Importing FMU-3.0: challenges in proper handling of clocks

Masoud Najafi   Ramine Nikoukhah

Altair Engineerng, France {masoud,ramin}@altair.com

Abstract

Compared to FMI-2.0, FMI-3.0 provides support for events and clocks. The behavior of the FMU in the presence of events and clocks introduces new challenges for importing FMUs in block diagram environments such as **Altair Activate** and **Scicos**. This paper discusses some of these challenges and proposes implementation strategies for supporting the import of FMI-3.0 in **Activate**.
Keywords: FMI-3.0, Synchronous clock, Signal based tool, Modelica tool

1 Introduction

The Functional Mock-up Interface (FMI) (Modelica Association, 2022) has become a de-facto tool independent standard for the exchange of dynamic models and for co-simulation. FMI-3.0 (Specification, 2022) version of the standard introduces many new features that allow for more advanced modeling and support for co-simulation algorithms. Clocks allow the synchronization of events between Functional Mock-up Units (FMUs) and the simulator (importer). Several new data-types and multi-dimensional arrays are also supported (Junghanns et al., 2021).

Activate is a modeling and simulation tool developed by Altair Engineering based on the open-source academic simulation software **Scicos** (INRIA). **Activate** environment can be used to create models of dynamical systems as signal-based block-diagrams. The basic blocks, such as FMUs can be interconnected to build complex model. This is very similar to the way diagrams are built in the SSP [1] (System Structure and Parametrization) standard.

Activate can also be used to create Modelica diagrams (Nikoukhah and Furic, 2009). The integration of the Modelica part of the model is done first by the aggregation of the Modelica components and creation of a Modelica program which is then processed by the Modelica compiler.[2] In **Activate**, the Modelica compiler provides an FMU block replacing the Modelica components in the original model.

Because of this FMI based integration of Modelica in **Activate**, **Activate** has been providing FMU import support through an **Activate** FMU block. More generally this block is also used for importing FMUs from other sources. The support of FMI-2.0 in the **Activate** environment had already been challenging and specific solutions had to be developed; the main problem being the way input-output dependencies are defined and treated in **Activate** and in FMU. See (Nikoukhah et al., 2017).

With FMI-3.0 and the introduction of the notions of clock, activation and synchronization, the FMU import in **Activate** presents new challenges. Even though the activation signals and synchronism have been part of the **Activate** semantics from the beginning, the small semantic differences between FMI-3.0 and **Activate** formalism makes it so that an FMU cannot be imported as a basic block in **Activate**. This was already not the case in some situations with FMI-2.0, as was presented in (Nikoukhah et al., 2017). With FMI-3.0, the problem becomes more involved.

This paper presents the difficulties and the solutions envisaged to provide maximum support for FMI-3.0 import in **Activate**. First a short overview of the way **Activate** handles activations (clocks) is provided and the differences with the FMI-3.0 treatment of clocks are discussed.

In Section 4, the solutions for importing FMI-3.0 in **Activate** are presented by considering different types of clocks. Each section provides an FMU example to illustrate the process.

2 Activate environment and activation signals

2.1 Double layer implementation

In the **Activate** environment, a model is constructed using blocks. The compiler however does not operate on these blocks; it interacts with Atomic Units[3] (AU). In many cases a block is associated with a single AU, but not always: a block may produce a diagram containing multiple connected AUs. This diagram produced programmatically by the block may depend on the values of the block parameters. Specifically, the choice of the AU(s), their parameters, and the topology of the diagram is specified by an OML[4] function associated with the block, which constructs the diagram based on the values of the block parameters.

The ability to programmatically instantiate an AU or a diagram of AU(s) is a powerful mechanism which is used

[1] https://ssp-standard.org/

[2] The Modelicac compiler is used to to compile and generate code for the modelica program in **Scicos**; the MapleSim compiler is used to generate an FMU in **Activate**.

[3] Also called basic blocks.

[4] A matrix based interpreted matrix-based language similar to Scilab, Octave, Matlab.

to present to the user as a block, for example through a library, a complex construction based on a diagram of AUs. The FMU block is an example of such a construction.

In general, an AU provides computational function APIs to be used by the simulator. The APIs are C functions that are called by the simulator at different stages of the simulation: computations of the outputs, of the state derivatives, of the next discrete state, etc. **Activate** compiler uses AU properties to construct the compiled structure of the model to be used by the simulator. These properties include for example the feedthrough properties of the AUs used by the compiler for proper scheduling of the activation order of the AUs. The computations done by the corresponding APIs however are transparent to the compiler.

Two very special AUs *IfThenElse* and *SwithCase* basic blocks play a fundamental role in defining conditional operations, used for example for subsampling. They are actually language constructs similar to *if* and *switch* statements in most programming languages, and are treated in a special way by the compiler. Other "special" AUs include activation sources such as the *InitialActive*, *AlwaysActive* and *SampleClock* blocks. The latter produces an activation signal containing a series of periodic events. Multiple *SampleClock* blocks can be used within a model with identical or different periods. The compiler treats them as synchronous clocks even if they don't have identical periods.

2.2 AU interactions

AUs have input and output ports. These ports are connected by links which represent the sharing of data between the ports. The value of an input port is provided by the output port linked to it. The AU of the output port computes the signal to be read by the AU of the input port, which in turn computes its outputs, when activated.

In the simple case where all the AUs are "always active" (continuous-time dynamics), the **Activate** compiler determines the order in which the AUs should be activated (their APIs called) to guarantee the signals flow properly in the network. This order is stored in the compiled structure of the model and used during simulation. It is also used for code generation.

In many cases however all the AUs in a model are not "always active". Consider for example the model of a physical plant controlled by a discrete-time controller. In such a model, some of the AUs are continuously activated (so always active) and others only at the ticks of the controller clock.

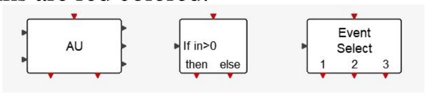
In the presence of multiple sources of activations, the compiler determines the order of block executions for all possible activation scenarios and stores them in the compiled structure to be used by the simulator[5] and the code generator.

[5] No online scheduling is ever performed by the simulator.

2.3 Activation signals and AU activation

AUs in **Activate** are activated by activation signals. The "always active" signal is an example of such a signal. An AU activated by such a signal is continuously active. Discrete activation signals define one or more isolated time instants of activation (called events). An AU activated by such a signal is activated at these discrete time instants.[6]

At the graphical level, by default the regular input and output ports are placed on the sides of the blocks and the activation input and output ports, respectively, on the top and at the bottom of the block. The activation ports and links are red colored.



The block on the left is a general AU with multiple input, output, regular and activation ports. The other two blocks are special AUs *IfThenElse* and *SwithCase* used to redirect their input activations to one of their output activation ports depending on the value of their regular inputs. These two AUs produce output activations which are synchronous with their input activations; something which is not possible with any other AU.

To simplify the construction of models at the graphical level, two mechanisms are used in **Activate** to reduce the number of activation ports and links:

- **Always active AU property:** Instead of explicitly creating a link from an always active activation source to the AU, the AU can be declared as having "always active" property.

- **Activation inheritance:** if an AU is not declared always active and does not have any activation input ports, then it can inherit its activations from its regular input signals. Specifically, it is activated by the activation signals which have activated the block which have produced its input signals.

These mechanisms are mere syntactic sugars: the corresponding activation signals are added to the model at a pre-compilation phase.

An AU may be activated by one or more activation signals, through one or more activation input ports. See Fig. 1 where the *EventDelay* block is activated by the union of two activation signals. The resulting activation signal contains then an initial event and events produced by the block itself, which are the delayed version of previous events. This model produces an activation signal consisting of a series of events evenly spaced in time (like an event clock).

For an AU having more than one activation input port, the AU is activated if any of the input ports receives an activation. In that case, the computational function API can know what activation(s) has caused the activation of the

[6] More generally an activation signal may define a union of isolated points and time intervals as activation times. But this level of generality is not pertinent to the FMI import issue considered here.

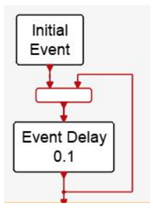


Figure 1. A new activation signal can be constructed from two or more activation signals where the activation times of the new signal is the union of the activation times of the other signals. This operation is realized by the *EventUnion* block as shown in this diagram.

AU. For example if the AU has two activation input ports, then it can be activated either because an activation signal has been received on its first input, on its second input or on both synchronously.[7] The way by which the activation has occurred is coded as an integer (the binary coding of the integer represents the input ports at the origin of the AU activation, so function call). In the case of AU with two activation input ports, the integer can take values 1, 2 and 3. The API can perform different computations for each value of the integer.

Even though the AU can be activated through different combinations of activation signals, there are no individual activation signals associated with each output port of the AU. The outputs are updated at times corresponding to the union of all the activations activating the AU. So, the activation signals associated with the outputs are all identical; the AU cannot associate individual activation signals to the AU outputs. Even if an output of the AU is not computed by the AU computational function API for a particular activation, it is considered to be up to date and is treated as if it had been recomputed (signals in **Activate** are persistent). This property is in contrast to the FMU-3 way of associating outputs to different clocks, making it impossible to represent an imported FMU as a single AU in the general case.

An AU can have activation output ports. An AU cannot generate an event which is synchronous with the event which has activated it. The generated event is delayed with respect to the activation of the block. The time delay can be set to zero, making the two events having the same time, but not synchronous. Synchronous events can only be generated by two special AUs *IfThenElse* and *SwithCase*. For details see (Campbell et al., 2010; Ext, 2022). This is another reason why a single AU cannot always represent an imported FMU-3.

The explicit treatment of Activation signals in **Activate** makes the import of FMU-3's amenable but not necessarily as single AUs (basic blocks). This was the case already for FMU-2, as we will recall in the next section. We will then show how FMU-3's can be imported as **Activate** blocks including multiple AUs. From the user point of view, this process is completely transparent. They will place the FMU block from the palette in the diagram and edit its parameter to point to the FMU to import. The block will then read the content of the FMU and programmatically create the content of the block.

[7]Here we assume that the block is not continuously active.

3 Activate FMU block for importing FMI-2.0 FMUs

This section recalls the way FMU-2's are imported in **Activate**. The process was in part presented in (Nikoukhah et al., 2017). The import of FMU-2 was a simpler task because there were no clocks and clock activations to consider; the system was always active. The main difficulty had to do with the way output/input dependencies are specified in FMI. In an AU, output/input dependencies are expressed as a vector of dependencies specifying which inputs affect any of the outputs. So, the dependency is solely a property of an input port. The reason is that an AU computes all of its outputs during a single activation, i.e., in the same API call, so all of its dependent inputs must be up to date when the call is made. An FMU on the other hand specifies output/input dependencies as a matrix specifying which output depends on which known variables including individual inputs. The FMU provides routines that allow the computation of output ports separately and take advantage of variable caching.

One way to deal with this discrepancy is to simply project the matrix of dependencies provided by the FMU into a vector of dependencies as required by **Activate**. This conservative approach properly assigns dependencies in **Activate** but "loses" information along the way. When one or more FMUs are imported in an **Activate** model, this may lead to the detection of algebraic loops by the **Activate** compiler that are not true algebraic loops (artificial algebraic loops). The result is that valid algebraic-loop-free models may end up not compilable by **Activate**.

There is no solution to this problem as long as the FMU block is to be implemented as a single AU. But as it was stated previously, **Activate** blocks can implement a diagram of AUs, the topology of which can depend on block parameters. It turns out, (Nikoukhah et al., 2017), that the matrix output/input dependency information provided by the FMUs can be implemented by a properly constructed diagram of AUs. The diagram would include a distinct AU associated with each output port, in charge of computing the output, and the diagram constructed so that its topology reflects the output input dependencies. Consider for example an FMU with 2 inputs and 2 outputs where the only output input dependency is that the first output depends on the first input. Then the FMU block could create the diagram shown in Fig. 2.

The dependency information is provided in the FMU XML file, which is available as a block parameter of the **Activate** FMU block. By reading and parsing the XML inside the FMU, the block generates a diagram of AUs. The diagram contains a central AU, always present, and an AU associated with each FMU output. The input ports of

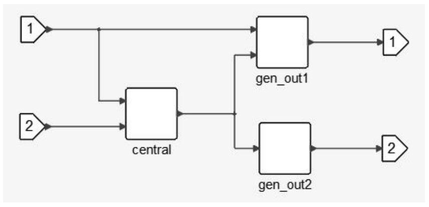


Figure 2. In this example, the first output depends directly on the first input. The second output does not depend directly on any input.

these AUs and their connections are tailored to the dependency information read from the XML file. In particular the AU associated with an output will have an input corresponding to an input of the FMU only if the corresponding output input dependency property is true.

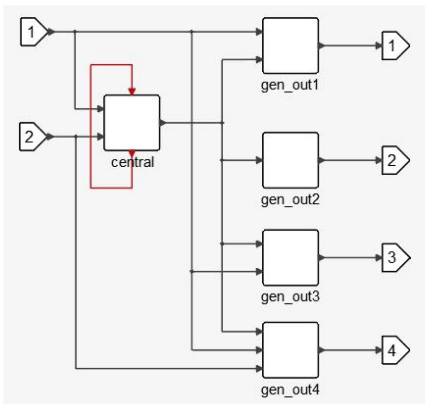


Figure 3. A central AU, always present, handles the state update tasks of the FMU and provides the FMU structure to the AUs in charge of computing the outputs. The output AUs are only connected directly to the inputs of the block if there is a corresponding output input dependency as specified in the FMU XML file.

The central AU includes the simulation APIs for state derivative computation and discrete state updates, etc., and does not have any input dependency. All the AUs in the network use the same internal structure, which is instantiated by the central AU. The central AU provides this structure to the other AUs through its output port.

The central AU is also endowed with an activation input and an activation output port. These ports are connected together with an activation links. The delayed events reactivating the central AU are used to implement time-events in FMI-2.0.

Fig. 3 shows a typical diagram resulting from the import of an FMU with 2 inputs and 4 outputs.

4 FMI-3.0 support

FMI-3.0 provides a number of new features for both Model-Exchange and Co-Simulation (Gomes et al., 2021). Some of the new features of FMI-3.0 are intrinsically supported in **Activate**. For example AU input output ports are not limited to scalars; they can be of type matrix and of different data types. But even though AUs have activation (clock) input and outputs, the semantic differences between FMI-3.0 clocks and **Activate** activations does not allow a simple mapping of FMI clocks into **Activate** activation signals.

Different types of FMI clocks require different treatments during the import process, as shown in the following sections. For each clock type in FMI-3.0, an FMU has been considered and the way it is imported in **Activate** is explained. Note that the way the clock is handled is FMI-3.0 is independent of the FMU type, *i.e.*, the FMU can be either Model-Exchange or Co-Simulation[8]. The FMU examples work identically for both FMU types.

4.1 Triggered input clocks

It may seem natural to map an FMU-3 with multiple input triggered clocks into an AU with multiple activation input ports. This however is not semantically correct because different outputs of the FMI may be associated with different clocks, *i.e.,* outputs may be differently clocked. But in an AU, all the outputs of the AU are computed on the union of all the activations activating the AU. So, a single AU can capture this aspect of the FMU behavior only if all the outputs of the FMU are associated with all of its clocks. Any other clock association requires a specific treatment.

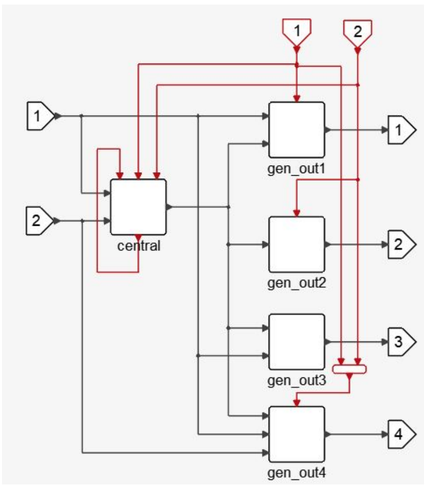
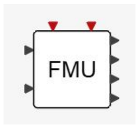


Figure 4. The diagram resulting from the import of an FMU-3 with 2 triggered clocks.

Consider the imported diagram in Fig. 3 and assume additionally that the imported FMU is an FMU-3 with two triggered input clocks where the first output is a clocked variable associated with the first clock, the second output is a clocked variable associated with the second clock and the last output associated with both. The third output is continuous-time (its `variability` attribute is `continuous`). The FMU block importing this FMU will instantiate[9] in the **Activate** model as shown below

[8]The Scheduled Execution FMU type has not been considered in this paper

[9]The block ports are automatically adjusted to the FMU specification



Its underlying diagram is shown in Fig. 4. The `gen_out3` block is declared always active.

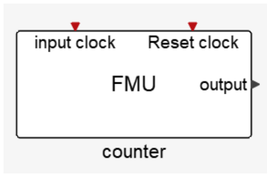
The generalization to the case where more input triggered clocks are present is straightforward.

4.1.1 Example: Clock tick counter with reset FMU

This FMU increments its output on each input clock tick.[10] The counter is reset to zero on the tick of the second input clock. The FMU has two triggered input clocks and one regular output port. The FMU model description for these ports are as follows:

```
<Clock name="input clock"   valueReference="4"
   causality="input" variability="discrete"
   intervalVariability="triggered"
   description="counter increments on ticks"/>

<Clock name="Reset clock"   valueReference="5"
   causality="input" variability="discrete"
   intervalVariability="triggered"
   description="Resets to zero on ticks"/>

<Int32 name="pre(counter)" valueReference="6"
   initial="exact"  variability="discrete"
   causality="local" description="pre(counter)"
   start="0" clocks="4 5"/>

<Int32 name="counter"      valueReference="7"
   previous="6"      initial="calculated"
   variability="discrete"   causality="local"
   description="counter internal value"
   clocks="4 5"/>

<Int32 name="output"       valueReference="8"
   variability="discrete"   causality="output"
   description="counter value"  clocks="4 5"/>
```

This FMU is imported as follows



The content of this FMU is shown in Fig. 5. Note that since there is no information in the model description of the FMU about using the time-events by the FMU, the central AU has always its first output activation port connected to its first input activation port (clock feedback).

4.2 Periodic clocks

In FMI-3.0, a time-based input clock can be defined as being periodic. The period and the offset of the clock can be constant or user-defined.

when the block parameters, in particular the FMU name and location, are provided as block parameters.

[10]The snippets of the C source code of the FMU are provided in the Appendix. FMU's presented in this paper are available upon request.

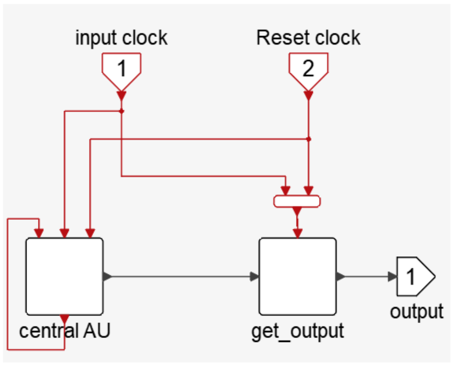


Figure 5. Importing the clock tick counter with reset FMU in **Activate**.

In the corresponding **Activate** block, such an input clock is not represented by an activation input port. Instead, the periodic clock is explicitly placed inside the diagram. Consider again the example with 2 inputs and 4 outputs and two clock inputs but now suppose the second clock is periodic with period P. The imported diagram can then be constructed as shown in Fig. 6.

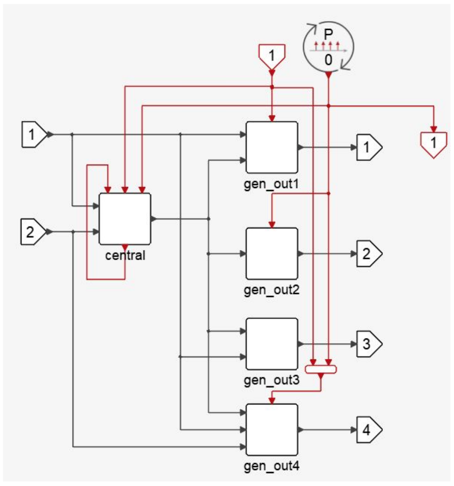
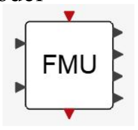


Figure 6. The second FMU clock is periodic. It does not lead to an input activation port, instead it is realized using a *SampleClock* block.

Note that the **Activate** FMU block in this case has only one activation input port. The periodic clock is placed inside the diagram and realized by a *SampleClock*.[11] In order to connect this triggered input-clock to other FMUs, an output clock port is added to the imported FMU block. This output activation port looks as follows in the **Activate** model



The generalization to more mixed triggered-periodic clock inputs is straightforward to imagine.

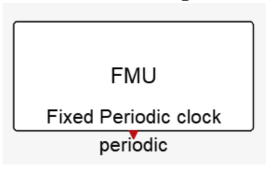
[11]All the SampleClock s in the model are synchronized by the compiler, even if they are in different diagrams.

4.2.1 Example: Periodic clock FMU

This FMU creates periodic clock ticks. The period and shift time (initial offset time) can be set by the user. The FMU does not have any regular output ports.

```
<Clock name="Fixed Periodic clock"
   valueReference="4" variability= "discrete"
   causality="input" intervalVariability="fixed"
   intervalDecimal="1.0" shiftDecimal="0.2"
   description="Fixed periodic clock "/>
```

This FMU is imported as follows



The content of this FMU is shown in Fig. 7.

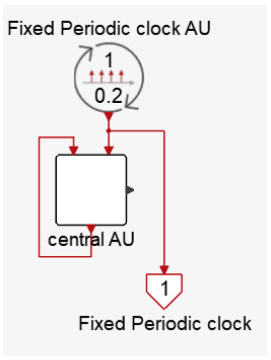


Figure 7. Importing the clock tick counter with reset FMU in **Activate**.

Although the periodic clocks of an FMU have `causality=input` attribute, these input clocks can be connected to other FMUs. For example, input periodic clock can be connected to triggered input clocks of other FMUs. In a signal based environement such as **Activate**, two input ports cannot normally be connected, due to causality incompatibility. However, with the way these FMUs are imported in **Activate**, this type of connection becomes natural.

The connection of the counter FMU and the periodic clock FMU is straightforward now and can be done in **Activate** as shown in Fig. 8.

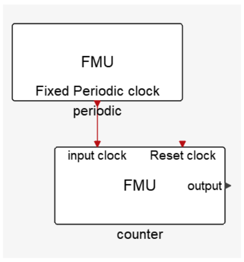
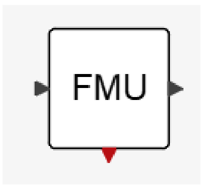


Figure 8. Connection of the periodic clock FMU to the counter FMU in the **Activate** model.

4.3 Aperiodic clocks

In FMI-3.0, a time-based input clock can also be defined as being aperiodic, i.e., `changing clock` and `countdown clock`. At each clock tick (or any event time for `countdown clock`), the time instant of the next clock tick is retrieved by the simulator (if any). This is similar to the way time events are handled in FMI-2.0, with the difference that the synchronism is ensured by the fact that the simulator (the importer) clearly activates the clock tick. Another difference between the ordinary time-event and aperiodic clocks is that unlike the time-events, clock ("input") ports can be connected to the triggered input clocks of other FMUs.

When imported, in the corresponding **Activate** block, such an input clock is not represented by an activation input port. Instead, an aperiodic clock (`changing clock` or `countdown clock`) is represented internally by an input clock and output clock in the central AU block. The output block is activated by the input clock. This is identical to the way time-events are handled in **Activate**. For example, importing an FMU with an input clock of type aperiodic results in



The content of this block is shown in Fig. 9.

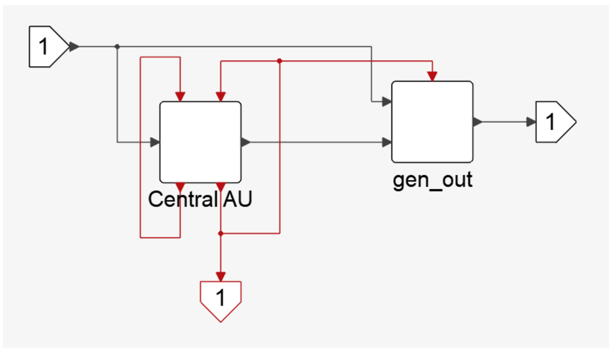


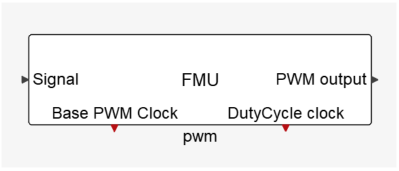
Figure 9. Importing an FMU with an aperiodic clock in **Activate**.

4.3.1 Example: PWM signal generator FMU

This FMU receives a continuous-time signal as input and creates a PWM (Pulse-Width Modulation) signal. The rate or the frequency of switching of the PWM is created by a periodic clock with `intervalVariability` attribute set to `fixed`. The duty cycle of the PWM varies as a function of the input signal, *i.e.*, at input equal to 0.0, the duty cycle is 0% and at input equal to 1.0, the duty cycle is 100%. The period and the first tick instant of the switching (defined by `intervalDecimal` and `shiftDecimal` respectively) are set by the user. In order to create the duty cycle switchings, a countdown clock is used. At every tick of the periodic clock, the next tick of the countdown clock is scheduled as a function of the input signal value. For the sake of clarity, the snippet of the C source code is given in the appendix.

```
<Clock name="Base PWM Clock" valueReference="1"
  causality="input" variability= "discrete"
  intervalVariability="fixed"
  intervalDecimal="0.1" shiftDecimal="0.0"
  description="PWM Clock" />

<Clock name="DutyCycle clock" valueReference="2"
  causality="input"      variability="discrete"
  intervalVariability="countdown"
  description="Duty cycle tick clock" />

<Float64 name="Signal"       valueReference="3"
  causality="input"    variability="continuous"
  description="input signal" start="0.1"
  clocks="1"/>

<Float64 name="PWM output"   valueReference="4"
  causality="output"    variability="discrete"
  description="PWM output"  clocks="1 2"/>
```

This FMU is imported as follows



The content of this FMU is shown in Fig. 10.

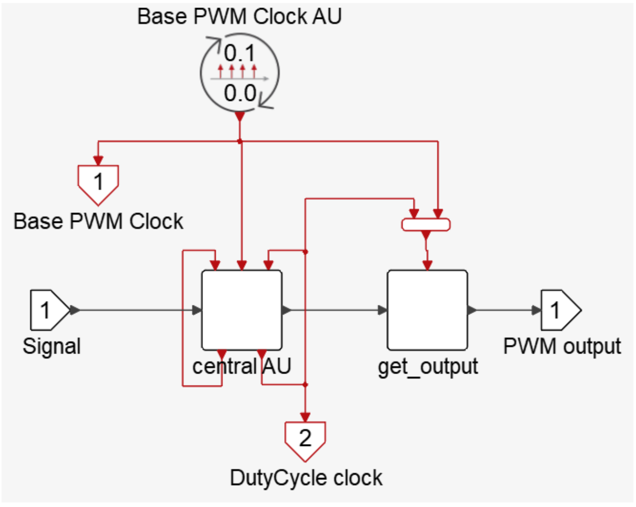
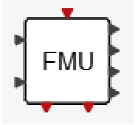


Figure 10. The content of the PWM FMU when imported in **Activate**.

4.4 Triggered output clocks

The FMI triggered output clocks correspond to output activation ports of the **Activate** FMU block. For example, if the FMI considered previously additionally has a triggered output clock, the corresponding **Activate** FMU block looks as follows in the **Activate** model



If the output clock is not synchronous with any of the input clocks, then the corresponding **Activate** event can be generated directly by the central AU. See Fig. 11.

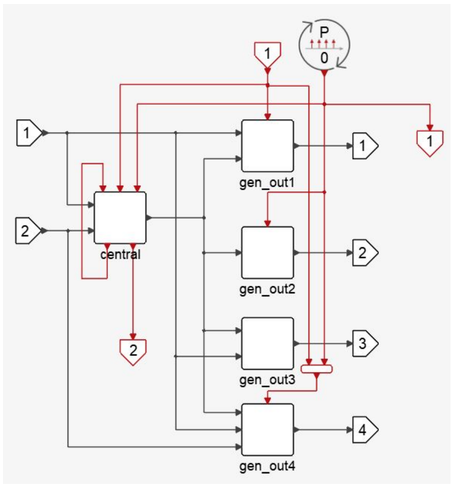


Figure 11. The diagram resulting from the import of an FMU-3 having a periodic clock and an asynchronous output clock, for example a clock triggered by an internal zero-crossing event.

On the other hand, if an output clock is dependent on (is synchronous with in the **Activate** terminology) an input clock, then it cannot be created as the output of the central AU. Only two special "blocks" *IfThenElse* and *SwitchCase* output activations are synchronous with their input activations.

Consider the same FMU again but now assume the output clock is dependent on the first input clock. The diagram can now be realized as shown in Fig. 12. In this case

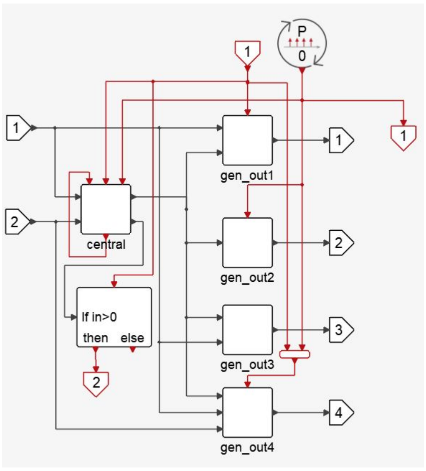


Figure 12. The diagram resulting from the import of an FMU-3 having a periodic clock and a synchronous output clock.

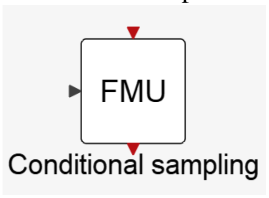
the activation of the synchronous clock is "signaled" via an additional output of the central AU. This Boolean signal has value true if the clock is to be fired. By feeding this value to an *IfThenElse* to generate (or not) the corresponding event, the event becomes synchronous with the corresponding input event (FMU clock).

4.4.1 Example: Conditional sampling FMU

This FMU has a triggered input clock, a triggered output clock, and a regular input port. The triggered output clock is activated synchronously with the input clock, only if the regular input of the FMU has a positive value. In this FMU, there is a direct dependency between the output clock and the input clock and should be handled correctly.

```
<Float64 name="Condition" valueReference="1"
 causality="input"  variability="discrete"
 description="condition for clock"  start="0" />

<Clock name="Input clock"    valueReference="2"
 causality="input"  variability="discrete"
 intervalVariability="triggered"
 description="Input clock from any source"/>

<Clock name="output clock"   valueReference="3"
 causality="output" variability="discrete"
 intervalVariability="triggered" clocks="2"
 description="Clock triggers if Condition>0"/>
```

This FMU is imported as follows



The content of this FMU is shown in Fig. 13.

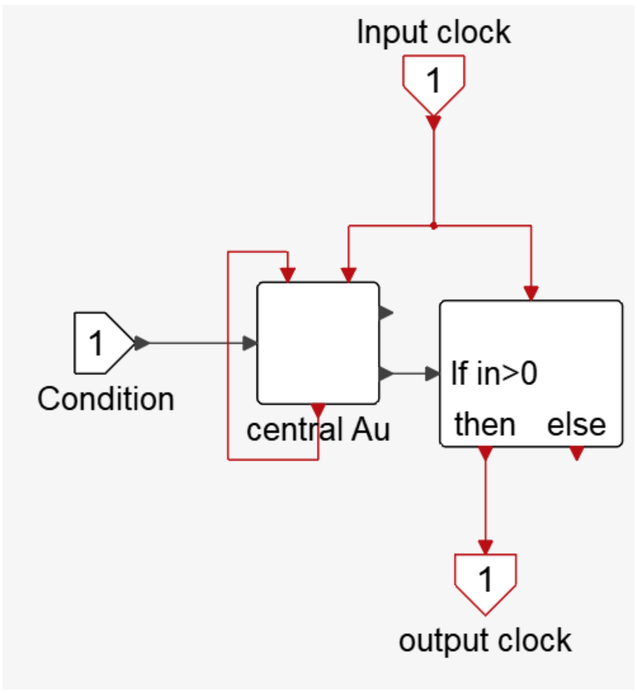
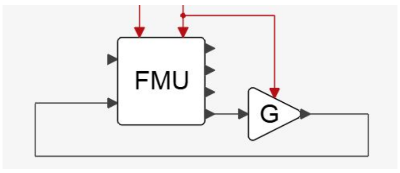


Figure 13. The content of the conditional output clock FMU when imported in **Activate** to keep the input/output synchronicity.

4.5 Clocked inputs

The FMU-3 input ports can be clocked, *i.e.,* inputs can be associated with clocks (both input or output clocks). When the FMU is imported, this information is used in the central AU to read inputs only when it can be accessed and is needed. But this information can also be used in the construction of the imported diagram so that the clock dependency is exposed to the **Activate** compiler, thus avoiding possible artificial algebraic loops.

Consider, for example, the FMU imported in the diagram in Fig. 4 and assume the corresponding **Activate** FMU block is used in the **Activate** model as follows:



If the second input is associated only with the first clock, there shouldn't be any algebraic loops in the model because there is no dependence of the forth output on the second input when the second activation input is active. There is no direct dependence in case of the first activation either. However the compiler does not see the absence of dependence of the forth output on the second input. This information is not coded in the topology of the diagram.

To include the dependence of inputs on clocks, the diagram can be modified by conditionally blocking the inputs based on corresponding clocks. In the above case, the model can be modified as shown in Fig. 14. The *SampleHold* AU is used here to block the second input except when the block is activated via the first activation port. Since it is not activated by the second activation, the model contains no algebraic loop and can be compiled.

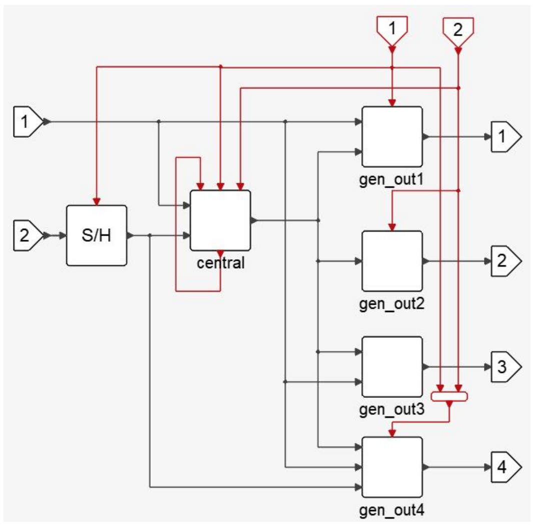


Figure 14. The *SampleHold* is used to provide the information that the second input is associated only with the first activation.

The dependence of every input on a clock can be coded in this way in the diagram.

General FMI-3.0s including multiple input and output clocks of different types can be imported by the systematic application of procedures presented above. This is done by an OML script which reads to content of the FMU model-description file (XML) and programmatically creates the required diagram.

5 Conclusion

In this paper, we showed that in general the import of a single FMU-3.0 cannot be realized by a single basic block in signal based block diagram environments such as **Scicos** and **Activate**. We examined different FMI clock types and discussed their properties and in particular their differences and similarities with the notions of activation, events and triggering in block diagram environments. We showed that there is no systematic one to one mapping of FMI clocks to block activations but the FMI clock behaviors can still be realized. The imported FMU is realized by a diagram containing multiple basic blocks depending on the type of clocks.

We presented a systematic process for creating this diagram in **Activate**. This process, which incrementally builds the imported diagram, may result in a diagram with a large number of blocks if the FMU has multiple clocks, and inputs and outputs. But the process is completely transparent to the user who sees the result as a single **Activate** block.

The import process for the user simply requires placing an FMU block, available in **Activate** palettes, inside the diagram and defining the path to the imported FMU as its parameter. The FMU block is then automatically instantiated with corresponding number of regular and activation input, output ports. It can then be used similarly to other **Activate** blocks in the construction of the **Activate** model. The corresponding internal diagram is created when the model is compiled. The diagram is not exposed to the user. It is only used internally for the compilation of the model and the construction of the compiled structure, which is used for simulation and code generation. By providing this FMU import feature, **Activate** can be used as an environment for connecting multiple FMUs (both ME and CS) to create simulation models, while respecting FMI clock semantics.

References

Altair activate, extended definitions, 2022. URL `https://2021.help.altair.com/2021/activate/extended_definitions.pdf`.

Stephen L. Campbell, Jean-Philippe Chancelier, and Ramine Nikoukhah. *Modeling and Simulation in Scilab/Scicos with ScicosLab 4.4*. Springer-Verlag New York, 2010. ISBN 0-262-16209-1.

Claudio Gomes, Masoud Najafi, Torsten Sommer, Matthias Blesken, Irina Zacharias, Oliver Kotte, Pierre R. Mai, Klaus Schuch, Karl Wernersson, Christian Bertsch, Torsten Blochwitz, and Andreas Junghanns. The fmi 3.0 standard interface for clocked and scheduled simulations. *Proceedings of the 14th International Modelica Conference.*, 2021.

INRIA. URL `http://www.scicos.org`.

Andreas Junghanns, Torsten Blochwitz, Christian Bertsch, Torsten Sommer, Karl Wernersson, Andreas Pillekeit, Irina Zacharias, Matthias Blesken, Pierre R. Mai, Klaus Schuch, Christian Schulze, Claudio Gomes, and Masoud Najafi. The fmi 3.0 standard interface for clocked and scheduled simulations. *Proceedings of the 14th International Modelica Conference.*, 2021.

FMI Website Modelica Association, 2022. URL `https://fmi-standard.org`.

Ramine Nikoukhah and Sebastien Furic. Towards a full integration of modelica models in the scicos environment. *Proceedings of the 7th International Modelica Conference.*, 2009.

Ramine Nikoukhah, Masoud Najafi, and Fady Nassif. A simulation environment for efficiently mixing signal blocks and modelica components. *Proceedings of the 12th International Modelica Conference.*, 2017.

FMI-3.0 Specification, 2022. URL `https://fmi-standard.org/docs/3.0`.

A Snippet of the C source code of the FMU in 4.1.1

```
fmi3Status fmi3UpdateDiscreteStates(
   fmi3Instance* comp,
     /* other function arguments */
   ) {
    /* some code here */
    if (comp->clki) {
          comp->counter = comp->counter+1;
          comp->clki = 0;
    }
    return fmi3OK;
}

fmi3Status fmi3SetClock(fmi3Instance comp,
    const fmi3ValueReference vr[], size_t nvr,
    const fmi3Clock value[]) {
     /* some code here */
    if (vr[nvr-1] == 4) {
        comp->clki = value[nvr-1];
        return fmi3OK;
    }
    return fmi3Error;
}
```

B Snippet of the C source code of the FMU in 4.2.1

```
fmi3Status fmi3UpdateDiscreteStates(
   fmi3Instance* comp,
     /* other function arguments */
   ) {
        if (comp->clki) {
             comp->clki = 0;
         }
        return OK;
}

fmi3Status fmi3SetClock(fmi3Instance instance,
    const fmi3ValueReference vr[], size_t nvr,
    const fmi3Clock value[]) {
     /* some code here */
    if (vr[nvr-1] == 4) {
        comp->clki = value[nvr-1];
        return fmi3OK;
    }
    return fmi3Error;
}


 fmi3Status fmi3GetIntervalDecimal(
    fmi3Instance instance,
    const fmi3ValueReference valueReferences[],
    size_t nValueReferences,
    fmi3Float64 intervals[],
    fmi3IntervalQualifier qualifiers[])
     /* some code here */
    if (vr[nvr-1]  == 4) {
        value[nvr-1] = comp->intervalDecimal;
        return fmi3OK;
    }
    return fmi3Error;
}

fmi3Status fmi3SetIntervalDecimal(
    fmi3Instance instance,
    const fmi3ValueReference valueReferences[],
    size_t nValueReferences,
    const fmi3Float64 intervals[])  {
     /* some code here */
    if (vr[nvr-1]  == 4) {
        comp->intervalDecimal = value[nvr-1];
        return fmi3OK;
    }
    return fmi3Error;
}
```

C Snippet of the C source code of the FMU in 4.3.1

```
fmi3Status fmi3UpdateDiscreteStates(
   fmi3Instance* comp,
     /* other function arguments */
   ) {
         if (comp->clkBase) {
            comp->output = 1;
            comp->clkBase = 0;
            {
                double uu;
                uu= (comp->signal >= 1.0) ?
                 1.0 : comp->signal;
                uu = (comp->signal <= 0.0) ?
                 0.0 : comp->signal;
                comp->duty=comp->period * uu;
            }
        }

        if (comp->clkCoundown) {
            comp->output = 0;
            comp->clkCoundown = 0;
        }
        return fmi3OK;
}

fmi3Status fmi3SetClock(fmi3Instance instance,
    const fmi3ValueReference vr[], size_t nvr,
    const fmi3Clock value[]) {
     /* some code here */
    switch (vr[i]) {
    case 1: comp->clkBase = value[i];
            break;
    case 2: comp->clkCoundown = value[i];
            break;
    default:
        return fmi3Error;
    }
    return fmi3OK;
}


 fmi3Status fmi3GetIntervalDecimal(
    fmi3Instance instance,
    const fmi3ValueReference valueReferences[],
    size_t nValueReferences,
    fmi3Float64 intervals[],
    fmi3IntervalQualifier qualifiers[])
     /* some code here */
    if (vr[i] == 1) {
        value[ii] = comp->period;
    }else{
      if (vr[i] == 2) {
        qualifiers[ii]=fmi3IntervalNotYetKnown;
        if (comp->duty >= 0) {
          value[ii]=comp->duty;
          if (comp->duty==comp->PreDuty)
            qualifiers[ii]=fmi3IntervalUnchanged
          else
            qualifiers[ii]=fmi3IntervalChanged;
          comp->PreDuty=comp->duty;
          comp->duty=-1.0;
        }
      }else{
         return fmi3Error;
      }
    return fmi3OK;
}

fmi3Status fmi3SetIntervalDecimal(
    fmi3Instance instance,
    const fmi3ValueReference valueReferences[],
    size_t nValueReferences,
    const fmi3Float64 intervals[])  {
     /* some code here */
    if (vr == 1) {
        comp->period = value;
        return fmi3OK;
    }
    return fmi3Error;
}
```