `DoStep`. This situation is handled by checking in the beginning of `DoStep` the variable `completes`. This code fragment also handles the situation when a previous call to `DoStep` started an execution, and the current call both completes the previous execution and starts a new execution. This is illustrated in Figure 9. Here, the first call to `DoStep` by the master algorithm is from t_0 to t_1, and which starts an execution. The second call is from t_1 to t_2, and which first writes the outputs from the previous execution and then starts a new execution of the Bloqqi program. In this example, the output values will be delayed (to time t_2 for the first execution) and old input values will be used (from time t_1 for the second execution). This delay is undesirable, but cannot be avoided without changing the master algorithm.

DOI 10.3384/ecp18148245

The error checking in the beginning of `DoStep` checks that the communication step size is not larger than the sampling period of the Bloqqi program, which in that case raises an error (`fmi2Discard`). We have chosen this behaviour as the Bloqqi program and the controller it represents can produce faulty results, as well as large delays, if used with a too large communication step size. For example, suppose the derivative in the controller is calculated by using the slope between two samples. It is then important that input values are updated between controller execution loops, as the difference would be zero otherwise. This would not happen with a `DoStep` that includes multiple executions of the Bloqqi program using the same sample of inputs.

5.2.2 Zero Execution Time

The estimated execution time can be set to 0. In this case, the master algorithm should make small communication steps from when the Bloqqi execution starts. For instance, assume that the first time the Bloqqi FMU should start executing is at time t_0 and that the Bloqqi sampling period is h, then the master algorithm should first call `DoStep(`t_0`,` hc_0`)` with a small communication step size hc_0 (it needs to pass the time t_0). Then, the master algorithm should take a longer communication step $hc_1 = h - hc_0$ to when the Bloqqi program is to start executing again, hence, it should make the call `DoStep(`t_1`,` hc_1`)`, where t_1=$t_0 + hc_0$. If it is not possible for the master algorithm to make communication steps of variable length, then it should instead make the communication steps small enough to reduce the delay of the control signal to an acceptable level.

5.3 Selection of FMU Kind

As mentioned in Section2.1, there are two different kinds of FMUs specified in the standard, Model Exchange and Co-Simulation FMUs. There are advantages with both variants of the standard and selecting which one to use is not always straightforward. Due to its simple interface and ease of implementation, Co-Simulation FMUs are supported in most tools and provides a good platform for sharing the control models. It also provides an inherently sampled platform which reflects the sampled nature of a controller executing the Bloqqi program. For these reasons Co-Simulation FMUs were choosen as an appropriate first target kind for the Bloqqi FMUs.

Generally when performing simulations with FMUs an external entity will have control of the sampling time, and for the Bloqqi FMUs it cannot be assumed that the sampling times coincide with the discrete times of the Bloqqi program. The extreme case, when the sampling period of the master algorithm is larger than the sampling period of the Bloqqi FMU, is handled through a `FMI2Discard` of the `DoStep`. The mismatch of sampling times will inevitably lead to unnecessary delays in the control algorithms during simulations. The additional delays brought by the sampling might have a negative effect on the stability region of a controlled process that would not be present in the actual physical system, which could lead to inaccurate results of the simulations. For these reasons the possibilities of using Model-Exchange FMUs and event states to assure that the Bloqqi execution points are executed at the correct times would be interesting to investigate during the continuation of the project. Such an FMU could perform all its calculation, including registering its next event, while in event mode, and not contribute to the integration during the continuous time mode.

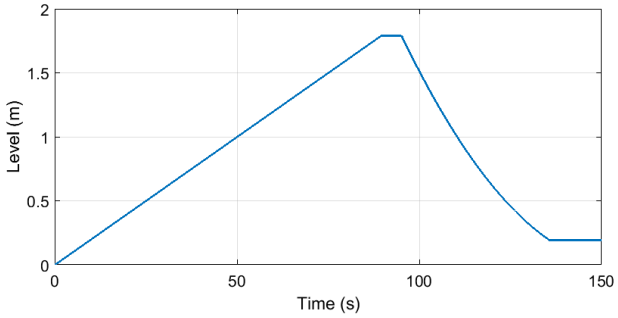


Figure 10. Simulation of the liquid level in a tank. The set point is first 1.8 meter and then changed to 0.2 meter.

6 Examples

We have exported Bloqqi programs as FMUs and run them together with plant model FMUs exported from Modelica models.

6.1 Tank Example

One example is a simple regulator for a tank of liquid. The regulator controls the liquid level using one input valve and one output valve, both valves can be opened or closed.

We have implemented the regulator in Bloqqi and modelled the tank in Modelica, and exported both models as FMUs. These two FMUs have then been aggregated (see Section 6.2) and simulated together. The simulation result can be seen in Figure 10. The figure shows how the liquid level changes over time. In this example, first, the set point of the liquid level is set to 1.8 m, and after it is reached, the set point is changed to 0.2 m.

The Bloqqi diagram for the tank regulator that opens and closes the valves is shown in Figure 11 (the diagram is called `Tank`). The parameter `setLevel` is the set point and the current liquid level is read from a sensor, which is represented by the block `levelSensor`. The blocks `lowerValve` and `upperValve` represent the actuators for the valves and take a boolean as input (`true` to open the valve and `false` to close the valve). The di-

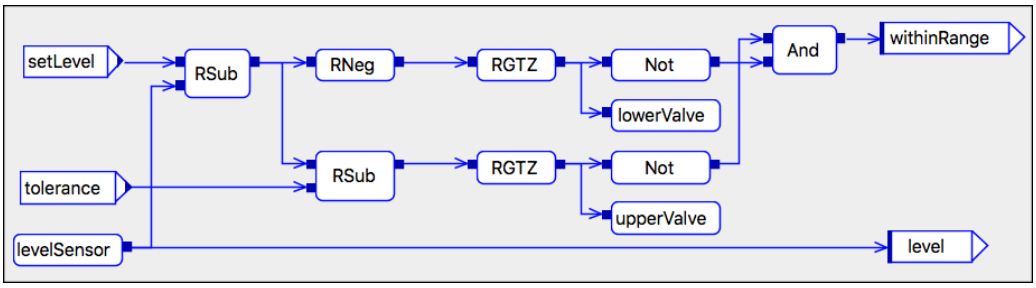


Figure 11. Simple tank regulator in Bloqqi. The block `levelSensor` reads the current liquid level. The blocks `lowerValve` and `upperValve` are actuators to the valves. The block `RGTZ` means greater than zero (for reals).

agram also has a parameter `tolerance` that allows the level to be within a range of the set point (`setLevel` ± `tolerance`). The output parameter `withinRange` yields true if this is the case. There is also another diagram `Main`, which is not shown in the figure, that has the `Tank` diagram as a block. The `Main` diagram will use the output parameter `withinRange` to change the set level.

The tank model in Modelica is specified as a *mass balance equation* (Fritzson, 2004), where the curent liquid level is specified in terms of the input flow and output flow of the tank. The equations for the tank are specified in the following manner:

```
der(level) = (inFlow-outFlow)/AREA;
inFlow = if upperValveOpen
  then IN_FLOW
  else 0.0;
outFlow = if lowerValveOpen
  then (OUT_VALVE_AREA)*sqrt(2*9.82*level)
  else 0.0;
```

Thus, the time derivative of the current liquid level is defined in terms of the difference in flow. The input flow depends on if the input valve is open and the output flow depends on if the output valve is open. The output flow is also dependent on the pressure, thus, on the current liquid level. This can be seen in the simulation results in Figure 10, where the emptying of the tank from 1.8 m to 0.2 m is not a straight line.

6.2 Composing FMUs

We have used FMI Composer[4] from Modelon for co-simulation of the Bloqqi FMU of the controller and the Modelica FMU of the tank. FMI Composer supports the SSP standard and can convert a system of FMUs to a aggregated FMU. The simulation of the composed FMU is then run in another tool, for instance, the FMI Toolbox for Simulink, also from Modelon, that we have used for the tank simulation shown in Figure 10.

For example, the composition of the tank regulator and the tank model is shown in Figure 12. The regulator FMU has one input, the current liquid level, and two outputs, which tells if the valves should be open or not. The tank FMU uses the output from the regulator to model the current liquid level, which is the output from the tank FMU.

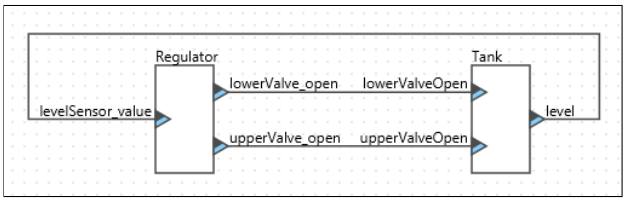


Figure 12. FMU composition of the tank regulator and tank model specified in FMI Composer.

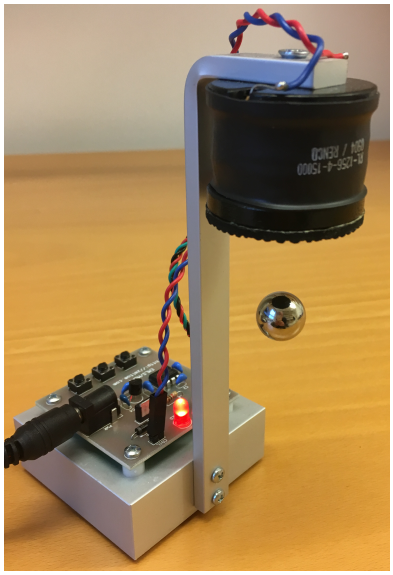


Figure 13. Magnetic levitation system.

6.3 Magnetic Levitation

Another example we have experimented with is a magnetic levitation system. The system uses an electromagnet to control a magnet in the air. This is shown in Figure 13, which is an education system from Zeltom[5].

We have run a PD-regulator defined in Bloqqi for controlling a simluation of the electromagnet, where the control signal is the voltage to the electromagnet and the set

[4] `http://www.modelon.com/products/`

[5] `http://zeltom.com/products/magneticlevitation`

DOI 10.3384/ecp18148245

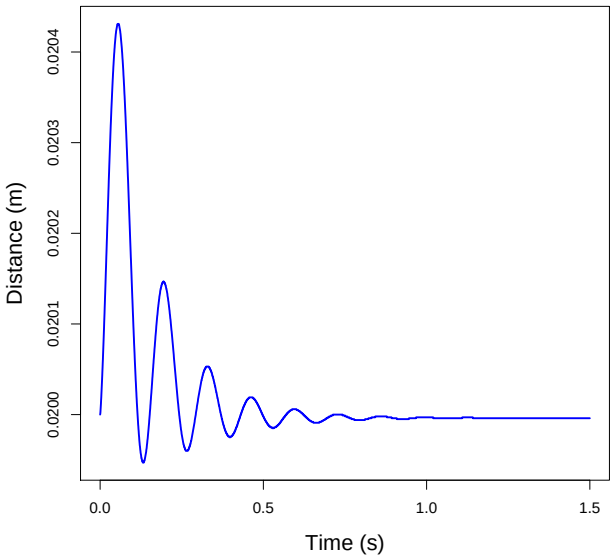


Figure 14. Simulation of magnetic levitation. The set point is 0.02 m distance between the electromagnet and the magnet.

point is the distance between the electromagnet and the magnet. The regulator has been simulated together with a Modelica model that describes the distance between the electromagnet and the magnet, given a voltage to the electromagnet. The Modelica model is written by Bernhard Thiele. Thiele has also specified a PD-regulator in Modelica for the system, which is the basis for our PD-regulator in Bloqqi. A simulation of the regulator and the model can be seen in Figure 14 (with the set point of 0.02 m and sampling period of 0.0005 s).

7 Related Work

Typically, code generation for block diagrams removes the hierarchical structure by flattening the diagrams, where each non-atomic block is replaced with its definition. Lublinerman et al. (2009) propose a technique for modular code generation, where the code generation for one diagram is independent of the context and uses minimal information about the internals of its blocks. Their technique handles diagrams that look cyclic, but when the diagrams are flattened, they are in fact acyclic. The Bloqqi compiler will not allow these kinds of diagrams, as described in Section 3.1.1.

The implementation of code generation for Bloqqi has similarities with the implementation of JModelica (Åkesson et al., 2010). Both implementations are specified using reference attribute grammars (Hedin, 2000) in the metacompilation tool JastAdd (Ekman and Hedin, 2007). In JModelica, the user selects a Modelica model to compile, and JModelica compiler will then remove the object-oriented and hierarchical structure for the selected model that results in a flat equation system. During this process, each model instance reachable from the selected model is substituted with the content of the model definition. In contrast, the Bloqqi compiler will only flatten the inheritance, by copying declarations in the supertype to the subtype, which results in a set of diagrams as described in Section 4.1. Thus, the Bloqqi compiler will retain the the hierarchy between diagrams. Both implementations make extensive use of *non-terminal attributes* (Vogt et al., 1989), which allows dynamically computed subtrees to be added to the abstract syntax tree in the compiler.

The problems, as discussed in 5.3, with representing discrete time systems, or any system including events, using the Co-Simulation FMI technology with regards to matching the master algorithm's sampling with the internal events of the FMUs is discussed in Cremona et al. (2017). In Cremona et al. (2017), a suggestion to add new step function called "doStepHybrid" which allows for an early return of the step function. This in combination with a possibility for the master to interrogate the system using functions like `getMaxStepSizeHybrid` would allow for the Bloqqi FMU to assure that each sample and execution event is properly matched. Additionally the concept of clocks and hybrid Co-Simulation is included in the alpha feature list of FMI 2.1 that was announced 2017-12-18 (fmi standard, 2017), this to support synchronization of variable changes across FMUs, and allow for co-simulation with events. Cremona et al. (2017) also proposes an integer representation of time in order to increase the accuracy of the time representation in simulation of FMUs.

Additional problems related to synchronization of discrete-time models is presented in Franke et al. (2017). Where a synchronus discrete-time extension to the FMI-standard is proposed to enable synchronization between FMUs with the environment, and other FMUs. The proposal includes a possibility to declare clocks and discrete-time states in the `modelDescription.xml`, and enables the environment to activate clocks in order to synchronize the environment with other FMUs.

TrueTime (Cervin et al., 2003) is a tool for simulating control systems, where it is possible to simulate task scheduling and network transmissions. These aspects are not covered in the Bloqqi FMUs, but it would be possible, for example, to model network delays as other FMUs connected to the Bloqqi FMUs.

8 Conclusions

We have in this paper described how Bloqqi programs can be exported as Co-Simulation FMUs. This is done by translating the programs to C code, which are then wrapped as FMUs. In this translation, each Bloqqi diagram is translated to a C function and each block (an instance of diagram) is translated to a function call. Be-

fore the translation, the inheritance structure is removed (but the hierarchical structure is retained). The paper also describes how the function `DoStep` is implemented for Bloqqi FMUs to handle that Bloqqi programs execute at discrete time, according to the sampling period and the estimated execution time.

Exporting Bloqqi programs as FMUs allow them to be easily imported to a simulation environment to be tested together with a simulation of the physical process. The physical process can be specified with any tool that supports FMI export/import. We have successfully tested running Bloqqi programs as FMUs together with FMUs exported from Modelica models.

In the future, we would like to support more of the FMI standard, for example, the possibility to make rollback, which is useful for simulating a larger class of FMU compositions (Broman et al., 2013). It would also be useful to generate Model-Exchange FMUs, and not only Co-Simulation FMUs. This would allow the Bloqqi FMU to tell the master algorithm about the discrete times at when it will start and complete its execution. We would also like to experiment with more and larger examples.

Acknowledgements

This research was supported by the Swedish Governmental Agency for Innovation Systems (VINNOVA), within the strategic innovation program Process Industrial IT and Automation, under contract number (2017-02371). We thank Bernhard Thiele for the magnetic levitation example, and Görel Hedin for comments on earlier draft of this paper, and Ulf Hagberg, Christina Persson, Stefan Sällberg and Alfred Theorin at ABB for sharing their expertise about the ABB tools for control systems.

References

Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Implementation of a Modelica compiler using JastAdd attribute grammars. *Science of Computer Programming*, 75(1-2):21–38, 2010.

Torsten Blochwitz, Martin Otter, Johan Åkesson, Martin Arnold, Christoph Clauss, Hilding Elmqvist, Markus Friedrich, Andreas Junghanns, Jakob Mauss, Dietmar Neumerkel, et al. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *9th International Modelica Conference*, 2012.

David Broman, Christopher X. Brooks, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. Determinate composition of fmus for co-simulation. In *Proceedings of the International Conference on Embedded Software, EMSOFT 2013*, pages 2:1–2:12, 2013.

Anton Cervin, Dan Henriksson, Bo Lincoln, Johan Eker, and Karl-Erik Årzén. How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime. *IEEE Control Systems Magazine*, 23(3):16–30, June 2003.

Fabio Cremona, Marten Lohstroh, David Broman, Stavros Tripakis, and Edward A. Lee. Hybrid Co-simulation: It's About Time. Technical report, Electrical Engineering and Computer Sciences University of California at Berkeley, April 2017.

Torbjörn Ekman and Görel Hedin. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, 2007.

fmi standard. Functional mock-up interface, 2017. URL `http://fmi-standard.org/`.

Niklas Fors. *The Design and Implementation of Bloqqi - A Feature-Based Diagram Programming Language*. PhD thesis, Lund University, October 2016.

Niklas Fors and Görel Hedin. Intercepting dataflow connections in diagrams with inheritance. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 21–24, 2014. doi:10.1109/VLHCC.2014.6883016.

Niklas Fors and Görel Hedin. Bloqqi: modular feature-based block diagram programming. In *2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016, Amsterdam, The Netherlands, November 2-4, 2016*, pages 57–73, 2016. doi:10.1145/2986012.2986026.

Rüdiger Franke, Sven Erik Mattson, Martin Otter, Karl Wernersson, Hans Olsson, Lennart Ochel, and Torsten Blochwitz. Discrete-time models for control applications with fmi. In *12th International Modelica Conference*, 2017.

P.A. Fritzson. *Principles of object-oriented modeling and simulation with Modelica 2.1*. Wiley-IEEE Press, 2004.

Görel Hedin. Reference Attributed Grammars. In *Informatica (Slovenia)*, 24(3), pages 301–317, 2000.

Jochen Köhler, Hans-Martin Heinkel, Pierre Mai, Jürgen Krasser, Markus Deppe, and Mikio Nagasawa. Modelica-association-project "system structure and parameterization" – early insights. In *1st Japanese Modelica Conference*, 2016.

Roberto Lublinerman, Christian Szegedy, and Stavros Tripakis. Modular code generation from synchronous block diagrams: modularity vs. code size. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 78–89, 2009.

Modelica. *The Modelica Association*, 2018. `http://www.modelica.org`.

Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. Higher-order attribute grammars. In *PLDI*, pages 131–145, 1989.