FMI-3.0 export for models with clock in a signal flow diagram environment

Masoud Najafi Ramine Nikoukhah

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

Abstract

The FMI-3.0 standard, recently released, introduces several promising features, such as clocks and arrays. FMI-3.0 supports various clock types, including time-based clocks, triggered input and triggered output clocks. **Altair Twin Activate** (TA), as a modeling and simulation environment, inherently supports hybrid systems combining continuous-time and discrete-time models. The discrete-time part is typically activated by events and clocks. The clock types provided by FMI-3.0 however may differ from those in TA. In the paper (Najafi and Nikoukhah 2022), we explained how different clocks defined in FMI-3.0 can be successfully imported into TA. Building upon this, our current paper aims to demonstrate how various clocks used in TA can be used in the export of a subsystem in both FMI-3.0 and FMI-2.0 formats. Specifically, we will explain the way input periodic clocks and input triggered clocks are exported.

Keywords: FMI, Synchronous clock, Signal based tool, Modelica tool

1 Introduction

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

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

TA can also be used to create Modelica diagrams (Nikoukhah and Furic 2009). The process begins with aggregating Modelica components to create a Modelica program, which is then processed by the Modelica compiler[2]. In TA, the Modelica compiler generates an FMU block that replaces the Modelica components in the original model. The resulting FMU for Modelica supports both ModelExchange or CoSimulation.

Due to this FMI-based integration of Modelica in TA, the tool offers FMU import support via a TA FMU block. More generally, this block can be used to import FMUs from other vendors (Nikoukhah, Najafi, and Nassif 2017).

With FMI-3.0 and the introduction of clocks, activation, and synchronization, FMU import and export in TA presents new challenges. Although activation signals and synchronization have been integral parts of the TA semantics from the beginning, slight semantic differences between FMI-3.0 and TA formalism prevent FMUs from being imported or exported like other native blocks in TA. This issue also existed, to a lesser extent, with FMI-2.0, as discussed in (Nikoukhah, Najafi, and Nassif 2017). The challenges and solutions for FMI-3.0 import have been presented in (Najafi and Nikoukhah 2022).

This paper addresses the difficulties and proposed solutions for providing extended support for FMI-3.0 export in TA. It begins with an overview of how TA handles activations (clocks) and discusses the differences with FMI-3.0's clock handling. Then, it presents solutions for exporting models as FMI-3.0 in TA, focusing on periodic and triggered input clock types.

1.1 Activation signals in TA

Activation signals in TA control the execution of block functions and can be explicitly manipulated, providing powerful modeling capabilities within the simulation environment. These signals are associated with red links connected to ports typically located at the top and bottom of blocks, as illustrated in Figure 1.

Activation signals are used to specify the activation times of the blocks to which they are connected. The most common usage involves the activation of blocks at a fixed frequency using signals generated by a `SampleClock` block. This block produces a series of isolated activations, known as events, which are regularly spaced in time. These events correspond to the clock ticks in FMI-3.0.

In TA, events can be explicitly manipulated: they can be conditionally subsampled, and unions and intersections of

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

[2] The Maplesim Modelica compiler is used in TA

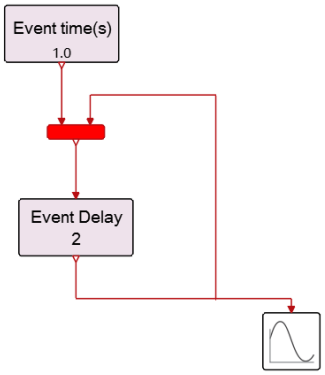


Figure 1. Event Delay model

events can be constructed. Blocks can generate delayed events, enabling operations such as event delaying. In the model shown in Figure 1, the output activation port of an event delay block is fed back to its input activation port. This setup creates a sequence of events where the time spacing between successive events corresponds to the value of the delay.

A first event generated by the `Event time(s)` block initiates the cycle, producing its first (and only) event at 1.0 seconds. The union of this activation signal and the activation signal fed back from the `EventDelay` block is generated by the red "Event Union" block, which then activates the `EventDelay` block at 1.0 seconds. At this point, the `EventDelay` block creates an event delayed by 2.0 seconds, so the next event will occur at 3.0 seconds. The simulation result of the model in Figure 1 is shown in Figure 2. Since the `EventDelay` block's activation output triggers itself, it continues to create events every two seconds for the remainder of the simulation. This combination of blocks mimics the behavior of an EventClock block, and indeed, the `EventClock` and `SampleClock` blocks are constructed with the same principle in mind.

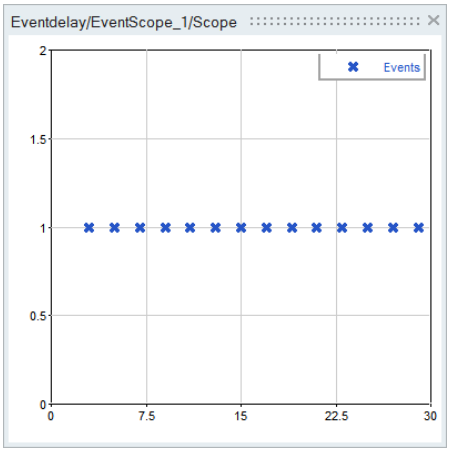


Figure 2. Event Scope results

Output events are defined by their time instants. Based on how the time instant of an event is determined, there are two different types of events in TA: predictable and unpredictable events.

Predictable or programmed events: When activated at any event time instant, a block can schedule another **event** on its activation output ports either at the current time instant or at any future time. The block specifies the event firing delay, *i.e.,* the duration after the block execution when the event should occur, for each of its output activation ports.

The block can also schedule initial output events. For instance, the block **Event time(s)** only schedules initial events and remains inactive during simulation.

The programmed events can be considered as similar to time-based clocks in FMI-3.0, in particular, `changing` and `countdown` time-based clock types.

Unpredictable or zero-crossing events: Activation signals may also be produced by blocks activated in continuous-time. If something happens inside the block, the block can program an event immediately or in the future. The `EdgeTrigger` block is a good example that produces an event based on a zero-crossing test. It generates an event when a condition occurs, such as when a variable reaches a threshold value. Figure 3 represents the simple model of a thermostat and its results (Figure 4).

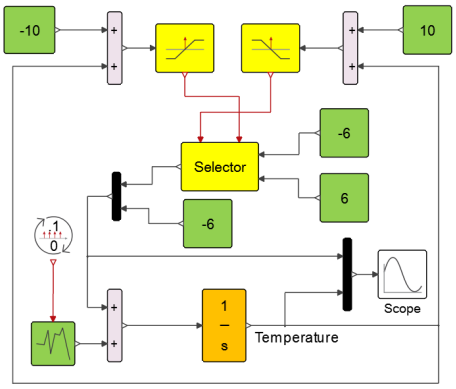


Figure 3. Simple Thermostat model

Two yellow `EdgeTrigger` blocks are used to activate the heater or the cooler when the temperature falls below -10 or rises above 10. These events trigger the `SelectInput` block, which, depending on the activation port through which it is activated, copies its first or second input (values -6 or +6) to its output. This output represents the heat flow added to a random signal and fed to an integrator, the output of which represents the temperature. The simulation results illustrate how the thermostat functionality is implemented by the zero-crossing blocks. This kinds of events are similar to the `triggered output` clock type in FMI-3.0.

The activation signals encountered so far are series

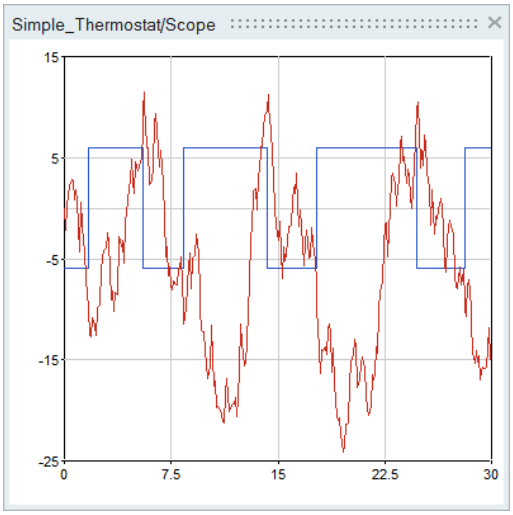


Figure 4. Results from Simple Thermostat model

of events, which are isolated activation signals in time, *i.e.*, discrete events. However, activation signals can be more general and include time intervals. The simplest activation signal of this type is the *always active* activation signal. Many basic blocks in TA palettes, such as the `SineWaveGenerator` or the `Integral` block, are "always active" by default, *i.e.*, they are (implicitly) activated by the *always active* block; they are active in continuous-time. There is no similar clock or activation type in FMI-3.0.

Another activation signal is the "initial activation". Some blocks are only activated once at the initialization phase, just before the start of the simulation. For example, the `Constant` block is declared initially active.

In the model in Figure 1, a sequence of events firing at regular intervals was created using the `EventDelay` block. This was achieved by programming an event on a regular basis. The resulting activation signal resembles the signal produced by the `SampleClock` block, but it is not of the same type. The one produced by the `SampleClock` block is of type *periodic*. The compiler recognizes this signal as periodic, which contains events firing periodically and synchronously with all other `SampleClock` blocks in the model. When a block is activated by a periodic signal, it has access to the period and offset information at compile time. This allows the block to adapt its behavior by computing specific block simulation parameters. For instance, the `SampledData` block computes the discrete-time linear system matrices corresponding to the discretization of a continuous-time linear system for the operating frequency. This frequency, which is the inverse of the sampling (activation) period, is available at compile time. `SampledData` block cannot be activated with non-periodic clock. The regular time-spaced events is similar to periodic time-based clocks in FMI-3.0, particularly fixed and constant clock types.

1.2 Synchronous vs. asynchronous activations

Activation signals are characterized by time periods or time instances. An event, for example, defines an isolated point in time specifying the time instant when the blocks receiving the event should be activated. However, the time of the event does not fully characterize the event, especially its relationship with other events. Two events may have identical times (simultaneous) but not be synchronous.

When two blocks are activated by the same event, the compiler must compute the order in which they should be activated depending on their connections and direct dependencies between inputs and outputs (port feedthrough properties) of blocks. If a block requires the value on one of its inputs to compute its output and this input is connected to the output of another block, then the latter block should be executed first. Generally, for any activation signal, the compiler computes an execution order of blocks. This order includes the blocks receiving the activation signal directly, or indirectly through inheritance.

Each "distinct" activation source has its own list of blocks and is treated independently of other activation sources. Even if two events produced by two "distinct" activation sources happen to have identical times, they are treated as independent events. At runtime, the two events are treated sequentially. Two "distinct" activation sources produce asynchronous activation signals. In general, any output activation port on a TA is considered a distinct activation source. However, there are two exceptions: basic blocks with direct event input-output dependencies are the conditional blocks `IfThenElse` and `SwitchCase`.

Consider, for example, the `IfThenElse` block, which represents conditional constructs similar to the **if-else** statement in classical programming languages such as C. The `IfThenElse` block has one activation input port and two activation output ports. Depending on the value of the signal on its regular input port, the block redirects its input activations to one of its output activation ports. In this case, the output activation signal is synchronous with the input activation signal. So, the compiler does not treat the output activation ports of the `IfThenElse` block as "distinct" activation sources. In other words, the origin of the output clocks is the same, making them synchronous. The `SwitchCase` block is the counterpart of the **switch-case** statements in classical languages. Other blocks, such as `Subsample`, built on top of these two basic blocks, also provide synchronous outputs. Note that all `SampleClock` blocks, even having periods and offset values, produce synchronous activations or events.

2 Code Generation and FMU export

Code generation is utilized to create C code from a TA superblock, capturing its dynamic behavior. The generated code serves various purposes, including creating new blocks to replace the original superblock, ensuring intellectual property protection, or exporting to other simula-

tion environments.

Two distinct code generation technologies are available in TA:

1. **Standard Code Generator**: This technology closely mirrors the behavior of the TA simulator, relying on the same libraries used by the simulator, particularly the libraries containing the simulation functions of the blocks. The generated code essentially replicates the actions performed by the simulator, resulting in performance comparable to simulation. However, the generated code is not intended for inspection or direct use and has dependencies on TA libraries. Therefore, when exported, the FMUs produced using this code generation technology contain several shared libraries.

2. **Inline Code Generator**: Unlike the standard code generator, the inline code generator does not rely on TA libraries for block simulation functions. Instead, it generates and inlines a specific code based on the types and sizes of the block input and output signals. The code is customized and highly optimized using for example by constant propagation and threshold based loop rolling. As a result, the generated code is more efficient and simpler. Additionally, all memory used by the code can be statically allocated. This code generator supports both discrete-time and continuous-time dynamics and, to some extent, multiple synchronous clocks. Various targets can be selected for code generation, such as a native TA block, a Python block, or an FMU block.

Both code generators support nested FMUs, enabling the export of Modelica models. In the standard code generator, the Modelica model is converted into an FMI-2.0 for ModelExchange, while in the inline code generator, it is converted to an FMI-2.0 for CoSimulation.

2.1 Events and clocks in FMU export

In general, the TA model may contain continuous-time and discrete-time states. Continuous-time states are, in general, always active and are invoked by the numerical solver. These states may be reinitialized or experience discontinuities at event times. Events can be triggered by either a clock or an external event. Discrete-time states are usually activated by clocks or external activations.

During the code generation process, the periods and offsets of all `SampleClocks` blocks are used to compute a base frequency which is used as parameter of a unique periodic clock. This clock drives the periodic part of the model directly or through subsampling. Thus, the final generated code is activated by one clock and possibly several external activation sources. The clock and external activations execute their own tasks at activation. These tasks may or may not have intersections or common variables.

There are several clock and event types in FMI-2.0 and FMI-3.0. Most of these event and clock types are successfuly imported in Activate (Najafi and Nikoukhah 2022). In this section, we examine the inverse problem: the way events and clocks defined in TA can be exported to FMUs.

2.2 FMI export for FMI-2.0

Synchronous clocks or external clocks are not supported in FMI-2.0. FMI-2.0 for CoSimulation does not support events. In FMI-2.0 for ModeExchange, events can be either time-event, input-event, or state-event. Among these event kinds, time-event looks appropriate to be used for export of clocks used in TA. In FMI-2.0, at each time event, the time instant of the next time-event is retrieved and the FMU is called at that time instant. This is the exact counterpart of the way events are generated in TA. The events can be either periodic or aperiodic; see, for example, the model in Figure 1.

During the FMU export, the initial event time is used at the very first time the FMU enters the Event-Mode. For the later event times, the next event time is programmed by the FMU and delivered to the FMU importer. The code snippet for handling the event inside the FMU is as follows:

```
if (fabs(comp->eventInfo.nextEventTime - currentTime)<Tolerance){
  updateOutput(x,xd,ins1,outs1,outs2);
  updateState(x,xd,ins1,outs1,outs2);
  comp->eventInfo.nextEventTimeDefined = fmi2True;
  comp->clock_tick++;
  comp->eventInfo.nextEventTime = comp->start_time+
     (comp->clock_tick) *ClockPeriod;
}
```

The `Tolerance` and `ClockPeriod` values are defined by the code generator. The problem with events in FMI-2.0 lies in event classification inside the FMU. When an event occurs, the FMU must distinguish whether it occurs due to a time-event, state-event, input-event, or if no particular event has occurred and the importer has simply pushed the FMU into event mode. To determine if the event is indeed a time event, the expected time-event time instant is compared with the current time of the FMU set by the API `fmi2SetTime`. While a good importer usually sets the current time precisely at the expected time event, the FMU should consider the general case and take into account numerical round-off errors by using an error tolerance in the comparison. This error tolerance may be problematic in many cases, which is why clocks were introduced in FMI-3.0 to eliminate uncertainties.

2.3 FMI export for FMI-3.0

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 TA, such as arrays. However, despite blocks in TA having activation (clock) inputs and outputs, the semantic differences between FMI-3.0 clocks and TA activations do not allow for a simple mapping of FMI-3.0 clocks into TA activation signals.

In FMI-3.0, besides the legacy time event already available in FMI-2.0, several clock types have been introduced. There is no exact one-to-one correspondence between TA clocks and FMI-3.0 clocks. The way FMI-3.0 clocks are imported in TA has been explained in (Najafi and Nikoukhah 2022).

2.3.1 Periodic Clock: Example of Clocked Counter

The simplest and most basic clock type in TA is the `SampleClock`, which is defined by offset and period values. Consider the sample clock shown in Figure 5, which activates a `counter`. The counter increaments its output on each clock tick and once reaches five resets to zero.

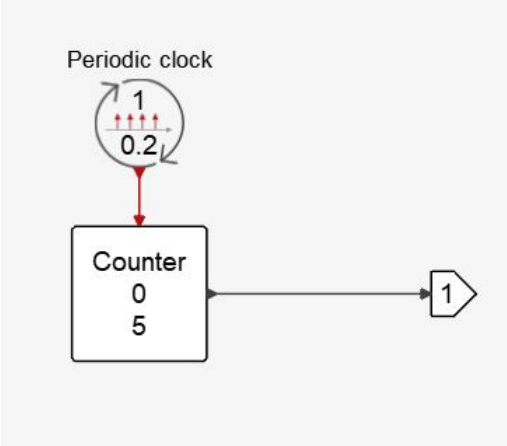


Figure 5. Clocked-counter

The sample clock is mapped to the time-based periodic clock with `intervalVariability="constant"`. The `intervalDecimal` is set to the basic period of the final clock, and the `shiftDecimal` or the clock offset is always set to zero in TA, as its value has been taken into account in computing the basic period of the clock. The clocks in TA cannot be exported with `intervalDecimal` fixed or tunable. The `clocks` attribute of the variable `Output` indicates the dependency on the clock, i.e., `"2"`.

```
<Clock name="SampleClock" causality="input"
valueReference="2"   variability="discrete"
intervalVariability="constant"
intervalDecimal="0.2" shiftDecimal="0"
description="Constant periodic input clock: 1, nevprt=1" >
</Clock>

<Int32 name="Output"  valueReference="3" variability="discrete"
clocks="2"   causality="output"   description="" >
</Int32>
```

This code snippet is generated for the above model. At the clock tick, at first, the model is evaluated, then the internal states are updated.

```
fmi3Status fmi3SetClock(fmi3Instance instance,
    const fmi3ValueReference valueReferences[],
    size_t nValueReferences,
    const fmi3Clock values[])  {
...
for (i=0;i<nValueReferences;i++)
  if (valueReferences[i]==2) {
    ...
    comp->Clk[k]=values[i];
  }
 }
...
}

fmi3Status fmi3UpdateDiscreteStates(...) {
 ...
 if (comp->Clk[0]) {
  updateOutput_clock_1 (outs1);
  updateState_clock_1 (outs1);
  ....
 }
...
}
```

The exact timing of the clock ticks is computed by the importer. At each clock time instant, the importer sets the corresponding clock and informs the FMU that the clock is enabled. So there's no place for uncertainty or error tolerance.

2.4 Triggered Input Clock: Incrementing the Counter

The next basic event source type in TA is the external activation. If a superblock is activated by an external activation, the compiler has no information about the periodicity of the events. The model part is executed when the event happens. This event type is the exact counterpart of the triggered event in FMI-3.0. Consider the model in Figure 6 where a counter is activated by an external unknown source.

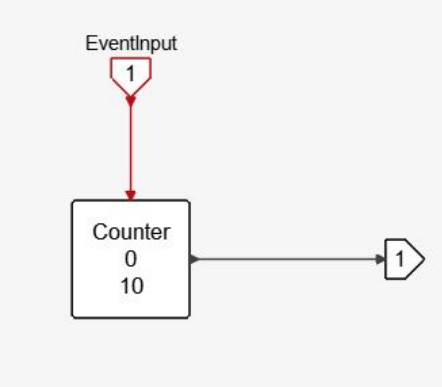


Figure 6. Triggered-counter

```
<Clock name="TriggeredClock1"  valueReference="2"
variability="discrete" intervalVariability="triggered"
causality="input"
description="External triggered input clock: 1, nevprt=1" >
</Clock>
<UInt8 name="Output"  valueReference="3"
variability="discrete"   clocks="2"
causality="output"  >
</UInt8>
```

2.5 Multiple Variable Access: Clocked Counter and Reset

In TA, several events and clocks can be used to access and update a single variable. For instance, the output of the block `Selector` in Figure 3, is updated by two input events. Another example is the `Counter with reset` block. Consider the model in Figure 7 where the

counter is incremented at activation time instants of the `SampleClock`.

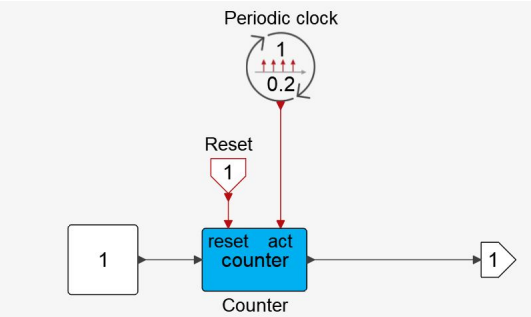


Figure 7. Counter-with-reset

Counter is reset to zero whenever the event of the external activation is fired.

reset	act	ouput
0	0	do nothing
0	1	increament by one
1	0	resent to zero
1	1	resent to zero

In this model, the counter output variable is accessed and updated by two different events.

```
<Clock name="TriggeredClock1"  valueReference="2"
variability="discrete"   intervalVariability="triggered"
causality="input"
description="External triggered input clock: 1, nevprt=1" >
</Clock>
<Clock name="SampleClock"  valueReference="3"
variability="discrete"   intervalVariability="constant"
intervalDecimal="0.2"   shiftDecimal="0"   causality="input"
description="Constant periodic input clock: 2, nevprt=2" >
</Clock>
<Float64 name="Output" valueReference="4" variability="discrete"
clocks="2 3"   causality="output"   description="" >
</Float64>
```

Note that the `clocks` attribute of the variable `Output` lists the dependency of the two clocks, i.e., `"2,3"`.

2.6 Synchronism Issue

Unlike in FMU, in TA, events can happen at the same time (simultaneous) but be asynchronous. Due to this difference, several situations should be considered to be handled. Consider, for example, a superblock having two external input events. In this case, the following table is considered to handle three possible different tasks in these situations.

Event-1	Event-2	Task
0	0	do nothing
0	1	task-1
1	0	task-2
1	1	task-3

For instance, if only Event-1 is activated, task-1 should be run. If Event-1 and Event-2 are activated synchronously, task-3 is run. This distinction between tasks is important in some situations where there is a common variable activated by two events. If no common variables are activated by both events, the execution of task-3 would have the same result as the execution of task-1 and task-2 in any order. Consider, for example, the counter in Figure 8.

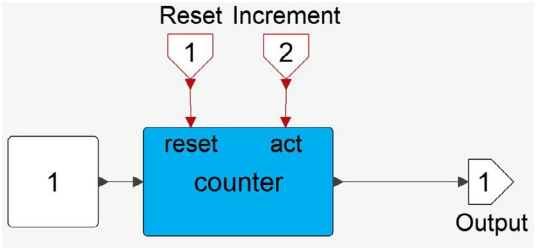


Figure 8. Counter with two-external-activations

If at a time instant both the reset and increment events are activated synchronously, in TA the output of the counter will be zero. If event ports are activated simultaneously, the order of execution is important. If the increment event input is activated after the reset event input is activated, the result will be different.

In the FMI-3.0 standard, when the FMU enters the event-mode, the importer should inform the FMU about the activated clocks by calling the API `fmi3SetClock`. With this API, the importer can enable the clocks one by one and then call the API `fmi3UpdateDiscreteStates` to execute the tasks corresponding to each clock of the FMU. The other possibility is to activate all clocks at once and then call the API `fmi3UpdateDiscreteStates`.

Actually, since there is no way in FMI-3.0 to indicate if input clocks are synchronous, *i.e.*, should tick together, the result of the simulation may be different in different importers. The only way to avoid this situation is to avoid using variables activated by different clocks. In this case, the order of execution does not matter. But this becomes a limitation for exporting a tool independent FMU.

2.7 Periodic Input Clock Connections

In TA, every clock source, dependent or independent, defines the information flow toward other input clock ports. No clock source can be connected to other clock sources. If the union of two clock sources is needed, a `Union` block can be used. For example, in Figure 9, the counter is incremented whenever each of the clock sources ticks.

In FMI-3.0, the `causality` attribute of periodic clocks is `"input"`, which should be interpreted as if the clock source is coming from the importer. This makes an open gate for arbitrary interpretation from FMU importers. For example, what should happen if two periodic clocks from two FMUs are connected together and connected to the triggered input clock of another FMU. In TA, this connection raises an error, but other tools may interpret it as the union (OR operation) or the intersection

 10.3384/ECP20791 DOI

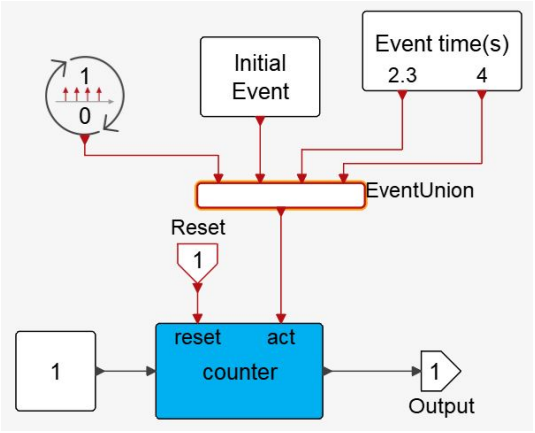


Figure 9. Example of clock unions (**EventUnion**)

(AND operation) of periodic clock ticks. This lack of definition would result in the FMU export of a model with such a connection being tool-dependent.

2.8 FMU for Cosimulation: Solver inlining

The way the clock is handled in FMI-3.0 is independent of the FMU implementation, *i.e.*, the FMU can be either Model-Exchange or Co-Simulation[3]. The FMI-3.0 FMUs work almost identically for both FMU implementations in handling clocks. The difference between the two implementations is in handling of the continuous-time dynamics. The introduction of clocks in FMI-3.0 has offered TA the opportunity of exporting the internal dynamics in a new way, *i.e.*, solver inlining.

In the FMU export for Co-Simulation, with the inline code generation, a variable-step or fixed-step solver is chosen to be used to simulate the continuous-time part of the model. Besides the classical solver linking, e.g., linking an Euler or RK4 solver, TA supports solver inlining which is a transformation method for embedding a numerical solver within the generated code. This transformation can be applied to a general model or part of it, to turn it into a purely discrete-time synchronous model with the resulting discrete-time (super) block behavior matching as closely as possible that of the original super block.

The model transformation for solver inlining, done during the model compilation process, is achieved by embedding a fixed-step numerical solver for discretizing the dynamics of the continuous-time components of the system. The exported model can be considered both as a pure discrete-time block and a Co-simulation component. The main usage of the solver inlining is for models exported by the inline code generator for embedded applications.

The basic idea behind the embedded solver is the conversion of the differential equations associated with the dynamics of blocks with internal continuous-time states to time difference equations. Difference equations are, in turn, can be implemented by discrete-time blocks running on a single base clock. This, however, does not work for variable step-solvers. The construction of the discrete-time version of the model in that case requires complex transformations. To see this difference, consider the following simple system, which can be implemented in TA by two blocks: an integrator block and a memoryless block realizing the function f.

$$y' = f(y)$$

The Euler solver uses a first order discrete approximation of this system:

$$y_{k+1} = y_k + h.f(y_k)$$

where

$$y_k = y(t_k)$$

and

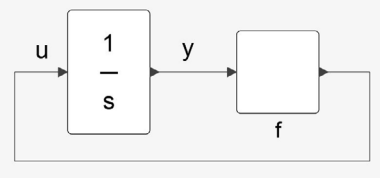
$$t_{k+1} = t_k + h$$

. The time instances when the state is updated correspond to a fixed frequency sampling of time t, with period h. The differential equation in this case is trivially translated into a difference equation. The system can be represented as a block diagram by separating the system into an integrator block and a memoryless block by noting that it can be rewritten as follows

$$y' = u$$

$$u = f(y)$$

The corresponding model can be constructed as follows.



In this case, the Euler discretization yields

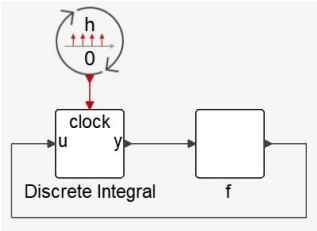
$$y_{k+1} = y_k + h.u_k$$

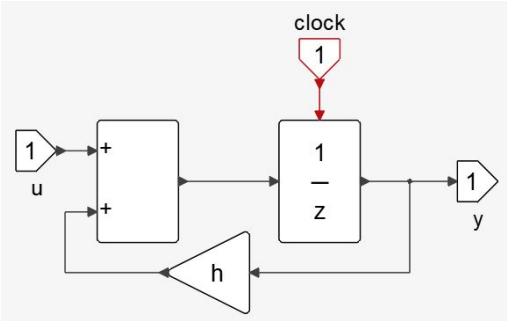
$$u_k = f(y_k)$$

So, the discrete-time version of the model is obtained by simply replacing the integrator block by a discrete block (Discrete Integral super block). The content of the Discrete Integral superblock is also shown.

The new model which is activated by a `SampleClock` block with period h is a purely discrete-time model. The original model is simply transformed by replacing the integrator with the Discrete Integral block and a `SampleClock` block. In a more general model with multiple integrator blocks, each integrator block can be replaced by its discrete-time equivalent, activated the `SampleClock` block. The stateless blocks of the model,

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





represented here by the f block, are not modified. For higher order approximations of the derivative however, the computations at each time step cannot be realized by single activations of discrete blocks. To see this, consider a fourth order Runge-Kutta (RK4) solver algorithm for the same system:

$$y_{k+1} = y_k + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

$$k_1 = h.f(y_k)$$

$$k_2 = h.f(y_k + k_1/2)$$

$$k_3 = h.f(y_k + k_2/2)$$

$$k_4 = h.f(y_k + k_3)$$

The computation of the of next discrete state y requires four evaluations of the function f with different arguments. To embed this solver in the continuous-time model to obtain a discrete-time model, the RK4 solver equations can be implemented as a more complex Discrete Integral superblock as shown in Figure 10.

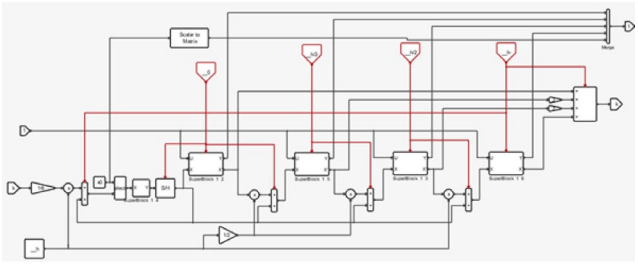


Figure 10. Discreteization of an integrator block using RK4 solver

Once the transformation is done, a C code is generated for the purely discrete model activated by the sample clock with period h.

The generated C code, which is activated by a periodic clock with a known constant period, can naturally be exported to an FMU. For FMI-2.0, a periodic regular time-event with the time interval equal to h will be used in the FMU. For FMI-3.0, the export is more natural; an input clock with the `intervalVariability="constant"` and `intervalDecimal` equal to h will be used.

The embedded solver transformation converts the continuous-time dynamics part of the model to discrete-time. In other words, the FMU has no continuous-time dynamics and the whole dynamics has been discretized and activated by a `periodic clock`. As a result, disregarding the discretization error, the FMU can be exported in the same way for both ModelExchange and CoSimulation in FMI-3.0.

3 Conclusion

The introduction of clocks in FMI-3.0 has provided the possibility of exporting more general models with continuous-time and discrete-time dynamics, particularly from TA. This paper has explored the integration of clocks within the context of **Altair Twin Activate** for FMU exports. Different clock and activation types are considered and the way they are exported to FMI-3.0 has been presented. The introduction of periodic clocks in FMI-3.0 has allowed the inlining of the numerical solver within the FMU, making it possible to achieve identical discrete dynamics in FMU export for both ModelExchange and CoSimulation.

References

Gomes, Claudio et al. (2021). "The FMI 3.0 Standard Interface for Clocked and Scheduled Simulations". In: *Proceedings of the 14th International Modelica Conference.*

INRIA (n.d.). URL: http://www.scicos.org.

Junghanns, Andreas et al. (2021). "The FMI 3.0 Standard Interface for Clocked and Scheduled Simulations." In: *Proceedings of the 14th International Modelica Conference.*

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

Najafi, Masoud and Ramine Nikoukhah (2022). "Importing FMU-3.0: challenges in proper handling of clocks". In: *Proceedings of Asian Modelica Conference 2022, Tokyo, Japan.*

Nikoukhah, Ramine and Sebastien Furic (2009). "Towards a full integration of Modelica models in the Scicos environment". In: *Proceedings of the 7th International Modelica Conference.*

Nikoukhah, Ramine, Masoud Najafi, and Fady Nassif (2017). "A Simulation Environment for Efficiently Mixing Signal Blocks and Modelica Components". In: *Proceedings of the 12th International Modelica Conference.*

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