Extended Definitions

Altair® Twin Activate™ 2023

Contents

Introduction

The chapters of this *Extended Definitions* book provide background and theoretical information on advanced features and other concepts used in **Altair® Twin Activate™**. The book is an extension of the User Guide available directly from within the product.

This book describes the features available in the *Professional Edition* of **Twin Activate**. Some of the topics (for example, chapters 13, 14 and the section 12.6) are not applicable to the *Basic Edition* of **Twin Activate**.
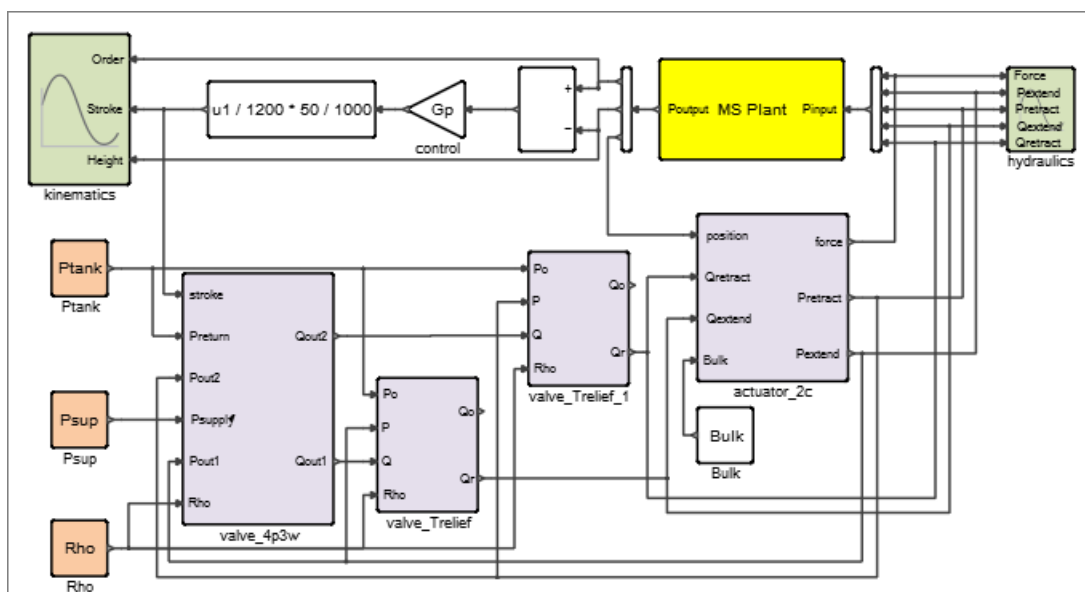
Twin Activate is a multi-discipline system simulation tool with powerful Modeling and Simulation capabilities. It includes, amongst other features,

- Handling of Continuous and discrete models through powerful hybrid simulator,
- Mixing both Signal-based as well as physical models using Modelica and Spice,
- Co-Simulation with different domains such as multibody dynamics, electromagnetics, electronic circuits,

- Functional Mock-up Interface for Import and Export,

- Library Management,

- Code generation capabilities,

- Strong scripting language integration

Note - All models used in this book can be found in the installation directory under: /tutorial_models/Extended_Book_Models/.

Chapter 1

Activation signals in Twin Activate

1.1 Introduction

Activation signals in **Twin Activate** control the execution of block functions. These signals can be explicitly manipulated providing powerful modeling capabilities not available in other simulation environments. Here we provide an introduction to the role that Activation signals play in the **Twin Activate** environment and present, through simple examples, their usage in modeling certain types of systems.

Associated with red links connected to ports usually placed on top and at the bottom of blocks, Activation signals are present in most **Twin Activate** diagrams. The most common usage is the activation of blocks at a fixed frequency with a signal generated by a **SampleClock** block. This block generates a series of isolated activations, called events, regularly spaced in time.
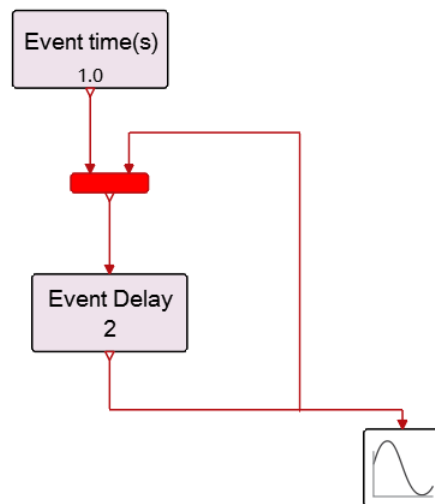
Figure 1.1: Event Delay model (Eventdelay.scm)

Events can be explicitly operated on in **Twin Activate**: they can be conditionally subsampled, and the union and intersection of events can be constructed. Blocks can generate delayed events, allowing for example the implementation of operations such as event delaying. In the EventDelay model shown

above, the output activation port of an event delay block is fed back to its input activation port, creating a sequence of events where the time spacing with successive events corresponds to the value of the delay.

A first event generated by the block **EventGenerate** initiates the cycle, which is 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 and activates the**EventDelay** block at 1.0 seconds. At this point, the **EventDelay** block will create an event that is delayed by 2 seconds, which means the next event will be generated at 3.0 seconds. Since the **EventDelay** block's Activation output activates itself, it will continue to create events every two seconds thereafter for the remainder of the simulation. This combination of blocks provides the same behavior as an EventClock block, and indeed the **EventClock** and **SampleClock** blocks are constructed in the same spirit.
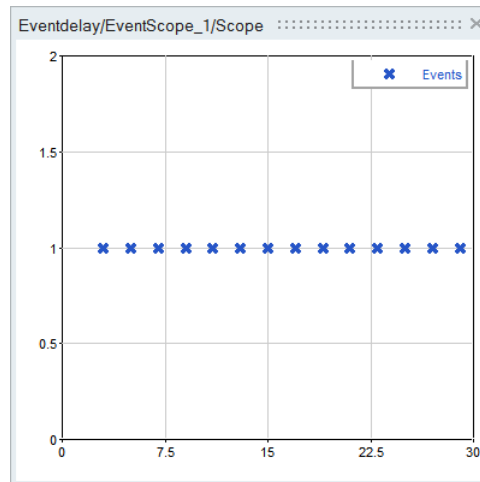
Figure 1.2: Event Scope results (Eventdelay.scm)

Note that the **SampleClock** block provides a similar behavior, but have a significant difference in how its activation output is synchronized with other blocks. More details on this can be found in Sections 1.2.5 and 1.4.2.

A random sequence of events (times between events following a random law) may be constructed using the **EventVariableDelay** block. This block delays events but the amount of delay is variable and its value is given by the value of the signal at its regular input port. The following diagram illustrates how such a random event process may be generated:
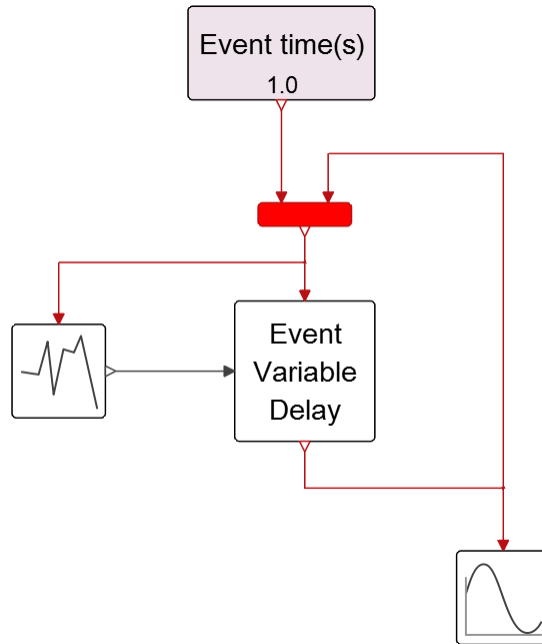
Figure 1.3: Random Event Delay model (RandEventdelay.scm)
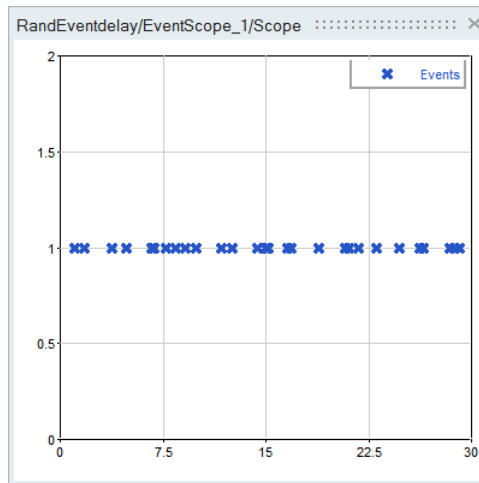
Figure 1.4: Event Scope results (Eventdelay.scm)

If the random law representing the inter-event times follows the exponential law then the random event process is a Poisson process. The Poisson process is often used in the modeling of queuing systems, for example to study the waiting time and queue size at Supermarket checkout lines.

1.1.1 Simple queue

A very simple model for a supermarket with a single checkout counter can be constructed as shown in figure 1.5

The arrivals here are modeled as shown previously but using an exponential law (see figure 1.6).
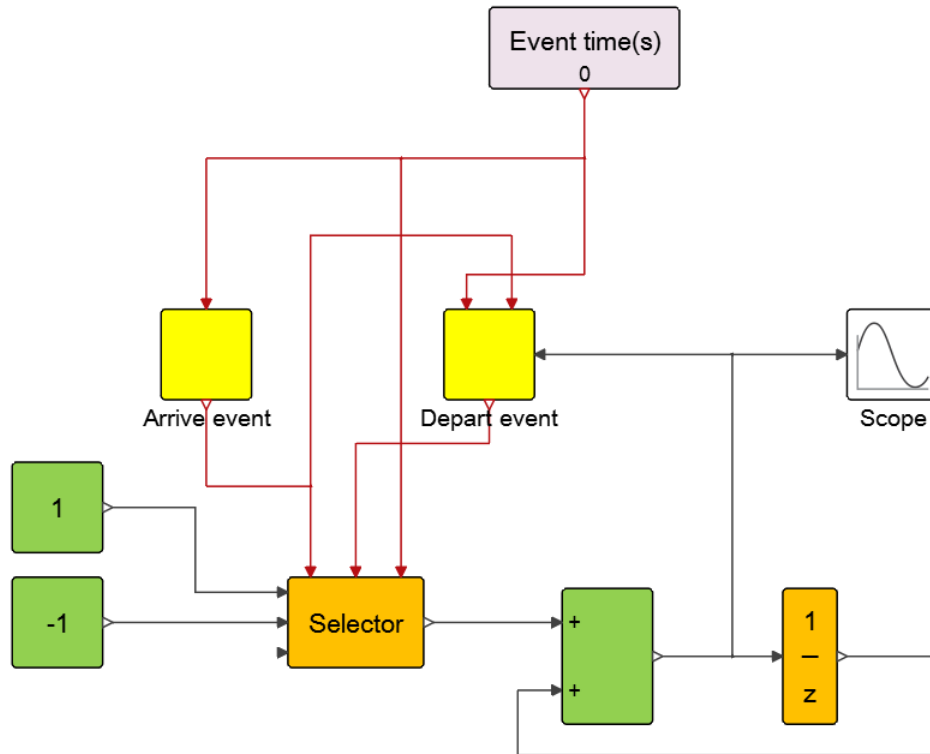
Figure 1.5: Simple Queuing model (Queue.scm)

The initial event starts the generation of arrival events.

But departures are slightly more complicated to model, even if exponential service time is considered. The reason is that unlike arrivals, departures may be halted, in particular when the queue is empty. Thus the state of the queue is required for modeling the departure process. As long as the queue is not empty, the departure process operates similarly to the arrival process: the feedback loop generates successive events. But when the queue is emptied, the feedback loop should be broken; no event should be generated in such a situation. When the queue is empty, a departure occurs only following an arrival. So the generation of the departure process depends also on the arrivals, in particular when the queue contains a single element (it has been empty and an arrival has just occurred). The departure diagram contains, in addition to the Activation input used to initialize the process (as in the case of arrivals) a second input receiving the arrival process. This Activation signal is filtered by an **IfThenElse block**. The Activation signal from the "Else" branch, corresponding to the case where the state of queue is 1, is used for generating departure events. Another **IfThenElse** block is used to filter out events if the queue is empty (figure 1.7).

At each event activation, the state of the buffer is updated displayed by the **Scope** block. It is obtained as the sum of the output of the **SelectInput** block and the previous state of the buffer (stored in the **DiscreteDelay** block. The **SelectInput** block copies one of its inputs to its output depending on the way by which it is activated. In particular if it is activated by an arrival event, the value of $+1$ is copied to the output. If it is activated by a departure event, the value of -1 is copied. The initial event copies the value 0 (an unconnected input is assumed to have value 0 in **Twin Activate**).

The simulation result shown in figure 1.8 gives the evolution of the state of the queue for the case of
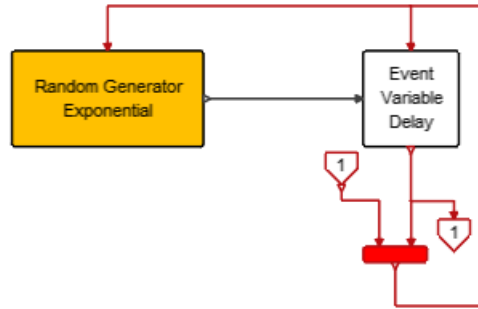
Figure 1.6: Arrival super block in Simple Queuing model

exponential service time.

It is of course simple to replace the exponential random delay with a constant delay or a more complex expression such as the sum of a constant with a random value. Indeed a realistic service time at a counter includes a time for payment which a priori does not depend on the number of purchased items and thus may be considered constant.

A useful block for modeling event sequences is the **EventDelayedChannel**. The **EventDelay** and **EventVariableDelay** blocks cannot be used to delay a train of events unless the delay is smaller than the time between two incoming events. The reason is that if an event is received by any of these block before the event programmed on its output is fired, the output event is reprogrammed and the previous event is lost. This has not been a limitation in the models considered previously but in some applications the **EventDelayedChannel** block must be used to delay Activation signals.

1.1.2   Traffic simulation

Consider modeling the traffic, as for example in figure 1.9. Each traffic light on a road may be modeled as a queuing system with each event corresponding to a vehicle.

Traffic light Super Blocks model the arrival and departure of cars (events). The regular output indicates the number of cars waiting behind the light. The road from one light to the next can be modeled using an **EventDelayedChannel** block where a travel time may be associated with each vehicle on the road. Unlike **EventDelay** blocks, **EventDelayedChannel** block contains an internal buffer that can be used to store the state of individual vehicles on the road between the two traffic lights. Note that cars on the road can travel at different speeds and pass each other. The **road** Super Block models the road assuming that the travel time of each car is an independent random number (see figure 1.10).

The numbers of vehicles, as a function of time, waiting behind the two lights are displayed by the scope and are illustrated below in figure 1.11

This paradigm may be used to model more complex queuing systems such as multiple dependent queues, finite capacity queue buffers, and complex timing models. Such models may be used to study stochastic properties of complex queuing systems and optimize control strategies via for example Monte Carlo methods.

Events may also be used in models of systems usually seen as finite state machines. Consider for example a simple thermostat-based cooling system that functions as follows: when the temperature goes above certain threshold value, it waits for a fixed amount of time and then turns on the cooling system unless in the meantime the temperature has dropped below the threshold value. For that,

Figure 1.7: Departure super block in Simple Queuing model

the block **EventVariableDelay**, and in particular its reprogramming capability may be used (see figure 1.12).

In this "Plant" diagram, when the output of the **Sum** block becomes positive, i.e., the temperature goes beyond the threshold temperature, one of the **EdgeTrigger** block generates an event which is programmed for `deltaT` unit of time later, at the output of the **EventVariableDelay** block. If the output of the **Sum** block becomes negative before this event is fired, or a "restart" event is received, the output event of the **EventVariableDelay** block is reprogrammed, in this case de-programmed since programming with a negative delay corresponds to not programming any event.

The Super Block "Cooling System" at the bottom of the diagram generates the on/off command that is actually sent to the device. The strategy is as follows: when the "on" or "restart" event is received, the cooling system is turned on for a fix period `CPer` of time. At the end of this period, if the temperature is still above the threshold temperature then the cooling system is turned off, otherwise it stays on for another `CPer` unit of time.

Note that the `CPer` period where the output is on may be repeated multiple times before the temperatures goes below the threshold value.

There are of course specialized tools for modeling these types of systems, what makes event modeling in **Twin Activate** particularly powerful is its ability to combine events, and continuous and discrete-time dynamical systems. Events may be used to create discontinuities in the dynamics of the system, and events may be generated by the dynamics. For example the thermostat model discussed previously is usually combined with a continuous-time model describing the action of the cooling system and the evolution of the temperature. A simple example is shown in figure 1.14.

The output of the integrator represents the room temperature. The simulation result is displayed in

Figure 1.8: Queue state evolution vs. time

figure 1.15.

1.1.3 Re-initialization of continuous-time state

The following example shows how an event can be used to re-initialize a continuous-time state. This example corresponds to a model of a hormone-releasing hormone pulse generator proposed in [1]. A **Twin Activate** model has been presented in [2].

The following diagram (figure 1.16) models a Super Block, named Pulse Generator, which provides both the event and the value of the jump corresponding to the event.

The outputs of this block are used to create a jump in the state of the system.

The complete model including the Pulse Generator Super Block is shown in figure 1.17.

The nonlinear dynamics of the system modeled by **MathExpression** blocks are as follows

$$\begin{aligned}
\dot{v} &= s(-v(v-c)(v-1) - k_1 g + k_2 a), \\
\dot{g} &= b(v)(k_3 a + k_4 v - k_5 g), \\
\dot{z} &= p(v) - k_6 z,
\end{aligned}$$

where

$$b(v) = b_1 - \frac{b_2}{1 + \exp(-b_3(v - b_4))}$$

and

$$p(v) = \frac{p_1}{1 + \exp(-p_2(h(v) - p_3))}$$

with $h(v) = v$ if $v > 0$ and $h(v) = 0$ otherwise. The s, a, k_i, p_i and, b_i are model parameters.

The **JumpStateSpace** block is used to store the system state and model its re-initialization. In the absence of activation on its input activation port, this block realizes a standard continuous-time linear system (in this case a simple integrator) with system input corresponding to the first input of the block. When activated through its activation port, the the state jumps to the value specified on the second input. In this case the state is $x = (v, g, z)$. The simulation results are given in figure 1.18.
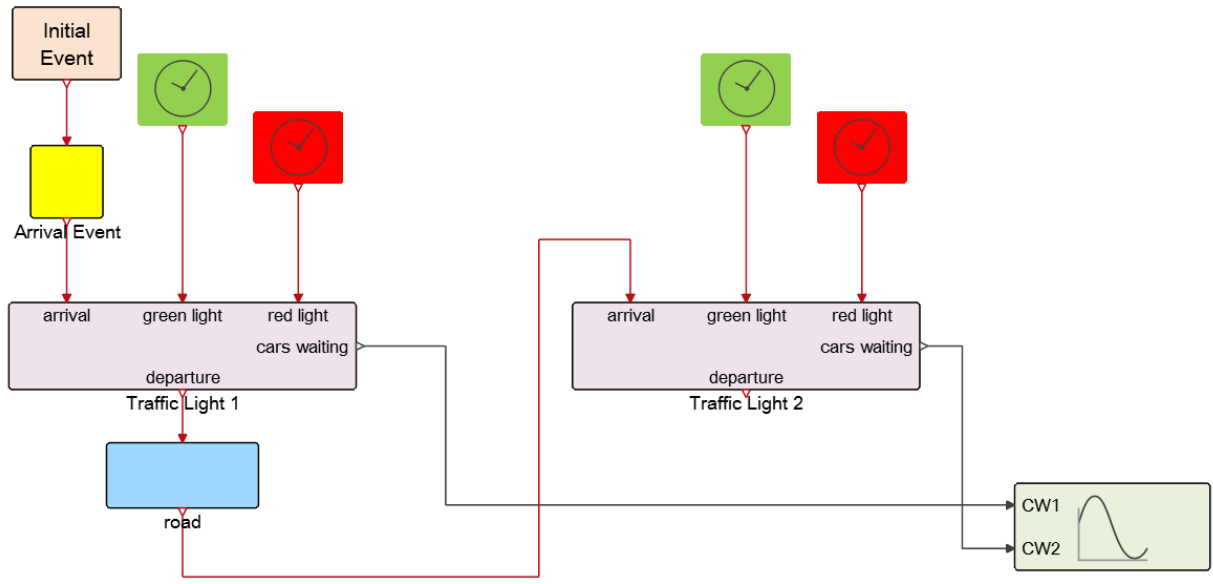
Figure 1.9: Circulation simulation model (Circul.scm)
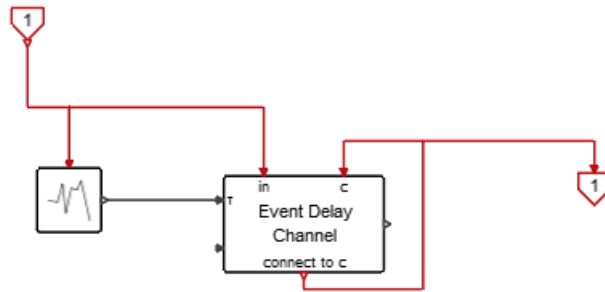
Figure 1.10: road super block in Circulation model

1.1.4 Communication delay

The ability to explicitly operate on Activation signals may also be used for studying the effects of communication delays in for example control systems. Consider a classical inverted pendulum controller with state feedback control (figure 1.19). The feedback is sampled and held over periods of 0.15 unit of time.
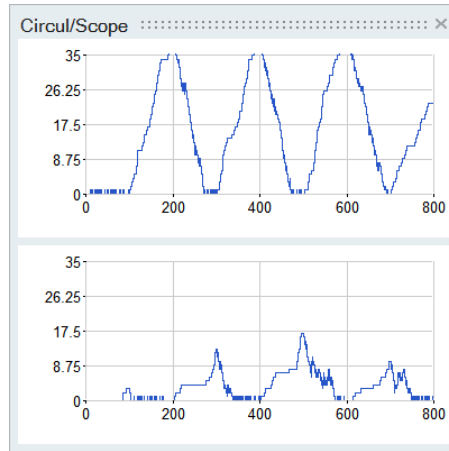
Figure 1.11: Number of vehicles behind the traffic lights
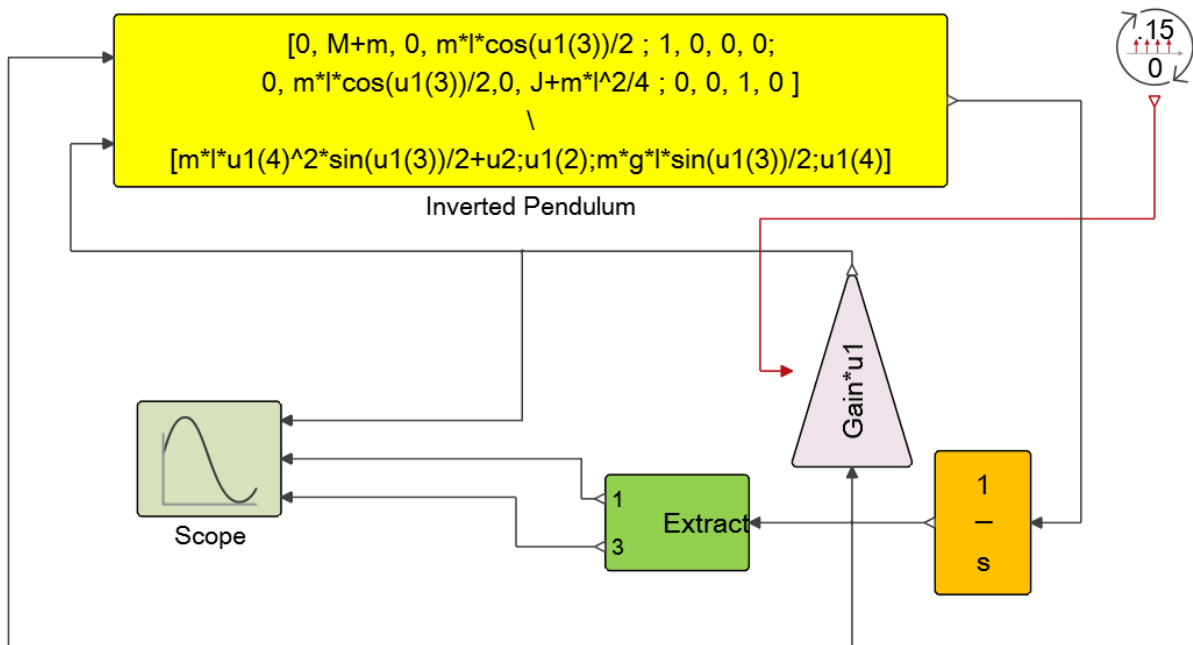
Figure 1.19: Inverse pendulum with delay model (PendInvcdelay.scm)

If the initial state is not far from the equilibrium state, then this controller is capable of stabilizing the system:
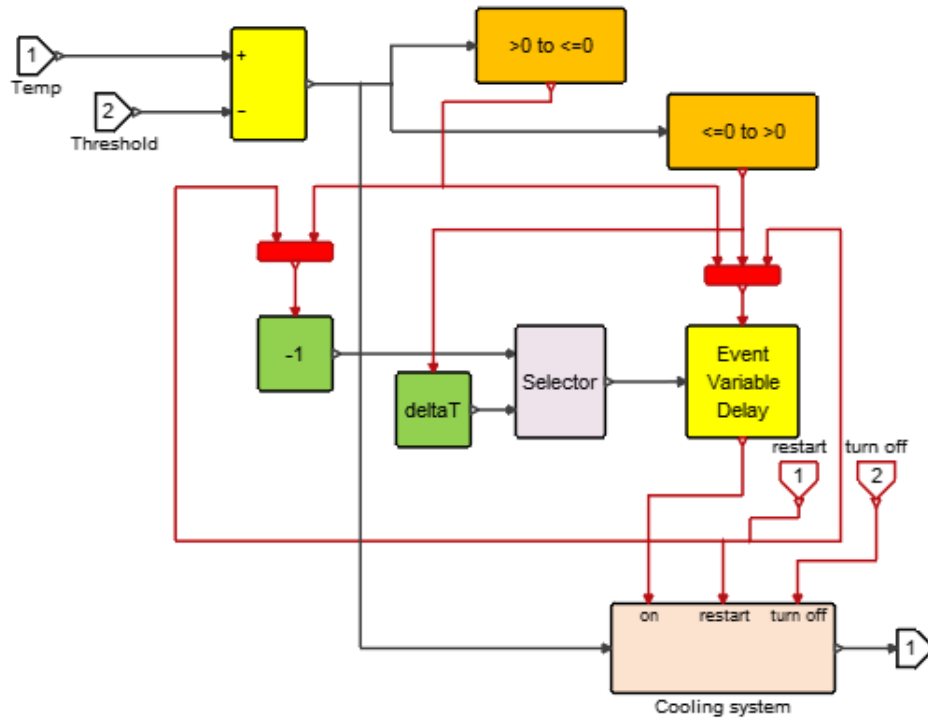
Figure 1.12: Plant super block in Cooling model (Cooling.scm)
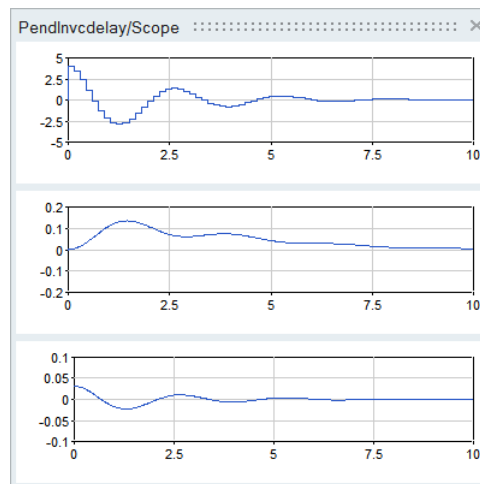
Figure 1.20: Results of Inverse pendulum with delay model (PendInvcdelay.scm)

To study the effects of communication delay in the application of the control signal, the model can be modified by applying a constant delay to the control signal. Since the control signal is a discrete signal, this can be done with a simple **EventDelay** block or a **EventVariableDelay** block with a constant input, as long as the amount of delay is less than the sampling period. Recall from Section 1.1.1, that if an event is received by this block before the event programmed on its output is fired, the output event is reprogrammed and the previous event is lost.
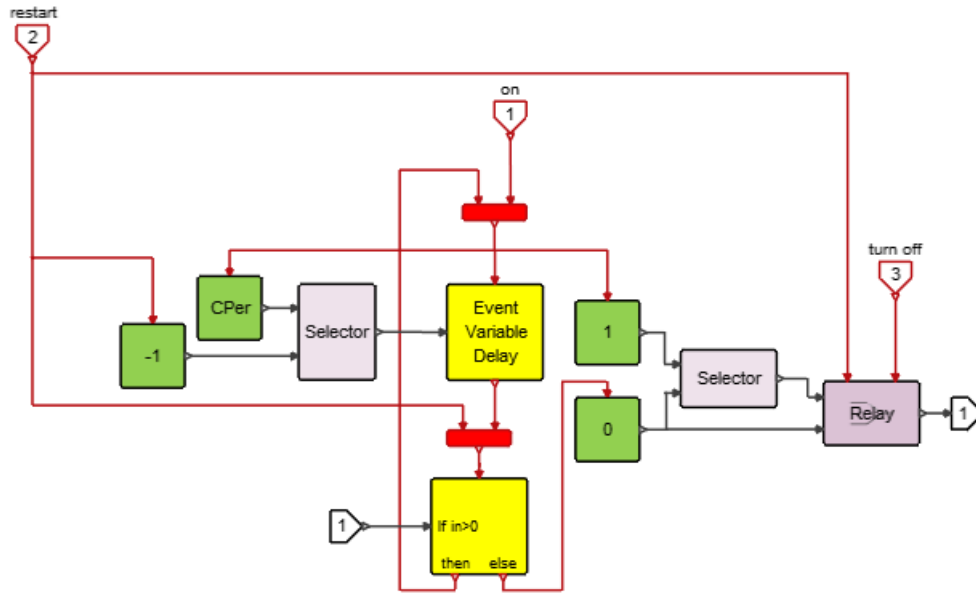
Figure 1.13: Cooling System super block (inside Plant) in Cooling model

PendInvcdelay01.scm
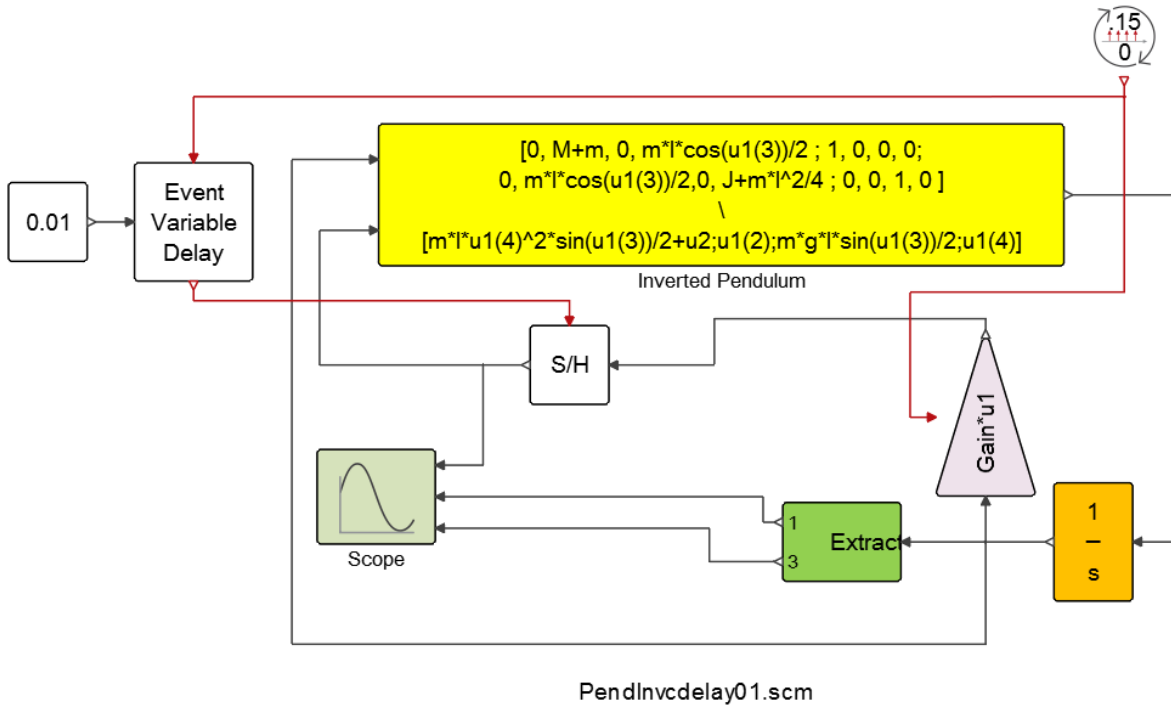
Figure 1.21: Inverse pendulum with delay model, variant 1 (PendInvcdelay01.scm)

With a delay of 0.01, the system remains stable but the convergence to the steady-state position takes more time:
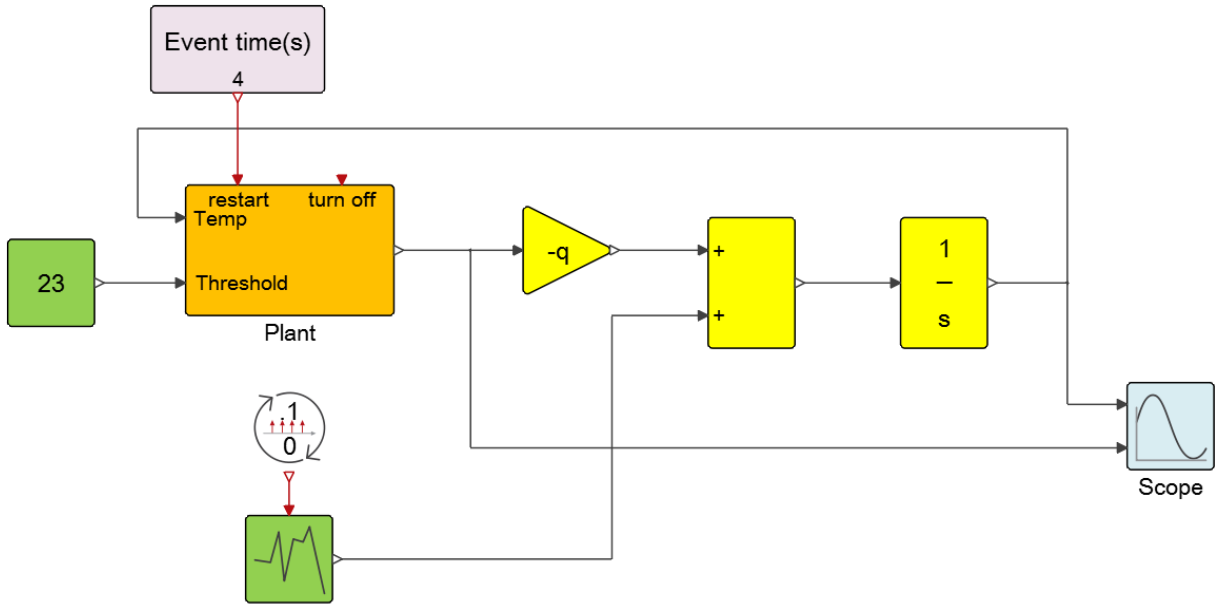
Figure 1.14: Cooling System model (Cooling.scm)
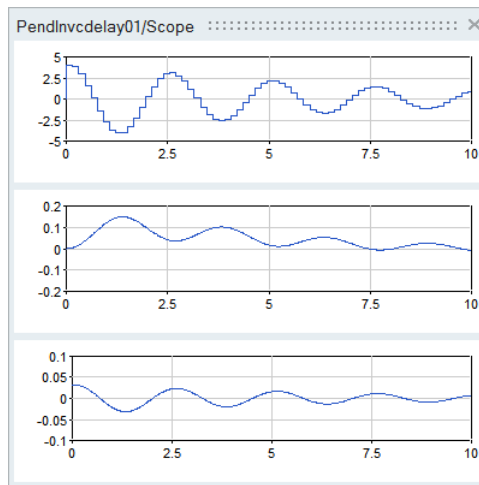
Figure 1.22: Results of Inverse pendulum with delay model, variant 1 (PendInvcdelay01.scm)

Increasing the delay to 0.02 completely destabilizes the system:
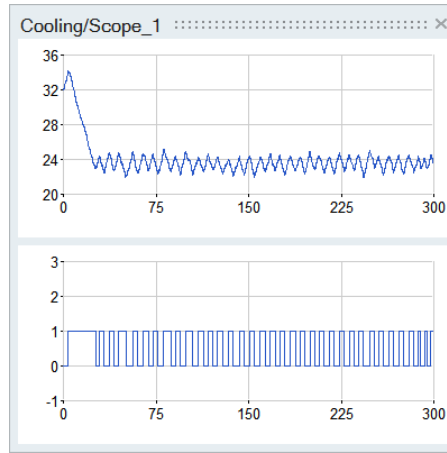
Figure 1.15: Room Temperature vs. Time
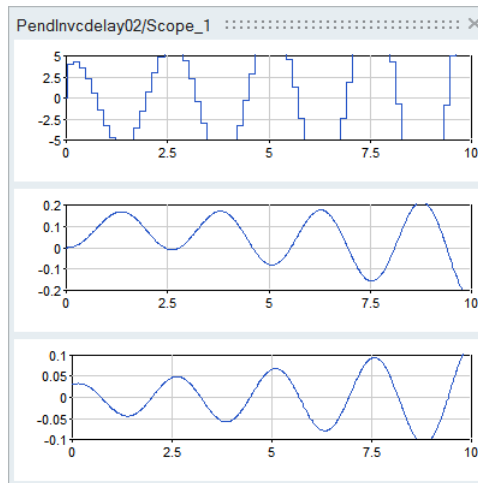
Figure 1.23: Results of Inverse pendulum with delay model, variant 2 (PendInvcdelay02.scm)

The model can be modified to study the effect of jitter (random delay) as shown in figure 1.24:
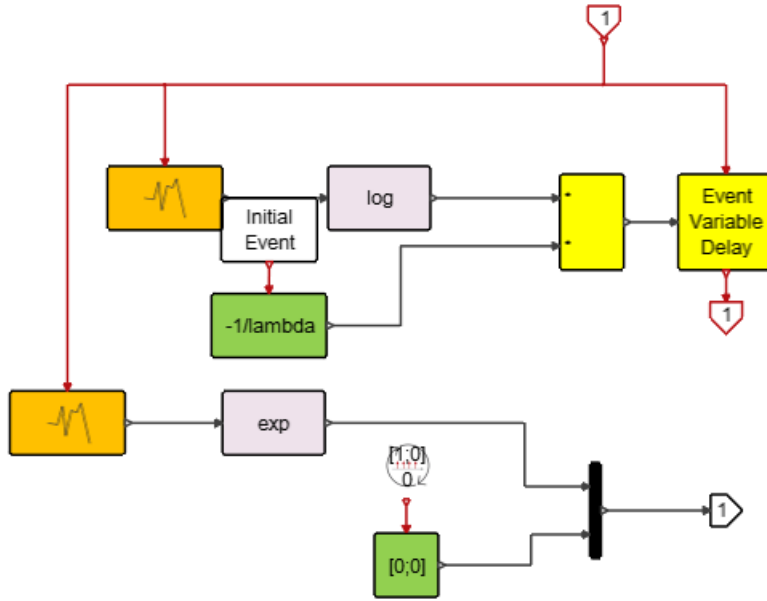
Figure 1.16: Pulse Generator superblock in lut.scm model
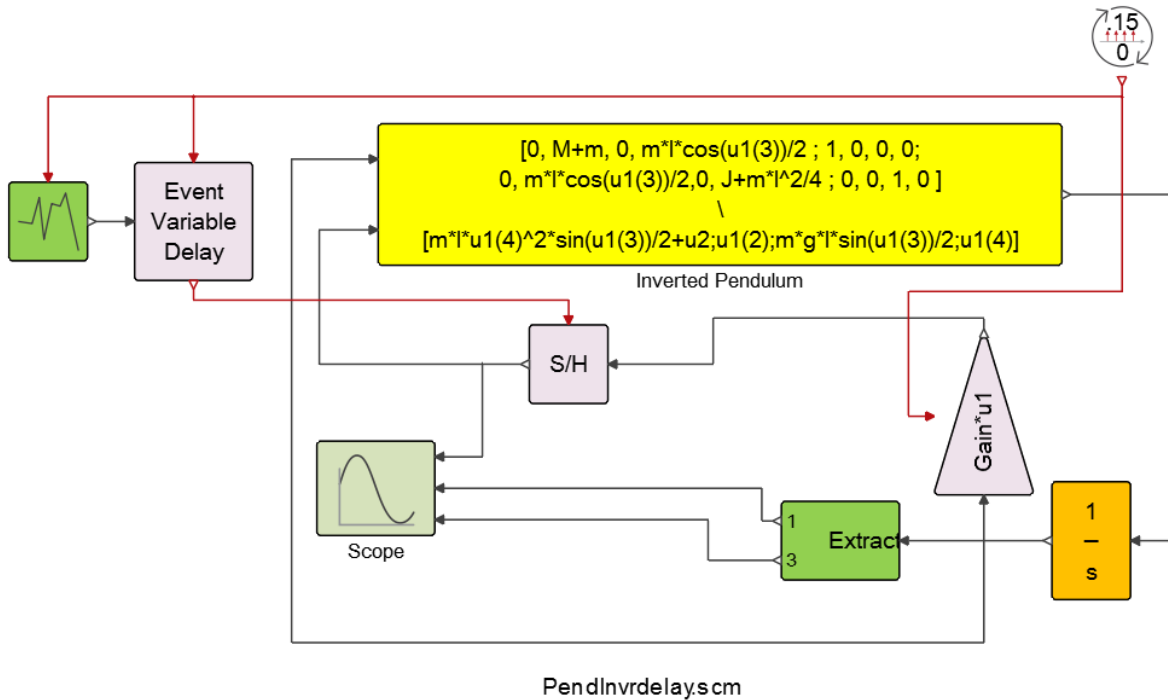
Figure 1.24: Inverse pendulum with random delay model (PendInvrdelay.scm)

This is done by feeding the **EventVariableDelay** block with a random signal instead of a constant.

If the amount of delay is larger than the sampling period, the **EventDelayedChannel** block must be used to model the behavior of the system.
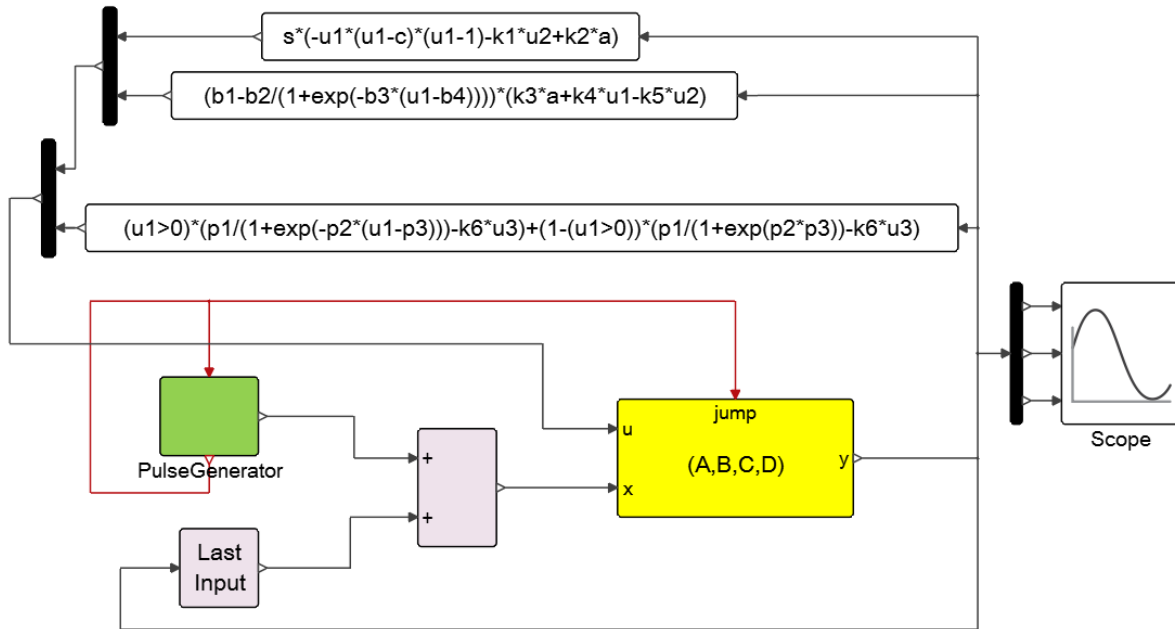
Figure 1.17: Hormone releasing model (lut.scm)

1.2 Types of activation signals

Activation signals are used to specify activation times of the blocks they are connected to. **Twin Activate** supports different types of activation signals.

1.2.1 Programmed events

Activation signals seen so far were mostly isolated timed events generated by blocks activated by other isolated timed events. These events were programmed in particular for given times in the future using the event delay blocks **EventDelay** and **EventVariableDelay**. When activated, **Twin Activate** blocks can program output events on their activation output ports. The block in particular specifies the event firing delay, i.e., how much time after the block execution the event should fire, for each of its output activation ports.

The block may also program initial output events. The block **EventGenerate** for example programs only initial events. It does nothing during simulation.

The time of a programmed event may be modified and the event may even be deprogrammed by the block before the firing time. This is done simply by programming a new output event by indicating the new firing delay. This operation overwrites the previously programmed event because only one event can be programmed at any time for any output activation port. Giving a negative delay value deprograms the event. When an event is re- or de-programmed, a warning is issued by the simulator.

1.2.2 Zero-crossing events

Activation signals may also be produced by blocks activated in continuous-time. The **EdgeTrigger** block seen previously is an example where an activation signal is produced based on a zero-crossing test.
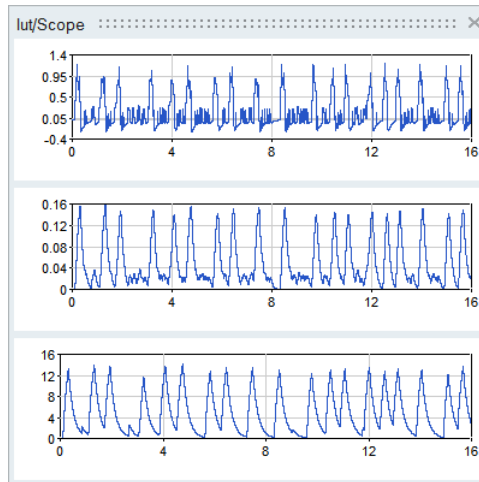
Figure 1.18: Results for the Hormone releasing model

Such events are generated when a signal crosses a given surface, in general the surface corresponding to the value 0. A number of blocks in **Twin Activate** palettes implement zero-crossing functionalities. Figure 1.25 represents the simple model of a thermostat and its results (figure 1.26).

Zero-crossing blocks are used to turn on the heater or the cooler when the temperature falls below -10 or rises above 10. The +to− and −to+ blocks generate events when the input of the block crosses zero respectively with a negative and a positive slope. The events activate the **SelectInput** block, which depending on the activation port through which it has been activated copies its first or second input (value -6 or +6) to its output. This output is the heat flow that is added to a random signal and fed to an integrator, the output of which represents the temperature. The simulation results show how the thermostat functionality is implemented by the zero-crossing blocks.

1.2.3   Continuous-time activation signal

The activation signals so far encountered are series of events, which are isolated activation signals in time. But activation signals are more general and include time intervals. The simplest activation signal of this type is the *always active* activation signal. The **SampleClock** block with period zero generates the always active activation signal; blocks that should be always active may be activated using this signal.

Twin Activate also provides a facility to declare a block "always active", without having to draw an explicit link from the **SampleClock** block. Such a link is added by the compiler to the virtual activation port numbered 0 of the block when the block is declared "always active". Many basic blocks in **Twin Activate** palettes, such as the **SineWaveGenerator** or the **Integral** block, are "always active".

1.2.4   Initial-time activation signal

Some blocks are only activated at the start of the simulation. This is the case for example of the **Constant** block with default parameters. Other blocks may need to perform specific tasks at initial time. The "initial activation" signal is used to activate blocks at the start of the simulation. This signal may be generated using the **InitialEvent** block (which contains a **SampleClock** block with period set to −1). As for the "always active" signal, the "initial activation" may be declared as a block property. The **Constant**
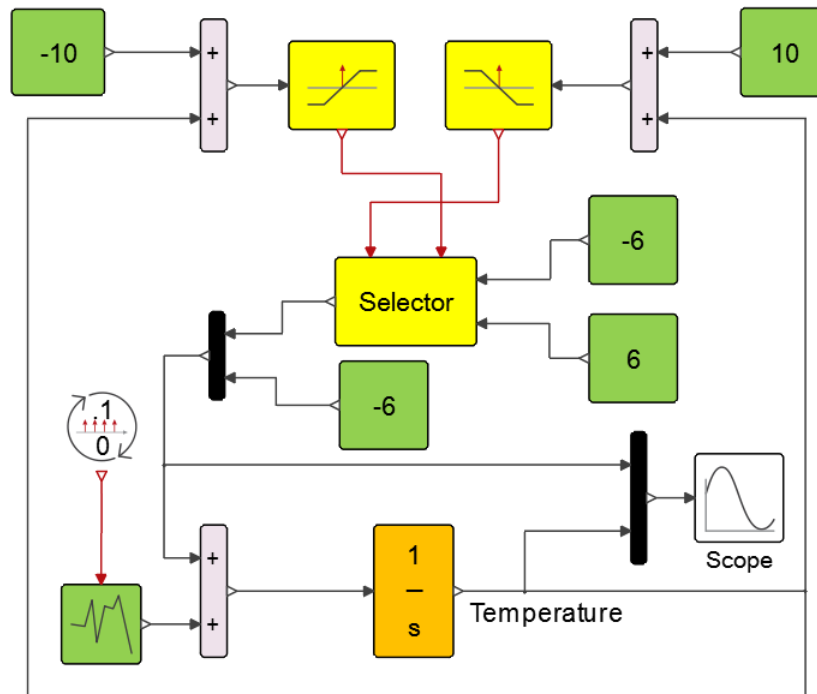
Figure 1.25: Simple Thermostat model (Simple_Thermostat.scm)

block is for example declared initially active. Note that an always active block is also initially active.

1.2.5 Periodic activation signals

In the first model considered in Section 1.1, a train of events firing at regular intervals was created using the **EventDelay** block. This was done in particular by programming an event on a regular basis. The Activation signal produced 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 knows that this signal is periodic contrary to the signal produced by the **EventDelay** block, which happens to contain events firing periodically thanks to the way the model is constructed.

An Activation signal is said to be periodic only if it is of type periodic.

Strictly speaking a **SampleClock** block is not a regular **Twin Activate** block. Not only does it produce Activation signals that are periodic but also synchronous (see Section 1.4).

The **Twin Activate** compiler has a special treatment for periodic activation signals. Some **Twin Activate** blocks operate only on periodic activation signals. When a block is activated by a periodic signal, it has access to the period and offset information at compile time. Thanks to this information, the block can adapt its behavior by computing specific block simulation parameters.The **SampledData** block for example 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. This block cannot function if it is activated by a non-periodic activation signal.

29 Extended Definitions   Altair Twin Activate 2023
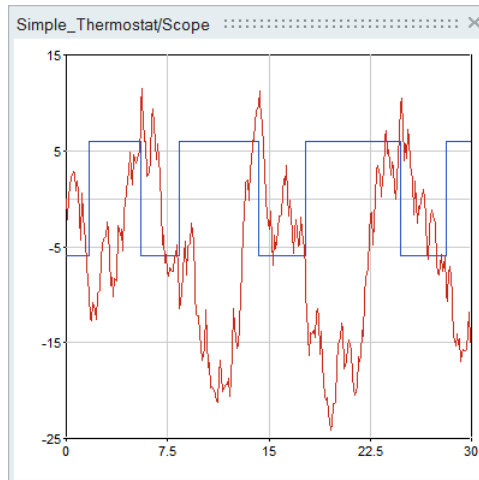
Figure 1.26: Results from Simple Thermostat model (Simple_Thermostat.scm)

The only blocks producing periodic activation signals are the **SampleClock** and the **ReSampleClock** blocks.

1.3 Activation inheritance

Specifying explicitly the activation signals of the blocks in the model is useful for specifying the model behavior unambiguously. But doing it systematically could make the model labyrinthian. In some cases the explicit graphical presence of activation links is not necessary for the comprehension of the model behavior. The ability to declare blocks initially active or always active is one mechanism to simplify the model by removing some activation links. The other mechanism is inheritance.

The inheritance mechanism is an editor facility that can be used to reduce significantly the number of activation links in models. Most basic blocks in **Twin Activate** may be both explicitly activated or activated by inheritance. Whether the block works by inheritance or external activation is specified via the "External activation" checkbox in the block parameter GUI. When the box is checked, the input activation port is added; unchecked, the block inherits its activation. For example, see the Scope block.

A block inherits its activation if it has no activation input ports and it is not declared always active. A block that inherits, either receives the union of the activations of all the blocks that provide its inputs on its only (invisible) input activation port, or receives the activations associated with each of its input on a separate (invisible) input activation port. The former is the default behavior. Indeed most blocks don't behave differently depending on the port through which they have been activated.

Consider the following model described in figure 1.27.

The yellow colored blocks inherit their activations. The yellow **Power** block has one input so its inheritance yields the exact behavior as that of the orange colored **Power** block. The yellow **Sum** block has two inputs and has default inheritance property so it is activated by the union of the activations generated by the two **SampleClock** blocks, as in the case of the orange colored **Sum** block. This is the proper inheritance property for the **Sum** block since no matter which input of the block is modified, the same summation operation must be performed.

The **SelectInput** block on the other hand has naturally multiple activation input ports. The block copies
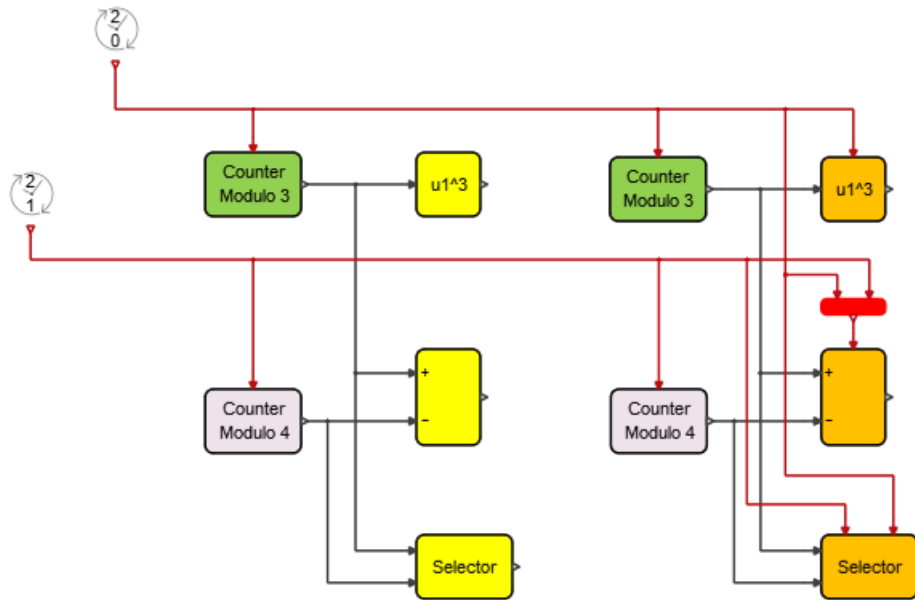
Figure 1.27: Model illustrating inheritance

one of its inputs to its output depending on the port by which it has been activated. So the inheritance property of the **SelectInput** block is necessarily different from that of the **Sum** block. Its behavior is that of the orange colored **SelectInput** block.

1.4 Synchronous vs. asynchronous activations

Activation signals are characterized by time periods and instances. An event for example defines an isolated point in time specifying the time instant when the block(s) receiving the event should be activated. The time of the event however does not completely characterize the event, and in particular its relationship with other events. In particular, two events may have identical times but not be *synchronous*. The distinction between synchronicity and simultaneity is a subtle but important point to understand since it plays a fundamental role in the semantic specification of the graphical language.

When two blocks are activated by the same event the compiler must compute the order in which they must be activated depending on the way they are connected to each other. In particular if one of the blocks requires the value on one of its inputs for computing its output and this input is connected to the output of other block, then the latter block should be executed first. In general, for any activation signal the compiler computes a table to specify the list of the blocks to be executed and the order in which they have to be executed. The list includes the blocks receiving the activation signal directly or indirectly through inheritance or the **ExtractActivation** block.

Each "distinct" activation source then 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 independent event. At run time, the two events are treated sequentially. Two "distinct" activation sources produce *asynchronous* activation signals. The notion of "distinct" activation sources however needs clarification. In general any output activation port on a **Twin Activate** is considered a distinct activation source. There are however exceptions:

1.4.1 Conditional blocks

Two **Twin Activate** blocks play a very special role in the construction of models. Strictly speaking conditional blocks are not **Twin Activate** blocks but are graphically represented as such to facilitate their usage. To make an analogy with classical programming languages such as C where blocks represent function calls, conditional blocks represent the conditional constructs: **if** and **switch case** statements. 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 cannot treat the output activation ports of the **IfThenElse** block as "distinct" activation sources.

The **SwitchCase** block is the other conditional block in **Twin Activate**. This block may have more than two output activation ports and the input activation signal is redirected to one of them depending on the value of the regular input of the block at the time of execution.

1.4.2 Sample and Resample Clock blocks

The **SampleClock** and **ReSampleClock** blocks are not regular blocks either. These "blocks" produce a train of events with specified phase and period, just as the **EventClock** block does. The **EventClock** is essentially a Super Block containing an **EventDelay** with feedback. But unlike the **EventClock**, the activation signals produced by multiple **SampleClock** and **ReSampleClock** blocks may be synchronous. The compiler replaces the synchronous **SampleClock** and **ReSampleClock** blocks with a unique **EventClock** and subsampling mechanisms using conditional blocks to produce synchronous activation signals corresponding to the outputs of all the **SampleClock** and **ReSampleClock** blocks.

SampleClock and **ReSampleClock** blocks are considered synchronous throughout the model unless the model contains **SyncTag** blocks. If a diagram contains a **SyncTag** block, then the **SampleClock** and **ReSampleClock** blocks inside the diagram are not considered synchronous with **SampleClock** and **ReSampleClock** blocks outside the diagram (but they remain synchronous among themselves in the absence of other **SyncTag** blocks).

Note that conditional and Sample blocks may be hidden inside Super Blocks. In particular the **FrequencyDivider** and **EdgeTrigger** blocks in **Twin Activate** palettes generate synchronous activation signals because they include conditional blocks.

Bibliography

[1] D. Brown et al., Modelling the Luteinizing Hormone-Releasing Hormone Pulse Generator. Neuroscience, Vol 63, no 3, 1994.

[2] S.L. Campbell, J.Ph. Chancelier and R. Nikoukhah, Modeling and Simulation in Scilab/Scicos with ScicosLab 4.4. Springer, 2010.

Chapter 2

Twin Activate Matrix Expression Block

2.1 Introduction

The Matrix Expression block is a **Twin Activate** block with multiple inputs and a single output. The input/output function is expressed in terms of a matrix formula called *matrix expression*. This document specifies the syntax and the semantics of the expressions that can be used in this block as matrix expressions.
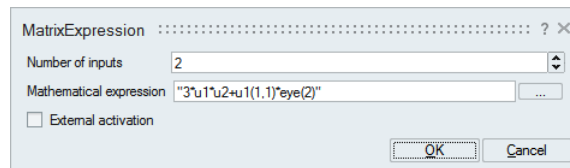
Figure 2.1: Matrix Expression Block

The inputs of the block are always denoted `u1`, `u2`,...,`u<n>`, where `n` is the number of input ports. These variable names are keywords that take precedence in the matrix expression over any parameter or function with the same name defined in the context of the **Twin Activate** diagram.

In addition to inputs, matrix expressions may include parameters. Unlike inputs, the parameters remain constant during simulation. Only parameters defined in the context of the **Twin Activate** diagram can be used in the definition of the matrix expression. Inputs and parameters are matrices of arbitrary dimensions. Only real double matrices are currently supported.

A large set of matrix operators and functions can be used in a matrix expression. For the most part these matrix operators and functions follow **OML**'s syntax and semantics. For example the + and − operators are used to add and subtract matrices of the same size but they accept also operations between matrices and scalars. The `*` operator denotes the matrix multiplication but also the product of a scalar and a matrix. The \ and / operators denote left and right matrix divisions. If the multiplication and division operators are preceded with a dot then the operation is performed element-wise. For example `A.*B` denotes the operation that consists in multiplying each entry of the `A` matrix with the corresponding entry of the `B` matrix and producing a matrix of the same size as `A` and `B`.

OML operators for matrix constructions and extractions are also generally supported. There are some differences, for example the arguments of the **OML** column concatenation operator may be separated by

commas or spaces whereas only comma-separated arguments are supported in the matrix expression block (for example in the matrix expression block, the valid **OML** concatenation operation `[A B]` must be expressed as `[A,B]`, which is also a valid **OML** expression). In some cases the matrix expression block is less restrictive than **OML**.

Most mathematical functions such as `sin`, `cos`, `exp`, accept matrix arguments in which case the function is applied to each entry of the matrix.

The list of these operators and supported functions are given in Sections 2.6 and 2.7.1. In addition to supported functions, any **OML** function available in the context of the diagram (both basic **OML** functions and user defined functions) can be used as long as their arguments do not depend on the values of the block inputs. These functions are evaluated once at compilation time. String arguments of such functions must be expressed with double quotes and not single quotes as in **OML**.

2.2 Examples

Matrix Expressions are similar to **OML** expressions with some restrictions and some extensions.

2.2.1 Basic operators and functions

Matrix addition and multiplication   The expression

`u1'*u2`

is the matrix product of `u1` transpose and `u2`. This is a valid expression if `u1` and `u2` have the same number of rows, or if one is one-by-one, i.e., is a **scalar**.

The expression

`A*u1+u2`

when the parameter `A` is defined in the context of the **Twin Activate** diagram containing the block is valid if the dimensions of matrices `A`, `u1` and `u2` are compatible. Note that the product `A*u1` is considered valid if the number of columns of `A` equals the number of rows of `u1`, or if `A` or `u1` is scalar. Similarly, the sum `A*u1+u2` is valid if either both terms of the sum have identical dimensions, or one is scalar.

Element-wise operators and functions   If matrix inputs `u1` and `u2` have equal size, the expression

`(u1.*u2)./(u1.^2+u2.^2)`

returns a matrix of the same size. The (i, j) entry of the result is equal to the product of the (i, j) entries of `u1` and `u2` divided by the sum of their squares.

The expression

`u1>3`

generates a matrix having the size of `u1` and including ones and zeros depending on whether the corresponding entry in `u1` is larger than 3 or not.

The expression

`sin(u1)`

returns a matrix having the size of `u1` with entries equal to the sine of the corresponding entry in `u1`.

If expression Conditional statements in **OML** are not expressions and are not supported. However matrix expression supports expressional if statements, and in particular their extension to the matrix case, where the conditions are applied elementwise. The keywords `.if`, `.then` and `.else` are used in the expressional if statements. The condition and the values returned in the `.then` and `.else` branches are normally matrices of identical dimensions however a scalar can also be used in which case it is (conceptually) expanded to a matrix.

For example the expression:

```
.if u1<0 .then u1 .else 0 .end
```

returns a matrix equal to `u1` in which the negative entries are set to zero.

Switch expression Similarly the 'switch' clause is expressional and is extended to the matrix case. For example the expression:

```
.switch u1
  .case a, u1
  .case b, -u1
  .otherwise 0
.end
```

implements an element-wise switch expression.

2.2.2 Matrix construction, extraction and assignment

Matrix construction Commas (for column concatenation) and semi-colons (for row concatenation) are used to construct matrices from other matrices (or scalars which are just one-by-one matrices). This is similar to constructing matrices in **OML**. For example:

```
[u1,1;u2]
```

is valid provided `u1` has one row and that the number of columns of `u2` exceeds that of `u1` by one.

Extracting a sub-matrix A submatrix of a matrix can be specified by indicating the row and column indices to be extracted. For example

```
u1([1,3],2)
```

returns the two-by-one matrix `[u1(1,2);u1(3,2)]`.

The extraction operator can also be used for repeating rows and columns or changing their positions. For example if `u1` is a two-by-two matrix

```
u1([1,1,1,2,2,2],[1,2])
```

returns a six-by-two matrix where each column is repeated three times and the rows are switched.

Single index extraction is also possible:

```
u1(2:n)
```

The matrix `u1` is seen as a vector containing the entries of the matrix aligned column-wise so the result is a column vector, unless `u1` is a row vector in which case the output is a row vector.

Special operators The use of the keyword `end` and the symbol `':'` simplify matrix expressions by allowing to refer to the last element or all the elements of a row or a column of a matrix without having to explicitly compute its size. For example

```
[u1(1,end);u2(end-1,:)]
```

returns the concatenation of the first row of `u1` with the one-before-the-last row of `u2`.

The `'end'` indicates the last row or column element depending on whether it is used in the first or second argument of the extraction operation, and `':'` denotes all the elements of the row or column. In the case of single index extraction, `'end'` indicates the last element (last row element of the last column) and `':'` all the elements of the matrix.

The symbol `'$'` may also be used in place of `'end'`.

Matrix assignment and row column deletion The `'='` sign is not used in the matrix expression since unlike in a **OML** script, an expression does not define intermediary variables. The matrix assignment operator `':='` however allows modifying entries of an existing matrix similar to the **OML** instruction `A(exp1,exp2)=exp3`. For example the following expression

```
u1(:,1):=3
```

returns a matrix identical to `u1` with all the elements of its first column set to 3. The right hand side of the `':='` operator should have the same size as the submatrix designated by the left hand side, or be a scalar.

Row and columns can be deleted by assigning an empty matrix to them. For example

```
u1(:,[1,3]):=[]
```

removes the first and third columns. Note that

```
u1(:,u2):=[]
```

is compiled but results in run-time error if any entry of u2 becomes less than or equal to zero, or becomes bigger than the number of columns of `u1`. A run-time error is also generated if two entries of `u2` (casted into integers) become identical. The reason is that the size of the result would then change during simulation. This violates the fixed-sized matrix principle discussed later.

2.2.3 Use of OML functions

Functions other than those listed in Section 2.6 may be used if they are **OML** functions available in the context of the **Twin Activate** diagram containing the Matrix Expression block. These **OML** functions can be used only if their arguments do not depend the block inputs; for example the expression

```
[u1;HammWin(L)]
```

is valid (the argument of the Hamming Window function, `L` must be defined, for example in the context of the diagram). In this case, this parameter is available at compile time and its value does not change during the simulation, contrary to `u1`. Note however that even though `u1` may vary in time, its size cannot. So

```
[u1;HammWin(size(u1,2))]
```

is also valid.

Only memoryless **OML** functions should be used because these functions are evaluated by **OML** only

once. So for example the expression

```
UnifRnd()
```

does not produce a random signal during simulation but rather a random output that remains constant during the simulation, contrary to what the user may expect.

2.3 Parser

When the **Twin Activate** diagram is compiled, the matrix expressions of the Matrix Expression blocks present in the model are parsed and converted into pseudo-codes used for evaluation during simulation[1]. The parser is similar to **OML**'s parser, in particular it uses the same rules of precedence, and allows for compositions of functions and operators. This is an important feature for the definition of matrix expressions for which it is not possible to define intermediary variables. For example the **OML** code

```
a=C*u1*u2
=a(1,2)
```

can be expressed (as in **OML**) as a single expression as follows

```
(C*u1*u2)(1,2)
```

There is no limit as to the level of composition allowed, for example expressions such as

```
[inv([u1,u2])(1,:)*u2,u2(1)]'(2)
```

are valid.

Matrix expressions can be expressed over multiple lines. But line breaks and spaces are not separators: in particular spaces cannot be used as matrix column separators and line breaks cannot be used as row separators. The valid **OML** expression

```
[u1 2
  1 3]
```

is not valid as a matrix expression and must be expressed as follows

```
[u1, 2;
  1, 3]
```

2.4 Limitations

2.4.1 Fixed-sized matrix principle

Matrix sizes of all intermediary matrices obtained in the course of evaluating the matrix expression, and of course the final result, are determined at compile time so that no dynamic allocation is used during simulation. This places limitations on the types of expressions allowed. For example

```
[1:u1]
```

is not allowed since the size of the result depends on the value of `u1` (the first input), which may change during simulation.

---

[1]In the future, they will also be used for C code generation

In some cases, even if the the size of the result does not depend on an input, the compiler cannot determine the size. For example the following expression is not accepted

```
[u1(end)+1:u1(end)+5]
```

and has to be re-written as follows

```
u1(end)+[1:5]
```

This means in particular that the expression

```
u2(u1(end)+[1:5])
```

is valid (but may produce a run-time out of index error during simulation depending on the value of `u1(end)`).

2.4.2   All the branches of conditional expressions evaluated

The 'if' and 'switch' clauses are extended to the matrix case and applied element-wise, thus the output may depend on the expressions in multiple branches. That is why all the branch expressions are always evaluated. For the same reason, logical or (`|`) and logical and (`&`) operators are not short circuited.

2.4.3   No support for logical type

The logical data type is not supported so the output of the relational and logical operations are considered regular matrices (which happen to contain only zeros and ones). This means in particular that matrix extraction based on logical tests cannot be used. For example

```
u1(u1>3):=3
```

cannot be used to select the values of `u1` that are larger than three and set them to 3. This can be done using the element-wise if expression

```
.if u1>3 .then 3 .else u1 .end
```

Note that in this simple example the expression can also be implemented as follows

```
min(u1,3)
```

2.5   Data types

2.5.1   Supported data type

The basic numerical data type used in the evaluation of a matrix expression is the matrix of real numbers coded as doubles (float64). The inputs, the output and all intermediary matrix evaluations during simulation use this data type. In particular, no distinction is made between a scalar and a one-by-one matrix.

Parameters defined in the diagram context can have any **OML** data types if in the matrix expression they are only used in the definition of the argument of a **OML** function that produces a matrix of real numbers. Strings can also be used explicitly in defining the expression under the same condition. For example the expression

```
f(1,"row")*u1
```

is valid provided the **OML** function `f` is defined in the context and returns a real matrix with proper size. Note the usage of double-quotes for defining a string. **OML** uses single quotes to define strings.

The reason for which **OML** data types can be used under the above condition is that **OML** functions and in general expressions that do not depend on block inputs are evaluated at compile time by **OML**. Such expressions are then replaced with the resulting matrices of real numbers in the matrix expression.

2.5.2 Unsupported data types

As previously stated, only matrices of real numbers are supported by supported functions. In particular it should be noted that there is no support for the logical data type. The output of logical and relational operators are considered matrices of real numbers containing zeros and ones. In a logical operation, zero is considered false and any non-zero number is true.

Currently complex numbers are not supported.

2.5.3 Data coding

The matrix data is stored in column-major format in memory. All the operations and supported functions use the IEEE standard for 64-bit floating point arithmetic. The block however generates an error if the result of the overall evaluation of the matrix expression, which is the output of the block, contains a NaN (Not a Number).

2.6 Supported operators

2.6.1 Arithmetic operators

The arithmetic operators available for matrix expressions are similar to those available in **OML**.

In the following tables, `<exp>` denotes any valid expression.

Matrix algebra		
`_`	matrix subtraction	`<exp>-<exp>`
	negation	`-<exp>`
`+`	matrix addition	`<exp>+<exp>`
`*`	matrix multiplication	`<exp>*<exp>`
`/`	matrix right division	`<exp>/<exp>`
`\`	matrix left division	`<exp>\<exp>`
`^`	matrix power	`<exp>^<exp>`
`'`	matrix transpose	`<exp>'`

The second operand of the matrix power operation must evaluate to a scalar.

Element-wise matrix operations		
`.*`	element-wise multiplication	`<exp>.*<exp>`
`./`	element-wise left division	`<exp>./<exp>`
`.\`	element-wise right division	`<exp>.\<exp>`
`.^`	element-wise power	`<exp>.^<exp>`

The operand expressions must evaluate to matrices with compatible sizes. For matrix algebra operators, the compatibility follows the standard matrix operations rules. For element-wise operations, both operands must have identical dimensions. In both cases an operand may be scalar, in which case it is "expanded" to a matrix of proper size[2].

2.6.2 Matrix construction and structuring

The following operators and functions can be used to construct matrices out of other matrices but also to construct special matrices. The size of the matrix cannot depend on the values of the inputs but may depend on their sizes since the sizes remain constant during simulation. Note that in some cases the compiler cannot determine that the size of the matrix does not depend on the values of the input; see Section 2.4.1.

Matrix construction		
`[]`	matrix constructor	`[<exp>,...,<exp>;`
`,`	column concatenation	`.........  ;`
`;`	row concatenation	`<exp>,...,<exp>]`
`:`	numeric sequence with step one	`<exp>:<exp>`
	numeric sequence (start:step:end)	`<exp>:<exp>:<exp>`
`eye`	identity matrix	`eye(<exp>)`
`ones`	matrix of ones	`ones(<exp>,<exp>)`
`zeros`	matrix of zeros	`zeros(<exp,<exp>)`

An existing matrix can be manipulated using the following functions

---

[2]The scalar is not really expanded for efficiency reasons but the behavior of the operator is as if it did.

Matrix extraction and assignment		
(,)	sub-matrix extraction	`<exp>(<exp>,<exp>)`
()	vector extraction	`<exp>(<exp>)`
:	all rows	`<exp>(:,<exp>)`
	all columns	`<exp>(<exp>,:)`
	all elements	`<exp>(:)`
end	last row	`<exp>(end,<exp>)`
	last column	`<exp>(<exp>,end)`
	last element	`<exp>(end)`
:=	entry assignment	`<exp>(<exp>):=<exp>`
	column assignment	`<exp>(:,<exp>):=<exp>`
	row assignment	`<exp>(<exp>,:):=<exp>`
	row deletion	`<exp>(:,<exp>):=[]`
	column deletion	`<exp>(<exp>,:):=[]`

2.6.3 Relational and logical operators

The relational operators can be used to compare two matrices of the same size or a matrix and a scalar. In the latter case, the elements of the matrix are each compared with the scalar. In both cases, the result is a matrix including ones and zeros.

Relational operators		
<	element-wise less than	`<exp> < <exp>`
>	element-wise greater than	`<exp> > <exp>`
<=	element-wise less than or equal	`<exp> <= <exp>`
>=	element-wise greater than or equal	`<exp> >= <exp>`
==	element-wise equal	`<exp> == <exp>`
~=	element-wise not equal	`<exp> ~= <exp>`
<>		`<exp> <> <exp>`

The logical operators produce ones and zeros but their arguments can be any number. The number zero is considered false and any other number, true.

Logical operators		
|	element-wise logical or	`<exp>|<exp>`
&	element-wise logical and	`<exp>&<exp>`
~	element-wise logical not	`~<exp>`
and	logical and of all matrix elements	`and(<exp>)`
or	logical or of all matrix elements	`or(<exp>)`

2.6.4 Conditional operators

Conditional operators 'if' and 'switch' are extended to the matrix case. Note that expressions in all branches of conditional operators are evaluated; see Section 2.4.2.

Element-wise if expression   The element-wise if expression operates on matrices and behaves like a standard 'if' expression for each entry of the matrix.

```
.if <cond_exp> .then <exp> .else <exp> .end
```

Normally all three expressions must have identical sizes. But one or two may be scalar. The result is a matrix with entries coming from the expressions in the 'then' and 'else' branches. In particular the (i, j) entry of the resulting matrix is equal to the (i, j) entry of the matrix obtained by evaluating the expression in the 'then' branch (or the value of the expression if it is a scalar) if the (i, j) entry of the matrix obtained by evaluating the condition expression `<cond_exp>` (or the value of the expression if it is a scalar) is not zero. Otherwise it is equal to the (i, j) entry of the matrix obtained by evaluating the expression in the 'else' branch (or the value of the expression if it is a scalar).

Element-wise switch expression   The element-wise switch expression is an extension of the element-wise if expression. The resulting matrix is composed of the entries of the matrices corresponding to expressions returned in different cases.

```
.switch <cond_exp>
  .case <case_exp>, <exp>
  ...
  .case <case_exp>, <exp>
  .otherwise <exp>
.end
```

The (i, j) entry of the resulting matrix is equal to the (i, j) entry of the matrix obtained by evaluating the expression in the branch where the (i, j) entry of the matrix obtained by evaluating the case expression `<case_exp>` is equal to the (i, j) entry of the matrix obtained by evaluating the condition expression `<cond_exp>`.

In case this (i, j) entry matches more than one cases, the first case is taken into account. If it matches no case, the (i, j) entry of the resulting matrix is set to the (i, j) entry of the matrix obtained by evaluating the expression following `.otherwise`. The `.otherwise` "case" must always be present.

2.7 Supported functions

The syntax of a function call is

```
<fun>(<exp>,...,<expr>)
```

The list of arguments may be empty.

2.7.1 Supported functions

Supported functions listed in this section can be used in matrix expressions. Their arguments can be any valid expressions. If all the arguments of a function do not not depend on the values of the inputs, the function is evaluated only once at compile time.

Element-wise functions

The following functions apply to matrices by applying the corresponding scalar function to each element of the matrix.

Trigonometric functions		
acos	inverse cosine	acos(<exp>)
acosh	inverse hyperbolic cosine	acosh(<exp>)
asin	inverse sine	asin(<exp>)
asinh	inverse hyperbolic sine	asinh(<exp>)
atan	inverse tangent	abs(<exp>)
atan2	four-quadrant inverse tangent	atan2(<exp>,<exp>)
atanh	inverse hyperbolic tangent	atanh(<exp>)
cos	cosine	cos(<exp>)
cosh	hyperbolic cosine	cosh(<exp>)
hypot	hypothenuse	hypot(<exp>)
sin	sine	sin(<exp>)
sinh	hyperbolic sine	sinh(<exp>)
tan	tangent	tan(<exp>)

Rounding functions		
ceil	round up	ceil(<exp>)
floor	round down	floor(<exp>)
int	round toward zero	int(<exp>)
round	round to nearest integer	round(<exp>)

Other functions		
abs	absolute value	abs(<exp>)
exp	exponential	exp(<exp>)
log	natural logarithm	log(<exp>)
log10	base 10 logarithm	log10(<exp>)
max	maximum	max(<exp>,<exp>)
min	minimum	min(<exp>,<exp>)
sign	sign (return -1,0 or 1)	sign(<exp>)
sqrt	square-root	sqrt(<exp>)

Matrix functions

The following matrix functions are available

45 Extended Definitions   Altair Twin Activate 2023

Matrix functions		
`det`	determinant of a square matrix	`det(<exp>)`
`diag`	create diagonal matrix from vector or vice-versa	`diag(<exp>)`
`expm`	matrix exponential of a square matrix	`expm(<exp>)`
`inv`	matrix inversion of a square matrix	`inv(<exp>)`
`logm`	matrix log of a square matrix	`logm(<exp>)`
`max`	maximum along rows unless argument is a row vector in which case maximum of all entries	`max(<exp>)`
`min`	minimum along rows unless argument is a row vector in which case minimum of all entries	`min(<exp>)`
`size`	vector including both matrix dimensions	`size(<exp>)`
	row dimension	`size(<exp>,1)`
	column dimension	`size(<exp>,2)`
`sum`	sum along rows unless argument is a row vector in which case sum of all entries	`sum(<exp>)`
	sum along rows	`sum(<exp>,1)`
	sum along columns	`sum(<exp>,2)`
`svd`	singular values	`svd(<exp>)`

2.7.2 Using OML functions

Any **OML** function available in the context of the **Twin Activate** diagram containing the Matrix Expression block can be used in the definition of its matrix expression provided the arguments of the function do not depend on the values of the inputs. Note that `size(<exp>)`, `size(<exp>,1)` and `size(<exp>,2)` may be used in constructing the arguments of the **OML** function regardless of whether or not the argument of the size functions depend on the inputs. The reason is that the output of `size` does not depend on the value of its argument.

Unlike for supported functions, a block parameter defined in a diagram context hides a **OML** function if it has the same name.

Chapter 3

C and OML Custom Blocks in Twin Activate

Twin Activate contains a large set of blocks organized in libraries and made available to the user through various palettes. These blocks provide basic mathematical functions needed to construct complex models of dynamical systems and also provide access to some input/output functionalities for files, graphical widgets, etc. In some cases however, it may be required or more convenient to express dynamical behavior of a block programmatically in **C** or in (**OML**) language , for example to take advantage of existing code or gaining access to OS functionalities in special ways. Constructing new **Twin Activate** blocks is possible in **Twin Activate** and the procedure is presented in the Block Builder documentation. An alternative to constructing a new block is using Custom blocks.

Custom blocks (**C** or **OML**) allow user to specify the dynamical behavior of the block in a block parameter. The actual simulation source code of the block (**C** or **OML**) or a reference to corresponding shared library or **OML** file becomes a parameter value, which is stored in the model where the block is used, and saved in the corresponding **scm** file. The model is thus self-contained -in other words, a library is not referenced by the block to define its behavior as is the case for most blocks. Indeed, **CCustomBlock** and **OmlCustomBlock** blocks are not new blocks.

So compared to creating and using new blocks, Custom blocks have the advantage of not requiring any library management operations. But even though their look and feel may be customized to some extent (in particular by encapsulating them in a masked Super Block) Custom blocks are not always the best solution. With Custom blocks, there is no mechanism for automatically upgrading the code of Custom blocks as there is for library blocks. Custom blocks are often used as a first step to develop and test **C** simulation codes destined for new blocks.

Both **CCustomBlock** and **OmlCustomBlock** blocks also provide the option of not including the source code of the simulation function. The simulation function of a **CCustomBlock** block may be provided by a shared library (dll file under Windows operating system). Instead of the **C** code, the block parameters include the path to the shared library and the name of the entry point. This option is particularly interesting for preserving intellectual property but note that the model is no longer self-contained. The shared library file must be distributed along with the model.

Similarly for **OmlCustomBlock** blocks the **OML** code does not need provided. An **OML** file defining the function and its name may used as block parameters instead. If a file is not specified **Twin Activate** assumes that the function is already defined in the environment.

The **MathExpression** and **MatrixExpression** blocks should be considered before using Custom blocks since they provide even simpler ways to express the dynamical behavior of a block. They are however limited to expressing stateless dynamics: block outputs depend on block inputs alone, and the expressions support a fixed set of functions that cannot be extended by the user.

3.1 Twin Activate block simulation function

A **Twin Activate** block is defined by its graphical properties (specified in XML), an **OML** code for its instantiation and parameterization, and its simulation routines which can be in **C** or **OML**. The Block Builder tool provides a user interface for interactively designing a new block, in particular its graphics, and the **OML** code for its instantiation and parameterization. For using the **CCustomBlock** and **OML** custom blocks however users need not worry about these aspects; Custom blocks provide generic graphics and parameters allowing users to focus on developing the simulation function.

Block simulation functions are called by the **Twin Activate** simulator during simulation to perform different tasks, such as updating the output or the state. Simulation functions have two arguments: `block` and `flag`. The task to be realized is specified by the argument `flag`. The argument `block` is a the block's data structure to be passed to the APIs to get and set block data. Block simulation code may be in **C** or **OML**.

3.2 Custom block parameter GUI

CCustomBlock and **OmlCustomBlock** blocks' parameter GUIs provide an interactive user interface to define block properties. Users can define the number and types of inputs and outputs, the states and simulation parameters (if any) and other properties. The interface also gives access to an editor for writing the block's simulation code.

Developing a simulation code requires a knowledge of the APIs needed to get and set block data, for example for reading block inputs and writing block outputs. But Custom blocks provide a utility to generate a skeleton of the simulation code by taking into account the block properties specified by the user. This significantly facilitates the development of the simulation code.

Block inputs, outputs and simulation parameters are named and the code skeleton generator takes into account this information in order to provide a code where the user, in most cases, simply needs to add instructions (in **C** or **OML**) involving the provided names as program variables. These variables are already declared and initialized when needed.

The Custom block parameter GUI, illustrated in Fig. 3.1 (the **C** block; **OML** block is similar), could be considered as an interactive interface. It contains five tabs for customizing the properties of the block.

- Ports tab is used to specify the inputs and outputs of the block. For regular ports (not activation ports), the port sizes, types, feedthrough properties and names can in particular be specified.

- States tab is used to initialize block states, both discrete and continuous-time. Discrete object states oz may be typed and named.

  If the simulation function is declared to be implicit in the SimFunctions tab, the initial value of the continuous-time state derivative should also be specified.

- Parameters tab is used to set the value of $rpar$ and $ipar$ parameters; a list of $opar$ parameters may also be defined, typed and named.
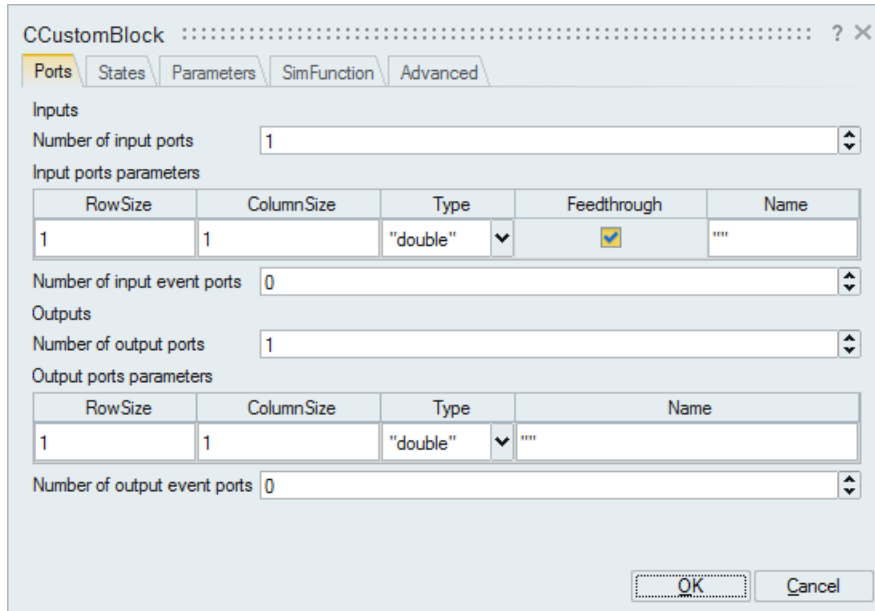
Figure 3.1: Block parameter GUI of **CCustomBlock** block.

These parameters are simulation parameters that are available to the simulation function. The object parameters ($opar$) may be matrices of various sizes and data types (including complex and various integer data types). Parameters $rpar$ and $ipar$ are vectors of real and integer data types.

Unlike for the **CCustomBlock** block, the use of integer data types in an **OmlCustomBlock** block is not useful since all integer parameters are converted to doubles in the **OML** code. The **OML** language does not currently support integer data types.

- SimFunction tab is used to specify the simulation function of the block. The code may be inlined, in which case the interface can be used to define the simulation code in **C** for **CCustomBlock** blocks and in OML for **OmlCustomBlock** blocks. A skeleton of the code may be generated based on information provided in other tabs. See Fig. 3.2.

  If the code is not inlined, then its location must be specified. In case of a **CCustomBlock** block, the path to the shared library and the name of entry point are given. In case of an **OmlCustomBlock** block, the path to an **OML** file and the name of the **OML** function are provided.

- Advanced tab may be used to define the number of zero-crossings and modes; the activation mode of the block is also defined in this tab. The activation mode options are *Always active*, *Initially active* and *Standard*. In Standard mode, the block is activated by inheritance if it does not contain any input activation ports.

  In most cases, if the block does not contain a continuous-time state, the Standard mode should be used. On the other hand, if the block contains continuous-time states, the the Always active mode is often more appropriate. Initially active blocs are rarely constructed as custom blocks.
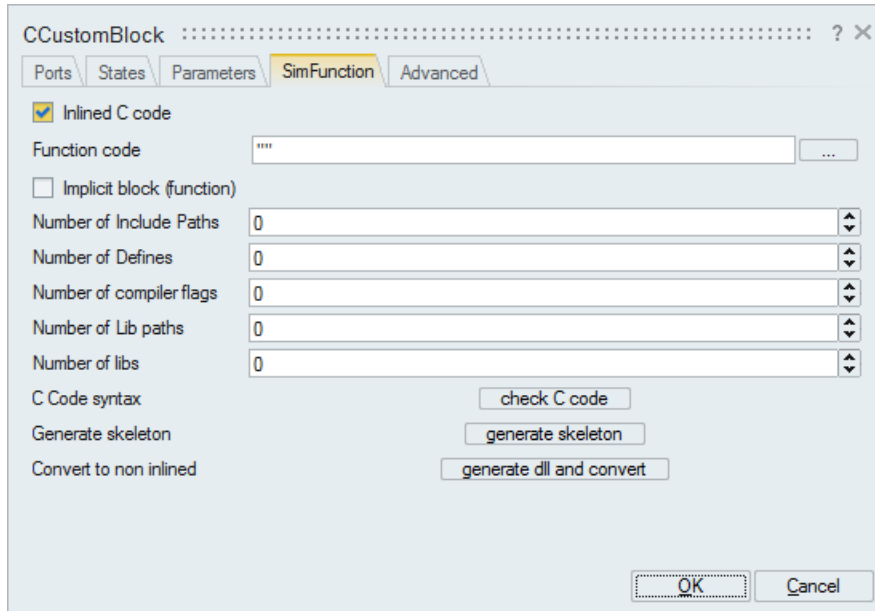
Figure 3.2: SimFunction tab used to specify the simulation function.

3.3 Examples

The usage of the Custom block parameter GUI is illustrated through a number of examples in this section.

3.3.1 Median block

A block that computes the median of the entries of its input matrix is considered. This block is constructed using Custom blocks. This is a simple exercise since the block has no states and no parameters. It has one input and one output, both assumed to be of type double. The size of the input is set to $[-1, -2]$ indicating that its size is arbitrary. The size of the output is $[1, 1]$. The names of the input and output are respectively set to `mat` and `med`.

Using a **OmlCustomBlock** block, generating the skeleton of the simulation code provides the following code Note that the instruction computing the output is commented so that the simulation can be run without error even prior to the modification of the code. It should be uncommented when the code to compute the variable `med` is added.

OML contains already a function `Median` that computes the median of each column of the matrix argument as a vector. To use this function to implement the **Median** block, the input may be converted to a column vector using the column operator. The skeleton code can thus be edited as follows:

The code has been simplified by excluding the initialization and terminal phases. In this case the block does not perform any special operations in these phases.

The resulting block is tested in this model

Figure 3.3: Skeleton of OML code

Figure 3.4: Edited Skeleton of OML code
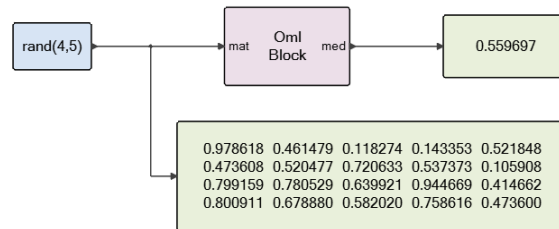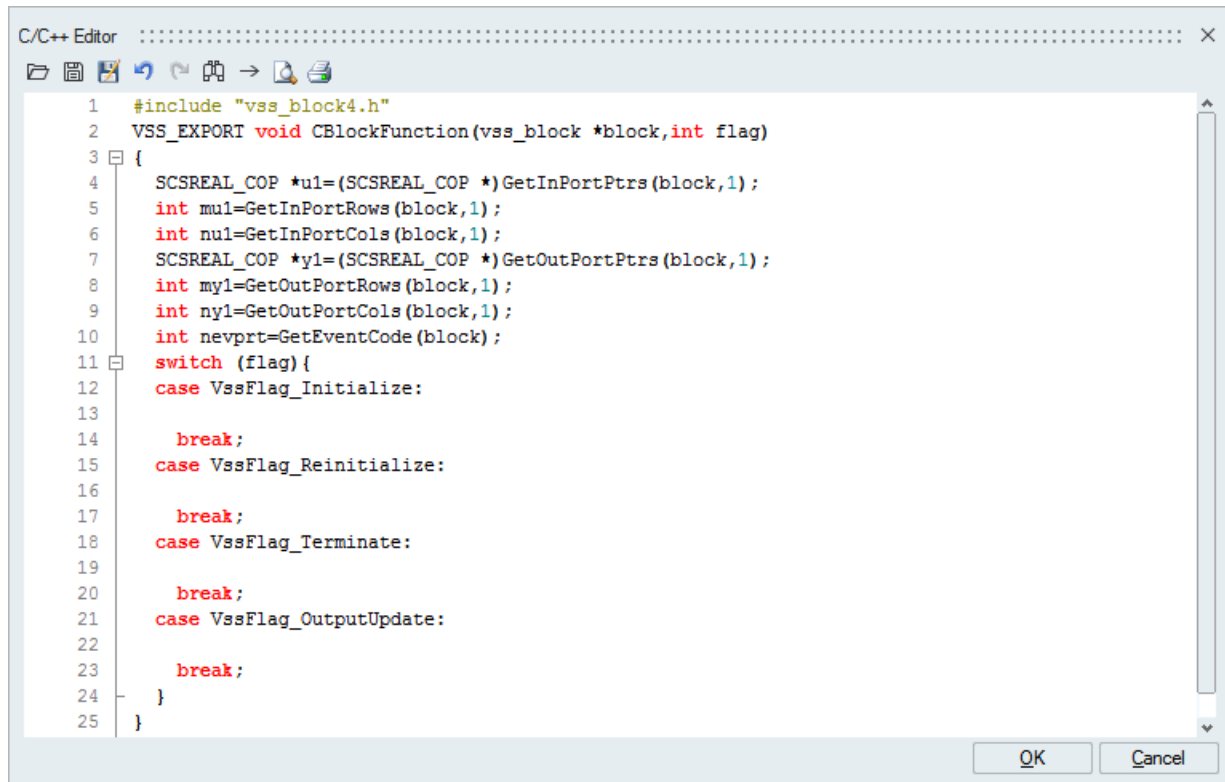
Figure 3.5: Median OML model (median_OML.scm)

The **OmlCustomBlock** block may be encapsulated in a masked Super Block to hide its generic parameter GUI, making it look like a basic block. In this case the block has no parameter.

The **CCustomBlock** block may be used in a very similar way. Generating the skeleton of the simulation

code for the same example provides the following **C** code


```c
#include "vss_block4.h"
VSS_EXPORT void CBlockFunction(vss_block *block,int flag)
{
    SCSREAL_COP *u1=(SCSREAL_COP *)GetInPortPtrs(block,1);
    int mu1=GetInPortRows(block,1);
    int nu1=GetInPortCols(block,1);
    SCSREAL_COP *y1=(SCSREAL_COP *)GetOutPortPtrs(block,1);
    int my1=GetOutPortRows(block,1);
    int ny1=GetOutPortCols(block,1);
    int nevprt=GetEventCode(block);
switch (flag){
case VssFlag_Initialize:

  break;
case VssFlag_Reinitialize:

  break;
case VssFlag_Terminate:

  break;
case VssFlag_OutputUpdate:

  break;
}
}
```

Figure 3.6: Skeleton of C code

There are different algorithms for computing the median, see for example [1]. The public domain **C** code `torben.c` from [1] is used here to implement the block's simulation code. See Fig. 3.7. This code has the advantage of not allocating runtime memory and leaving the input matrix intact.

```
C/C++ Editor  ··············································································  ×

  1    typedef double elem_type ;
  2
  3    elem_type torben(elem_type m[], int n)
  4  { {
  5         int          i, less, greater, equal;
  6         elem_type  min, max, guess, maxltguess, mingtguess;
  7
  8         min = max = m[0] ;
  9         for (i=1 ; i<n ; i++) {
 10             if (m[i]<min) min=m[i];
 11             if (m[i]>max) max=m[i];
 12         }
 13
 14         while (1) {
 15             guess = (min+max)/2;
 16             less = 0; greater = 0; equal = 0;
 17             maxltguess = min ;
 18             mingtguess = max ;
 19             for (i=0; i<n; i++) {
 20                 if (m[i]<guess) {
 21                     less++;
 22                     if (m[i]>maxltguess) maxltguess = m[i] ;
 23                 } else if (m[i]>guess) {
 24                     greater++;
 25                     if (m[i]<mingtguess) mingtguess = m[i] ;
 26                 } else equal++;
 27             }
 28             if (less <= (n+1)/2 && greater <= (n+1)/2) break ;
 29             else if (less>greater) max = maxltguess ;
 30             else min = mingtguess;
 31         }
 32         if (less >= (n+1)/2) return maxltguess;
 33         else if (less+equal >= (n+1)/2) return guess;
 34         else return mingtguess;
 35  }
 36
 37    #include "vss_block4.h"
 38    VSS_EXPORT void CBlockFunction(vss_block *block,int flag)
 39  {
 40      SCSREAL_COP *mat=(SCSREAL_COP *)GetInPortPtrs(block,1);
 41      int mmat=GetInPortRows(block,1);
 42      int nmat=GetInPortCols(block,1);
 43      SCSREAL_COP *med=(SCSREAL_COP *)GetOutPortPtrs(block,1);
 44      switch (flag){
 45      case VssFlag_OutputUpdate:
 46        med[0]=torben(mat, mmat*nmat);
 47        break;
 48      }
 49  }
```

                                              OK   Cancel

Figure 3.7: **C** simulation code of the Median block.

3.3.2 Variable discrete delay

A discrete delay block is considered where the amount of delay is variable and is given by an additional block input. Let u_1 and u_2 be the block inputs, then the output is given by

$$y(k) = u_1(k - u_2(k)),$$

The delay input $u_2(k)$ is supposed to be receive values between 1 and N. But if $u_2(k)$ is less than 1 then the delay is set to 1; if it is larger than N then it is set to N.

The block is implemented using a circular buffer of size N placed in the discrete state z of the block. An integer state $oz(1)$ is used to track the position of the last entry of z used to store the latest value of the input.

The **CCustomBlock** block is used to implement the Variable delay block. The block has two inputs. For simplicity, we assume both are scalars but the extension to vector case is straightforward. The first input u_1 is the delayed signal, which contrary to the second input representing the amount of delay, does not have feedthrough property. See Fig. 3.8 for the parameterization of the ports of the **CCustomBlock** block.
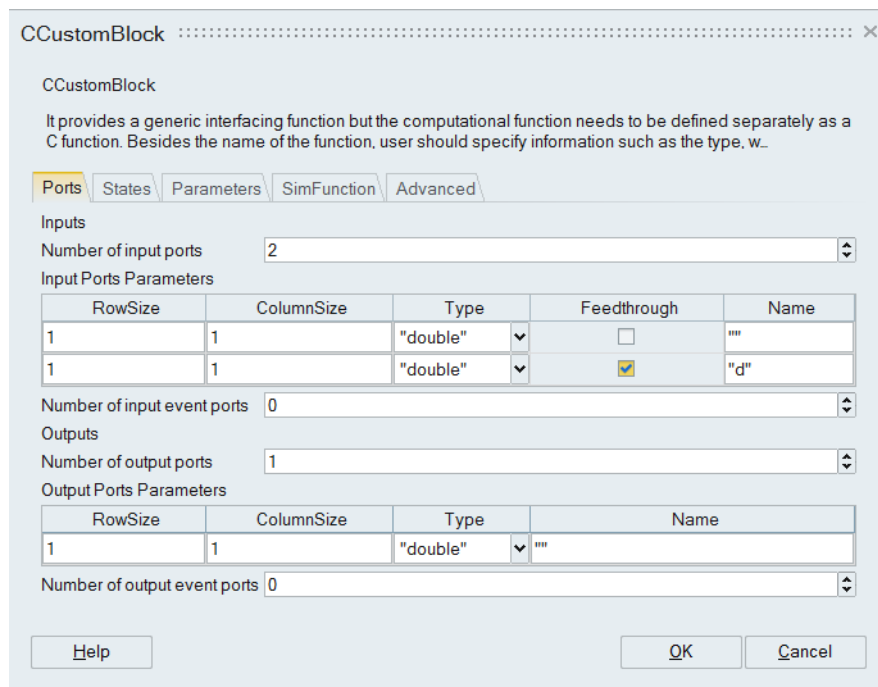
Figure 3.8: Parameterization of the ports of the Variable delay block.

The states are initialized as illustrated in Fig. 3.9. The buffer entries are initially set to zero and an integer state is used to indicate the current position in the buffer.

The block has no parameters and the simulation code is given in Fig. 3.10.

The following model is used to test the Variable delay block:
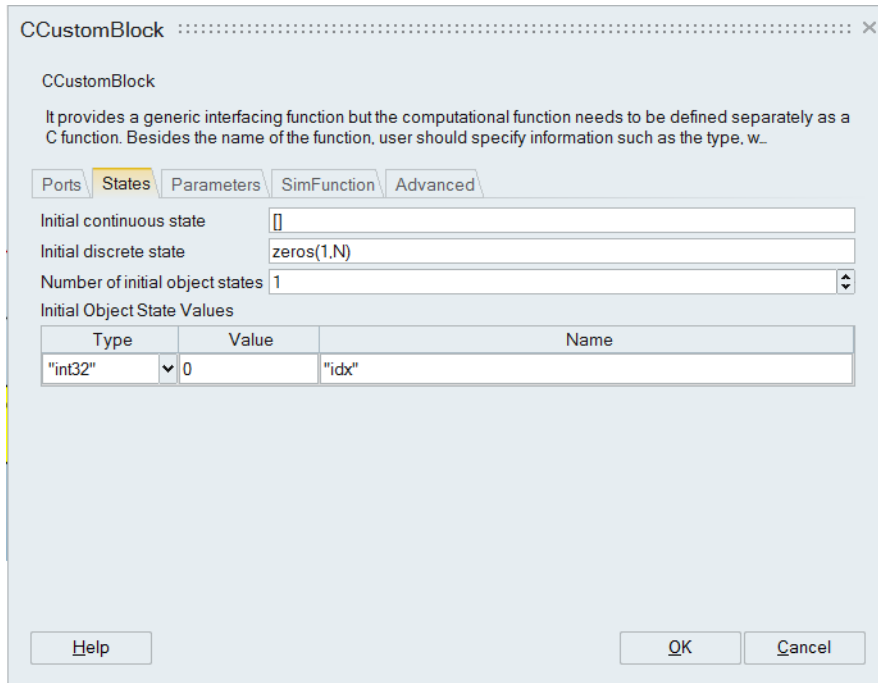
Figure 3.9: Parameterization of the states of the Variable delay block.
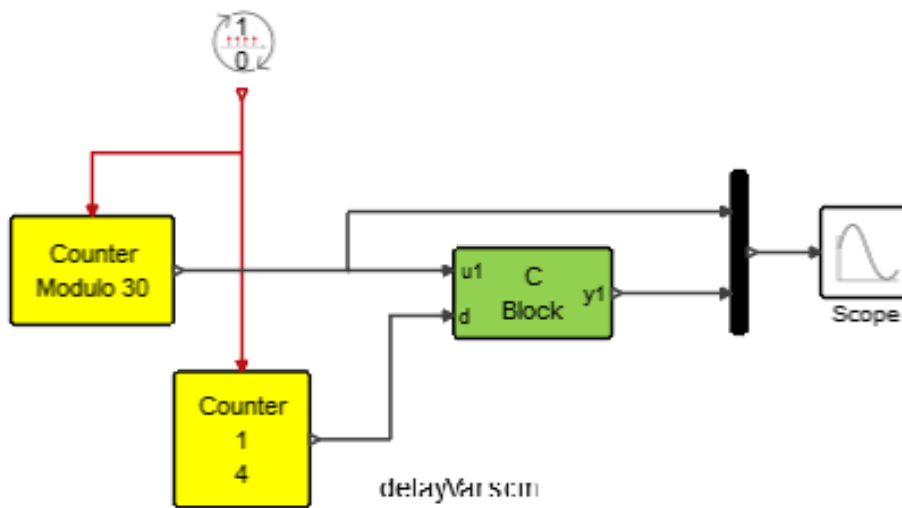
Figure 3.11: Variable Delay model (delayVar.scm)

The simulation result is given in Fig. 3.12.

```
C/C++ Editor ···································································· ×

  1    int mod(int a, int b)
  2    {
  3         int r = a % b;
  4         return r < 0 ? r + b : r;
  5    }
  6
  7    #include "vss_block4.h"
  8    VSS_EXPORT void CBlockFunction(vss_block *block,int flag)
  9    {
 10      SCSREAL_COP *u1=(SCSREAL_COP *)GetInPortPtrs(block,1);
 11      SCSREAL_COP *d=(SCSREAL_COP *)GetInPortPtrs(block,2);
 12      SCSREAL_COP *y1=(SCSREAL_COP *)GetOutPortPtrs(block,1);
 13      int nevprt=GetNevin(block);
 14      int nz=GetNdstate(block);
 15      SCSREAL_COP *z=GetDstate(block);
 16      SCSINT32_COP *idx=(SCSINT32_COP *)GetOzPtrs(block,1);
 17      int i,n;
 18      switch (flag){
 19      case VssFlag_OutputUpdate:
 20        n=(int)(*d-1);
 21        if(n>nz-1) n=nz-1;
 22        else if(n<0) n=0;
 23        i=mod(*idx- n,nz);
 24        y1[0]=z[i];
 25        break;
 26      case VssFlag_StateUpdate:
 27        *idx = mod(*idx +1,nz);
 28        z[*idx]=*u1;
 29        break;
 30      }
 31    }

                                        OK      Cancel
```

Figure 3.10: Parameterization of the states of the Variable delay block.
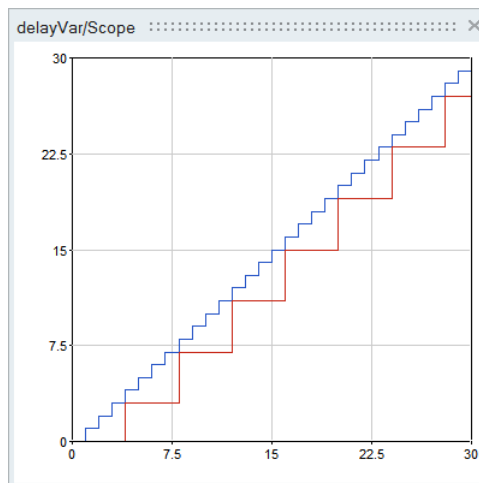
Figure 3.12: Simulation results show that the input signal is delayed differently at each step.

3.3.3 Fluid flow

The fluid flow between two tanks is modeled. The fluid levels in the tanks are denoted by h_1 and h_2 satisfying the following equations:

$$
\begin{aligned}
\dot{h}_1 &= -\frac{A_\mathsf{d}}{A_\mathsf{t}} V \\
\dot{h}_2 &= \frac{A_\mathsf{d}}{A_\mathsf{t}} V
\end{aligned}
$$

where V represents the speed of flow. A_t represents the cross-sectional areas of the tanks. A_d represents the cross-sectional area of the pipe connecting the two tanks:

$$
A_\mathsf{d} = \frac{\pi D^2}{4} \tag{3.1}
$$

where D is the diameter of the pipe.

The speed V satisfies the Bernoulli equation:

$$
h_1 + H = h_2 + f\frac{2L}{D}\frac{V^2}{2g} + \frac{V^2}{2g}
$$

with H denoting the contribution of the pump. $H = 0$ if the pump is off; when the pump is on:

$$
H = 4 - 8.0 \times 10^4 Q^2. \tag{3.2}
$$

The flow Q is given by the equation of continuity:

$$
Q = A_\mathsf{d} V. \tag{3.3}
$$

The friction factor f satisfies one of following relations depending on the flow phase: turbulent, laminar or transitional:

$$
\begin{cases}
\dfrac{1}{\sqrt{f}} = -2\log_{10}\left(\dfrac{e}{3.7} + \dfrac{2.51}{R_\mathsf{e}\sqrt{f}}\right) & \text{if } R_\mathsf{e} > R_\mathsf{eT} \\
f = \dfrac{K_\mathsf{s}}{R_\mathsf{e}} & \text{if } R_\mathsf{e} < R_\mathsf{eL} \\
f = f_\mathsf{L} + \dfrac{R_\mathsf{e} - R_\mathsf{eL}}{R_\mathsf{eT} - R_\mathsf{eL}}(f_\mathsf{T} - f_\mathsf{L}) & \text{Otherwise}
\end{cases}
$$

The phase depends on the value of R_e, the Reynold's number:

$$
R_\mathsf{e} = \frac{DV\rho}{\mu} \tag{3.4}
$$

This system cannot be readily expressed as a system of ordinary differential equations and must be modeled as an implicit system, i.e., a system of differential algebraic equations (DAE). Such systems may be modeled as **Twin Activate** blocks, called implicit blocks. **C** and **OML** Custom Blocks can be used to implement implicit blocks.

The difference between an explicit block and implicit block is that in an explicit block the continuous-time dynamical behavior is an explicit system

$$
\begin{aligned}
\dot{x} &= F(t, x, u) \\
y &= G(t, x, u)
\end{aligned}
$$

Here u and y represent respectively the inputs and the outputs of the block. The block simulation code provides the functions F and G.

The continuous-time dynamical behavior of an implicit block is

$$\begin{aligned} 0 &= E(t, x, \dot{x}, u) \\ y &= H(t, x, \dot{x}, u) \end{aligned} \tag{3.5}$$

The block simulation code provides the functions E and H. Unlike the explicit case where the simulation code must compute $\dot{x}$, in the implicit case the code computes a residual, which may depend on $\dot{x}$.

Since algebraic relationships may be included in (3.5), it is possible to include all system equations in (3.5). In this case, the size of the state x (so the size of the residual) would be equal to the number of equations in the system. But for better performance and precision, it is better to reduce the size of residual by computing explicitly system variables, when possible, and thus removing as many algebraic equations as possible. Here in particular instead of including Equations (3.1), (3.2), (3.3) and (3.4) in (3.5), A_d, H, Q and R_e are computed explicitly and the state of the block is reduced to

$$x = \begin{pmatrix} h_1 \\ h_2 \\ V \\ f \end{pmatrix}.$$

The **C** simulation code in **CCustomBlock** block for this example is given in Fig. 3.13.

All system parameters are defined as block parameters, so they can be modified without having to edit the **C** code. The block has one input (`pump`) taking values 0 and 1 indicating respectively that the pump is off or on. The block outputs are the fluid levels in the tanks.

Note that in the **CCustomBlock** block parameter GUI, the **Implicit block** property checkbox must be checked, the initial state specified and the **Activation mode** be set to **Always Active mode**.

The following model is used to test the two-tank block:
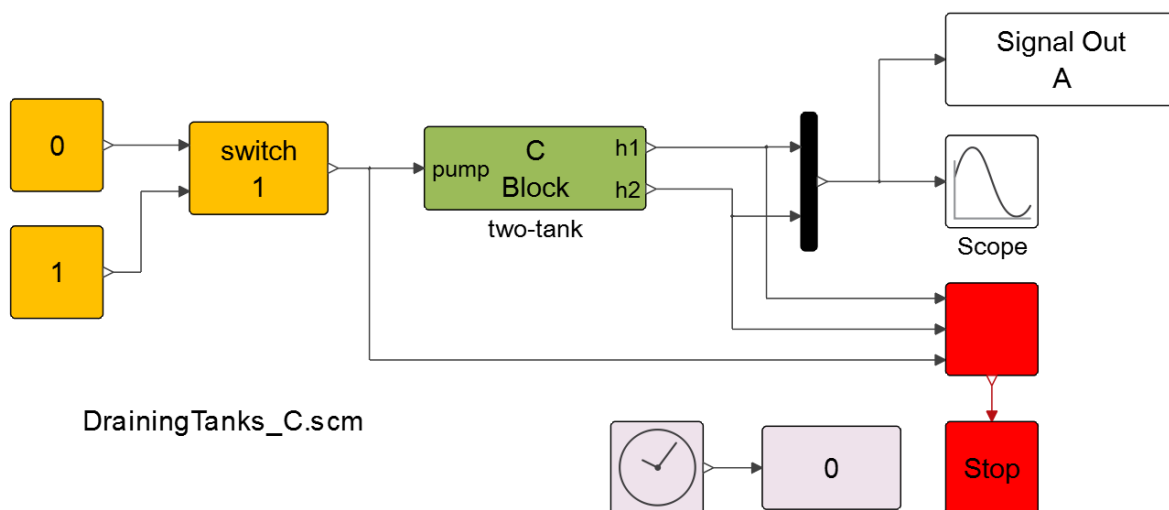
Figure 3.14: DrainingTanks C model (DrainingTanks_C.scm)

Figure 3.13: **C** simulation code of the two-tank block.

The pump may be turned on and off by double clicking on the **Switch** block. With the pump off, the simulation stops when the fluid levels are equal and the simulation time is indicated in the `Display` block.

The simulation results for both the case the pump is on and off are given in Fig. 3.15:

Altair Twin Activate 2023
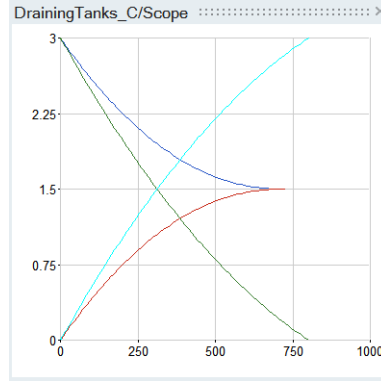
Figure 3.15: Simulation results: h_1 and h_2 with pump off and on.

3.3.4 Simplified explicit automaton block

In chapter 17, the implementation of hybrid automata in **Twin Activate** has been illustrated. The Automaton block[1] is an implicit block which allows modeling general hybrid automata with Differential algebraic equations (DAE). In this section we try to demonstrate the way a simplified Automation block for explicit hybrid automata with ordinary differential equation (ODE) is built. The sticky balls example (17.3.3) will be implemented using **CCustomBlock** block with the new explicit automation block.

As explained in (17.3.3), the sticky balls model works in two modes, stuck and separate. In each mode the dynamical model is described as an ODE of size four (see 17.11, 17.12, and 17.19). In order to simulate the sticky balls models, the **CCustomBlock** block contains the state vector and outputs it. The state vector, in separate mode and stuck mode is $[x_1, x_2, v_1, v_2]$ and $[x_3, v_3, s, q]$, respectively. The block outputs the state vector at the second output port whatever the mode is. The derivative of the states should be computed as a function of states and given back to the **CCustomBlock** block as input. The derivative of state in separate mode is fed to the first input and that of the stuck is given to the second input. During the mode transition, the initial value of state vector should be given to **CCustomBlock** block to reinitialize the state vector. The initial value of states are concatenated with the state derivative of the destination mode. In order to detect a mode transition, a zero-crossing function is used in each mode. In the separate mode, the zero-crossing function in separate and stuck modes are 17.18 and 17.20, respectively. The zero-crossing functions are concatenated with the state-derivative and initial vector for each mode. The overall model of the sticky balls are shown in Fig. 3.16.

The first output of the **CCustomBlock** block contains the current mode and the second output is the state vector.

See Fig. 3.17 for the parameterization of the ports of the **CCustomBlock** block.

The states are initialized as illustrated in Fig. 3.9.

The block has two integer parameters, i.e., number of modes of the automaton, the initial mode. There are also two object parameters of type integer, i.e., the destination modes, when the zero-crossing happens. In the case of sticky balls model, the destination modes is simple, when the mode is One, the destination mode when a zero-crossing happens, is Two and when the mode is Two the destination mode is One. The parametrization is given in Fig. 3.19.

The number of zero-crossing function is one and the block has no internal mode for automatic handling

---

[1]Available in the Hybrid palette.
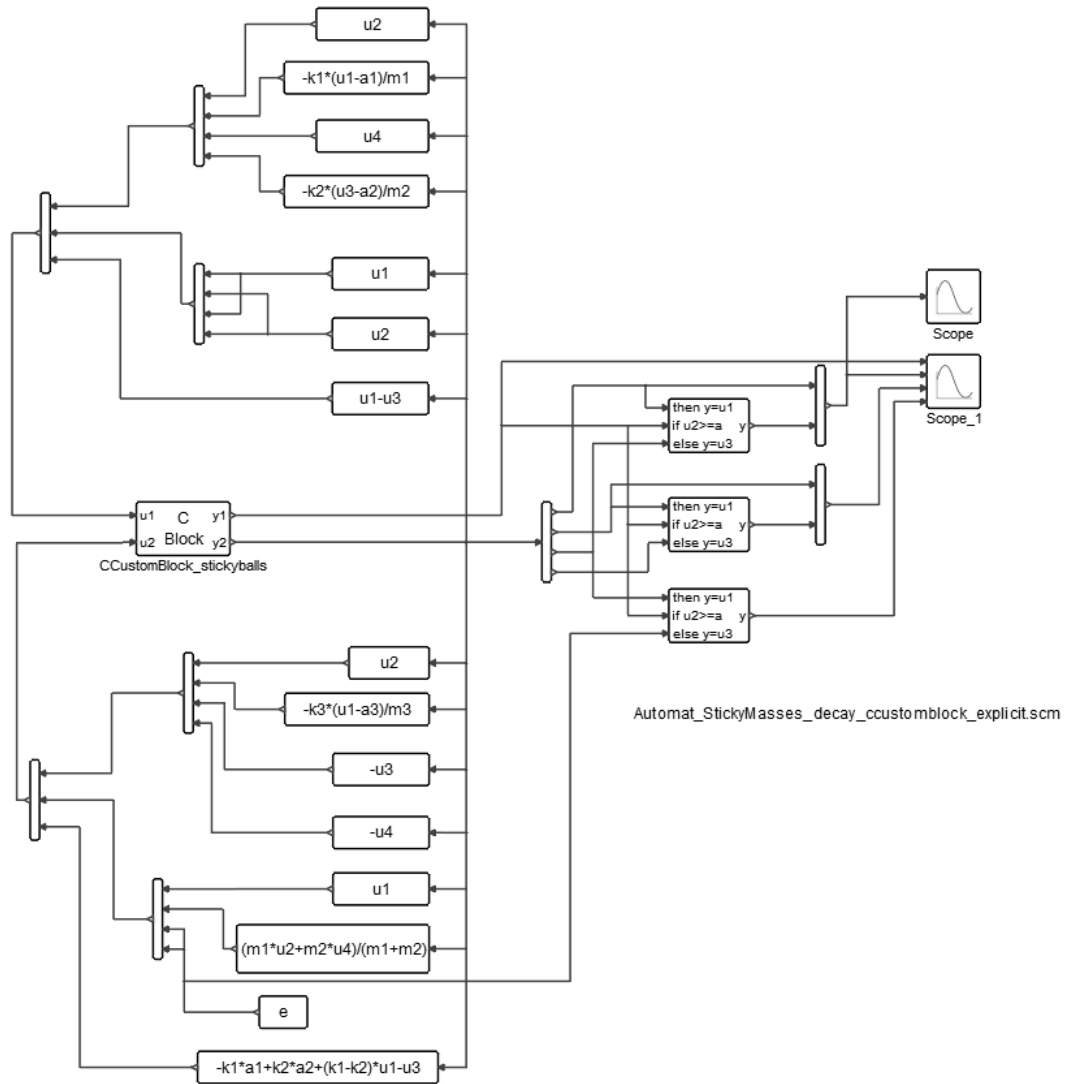
Figure 3.16: Complete model of sticky masses
(Automat_StickyMasses_decay_ccustomblock_explicit.scm)

the zero-crossing. The **C** simulation code in **CCustomBlock** block for this example follows:

```
 #include "vss_block4.h"

typedef struct {
int cMode, pMode;
}automat_t;

#define WD      ((automat_t *) GetWorkPtrs(block))
#define _cMode  (WD->cMode)
#define _pMode  (WD->pMode)

VSS_EXPORT void CBlockFunction(vss_block *block,int flag)
{
double * y0, *y1, *ui;
double* g=GetGPtrs(block);
double* x=GetState(block);
double* xd=GetDerState(block);
double* evout= GetNevOutPtrs(block);
```
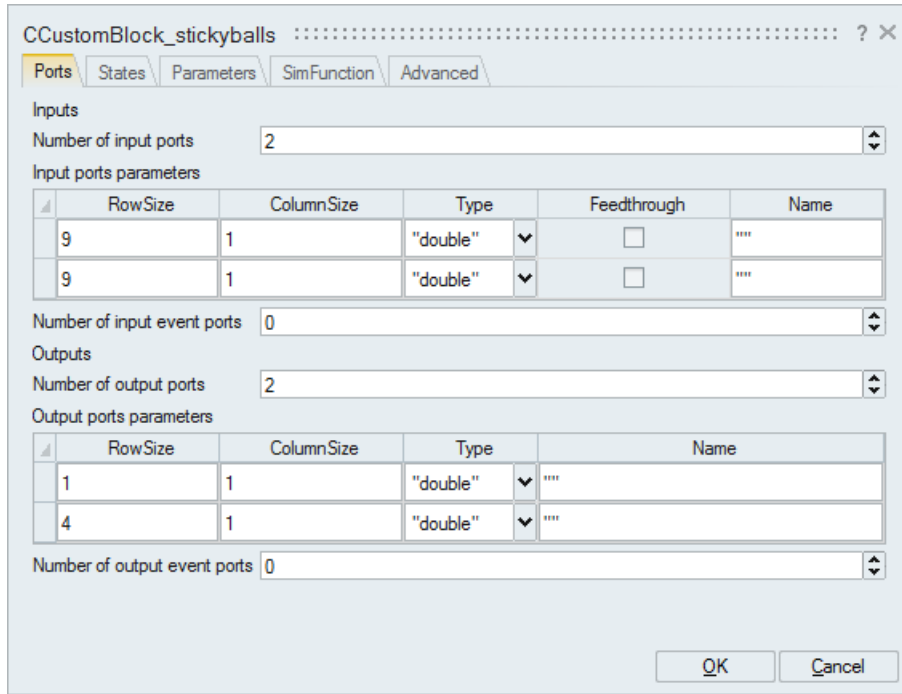
Figure 3.17: Parameterization of the ports of the explicit automaton block for sticky balls model.

```
int* ipar=GetIparPtrs(block);
double* rpar=GetRparPtrs(block);
int* jroot=GetJrootPtrs(block);
int ng=GetNg(block);
int NX=GetNstate(block);
int* tmodev, NMode, Minitial,j,k;
double time=GetVssTime(block);

NMode=ipar[0];
Minitial=ipar[1];

switch(flag)
{
case VssFlag_Initialize:
{
if ((GetWorkPtrs(block)=(automat_t*) vss_malloc(block,sizeof(automat_t)))== NULL )
            { SetBlockError(block,1); return; }
_pMode=_cMode=Minitial;
}
break;
case VssFlag_OutputUpdate:
{
y0=GetRealOutPortPtrs(block,1);
y1=GetRealOutPortPtrs(block,2);
y0[0]=(double) _cMode;
            memcpy(y1,x,sizeof(double)*NX); y1+=NX;
}
break;

case VssFlag_Derivatives:
{
ui=GetRealInPortPtrs(block,_cMode);
memcpy(xd,ui,sizeof(double)*NX);
}
break;
case VssFlag_ZeroCrossings:
```
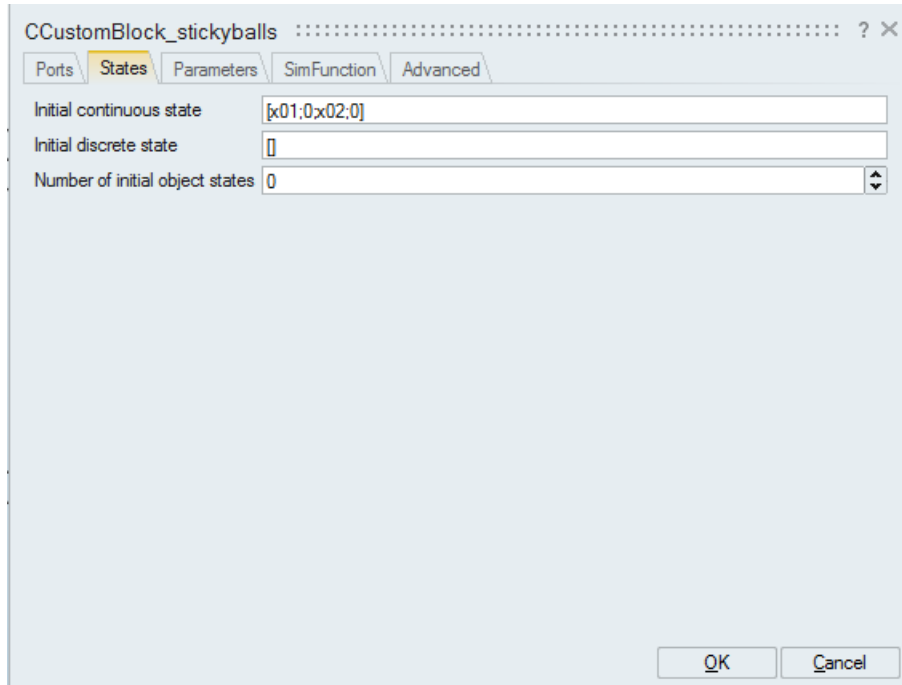
Figure 3.18: Parameterization of the states of the explicit automaton block for sticky balls model.

```
{
ui=GetRealInPortPtrs(block,_cMode);
for (j=0;j<ng;j++) g[j]=0;
for (j=0;j<GetInPortRows(block,_cMode)-2*NX;j++) {
g[j]=ui[j+2*NX];
}
}
break;

case VssFlag_StateUpdate:
{
if (!( isZeroCrossing(block)
                || GetSimulationPhase(block)==PHASE_DISCRETE
                || GetSimulationPhase(block)==PHASE_MESHPOINT )) return;

tmodev=Getint32OparPtrs(block,_cMode);
ui=GetRealInPortPtrs(block,_cMode);

for (k=0;k<GetInPortRows(block,_cMode)-2*NX;k++){
ui=GetRealInPortPtrs(block,_cMode);
if( (isZeroCrossing(block) && jroot[k]==1) ||
(GetNevIn(block)==1 && ui[k+2*NX]>0 )   ||
(GetSimulationPhase(block)==PHASE_DISCRETE  && ui[k+2*NX]>0) ||
(GetSimulationPhase(block)==PHASE_MESHPOINT && ui[k+2*NX]>0)
) {
_pMode=_cMode;
_cMode=tmodev[k];
            ui=GetRealInPortPtrs(block,_cMode);ui+=NX;
memcpy(x,ui,sizeof(double)*NX);
DoColdRestart(block);
break;
}
}
}
break;
default:
```
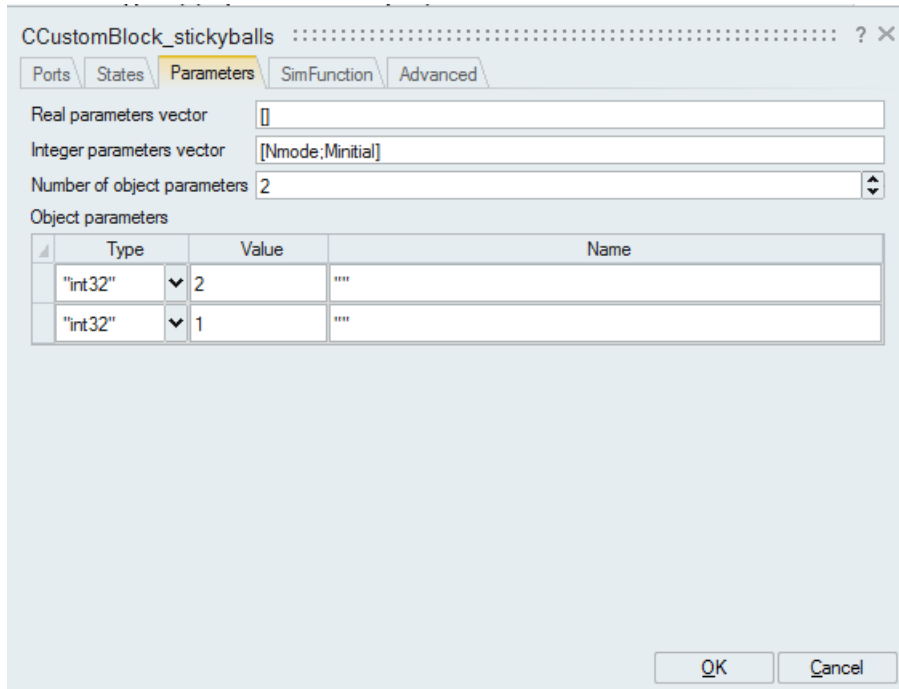
Figure 3.19: Parameterization of the parameters of the explicit automaton block for sticky balls model.

```
break;
}
}
```

In order to hold the current and previous modes, they are put into a structure and stored in the block's internal workspace. The workspace is allocated and initialized in `VssFlag_Initialize` flag using block integer parameters.

In the `VssFlag_OutputUpdate` flag, the outputs are computed. The first output is straightforward and in order to compute the second output, the state vector `x=GetState(block)` is copied into the output using the **C** `memcpy` command. The size of the state vector is obtained using `nx=GetNstate(block)`.

In the `VssFlag_Derivatives` flag, first, based on the current mode `_cMode` the pointer to the appropriate input port is grabbed, then the first `nx` elements are copied into the state-derivative vector (`xd=GetDerState(block)`).

In the `VssFlag_ZeroCrossings` flag, first, based on the current mode `_cMode` the pointer to the appropriate input port is grabbed. The first `2nx` elements of the input port are state derivative and initial value of states the next `nz=GetNg(block)` elements are the zero-crossing functions. Hence, these elements are copied into the state-derivative vector (`g=GetGPtrs(block)`).

In the `VssFlag_StateUpdate` flag, the mode transition is performed. The mode can change at zero-crossing instants or at non-try simulator points such as discrete events or simulator meshpoints. At other instants, the call to the `VssFlag_StateUpdate` flag is ignored. If a valid transition happens, the zero-crossing surface of the next mode will be positive. Based on this, the positive zero-crossing is found and the corresponding mode is found. Then the internal state of the **CCustomBlock** block is updated.

The simulation result of the model 3.16 is given in fig. 3.20.
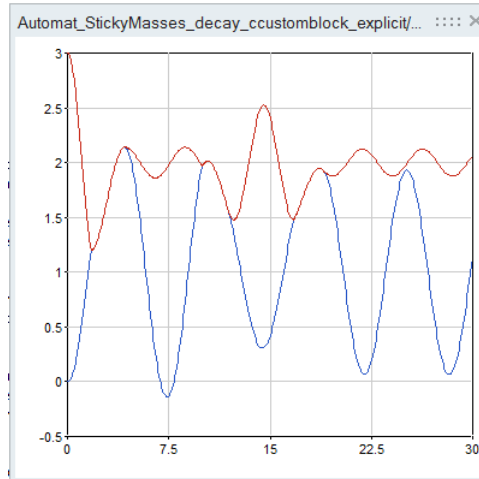
Figure 3.20: Simulation result of the model 3.16.

3.4 Non-inlined simulation functions

The **C** and **OML** functions used in custom blocks need not necessarily be provided in the block (be inlined). They can be provided in a shared library (dll file under Windows) for **CCustomBlock** blocks or **OML** files for **OmlCustomBlock** blocks. The choice of inlining the function is made using a checkbox in the SimFunction tab. For example in Fig. 3.21 the simulation function of the **CCustomBlock** block is defined in the file **foolib.dll**. Since the path of the file is not specified, the file is supposed be in the same directory as the model.
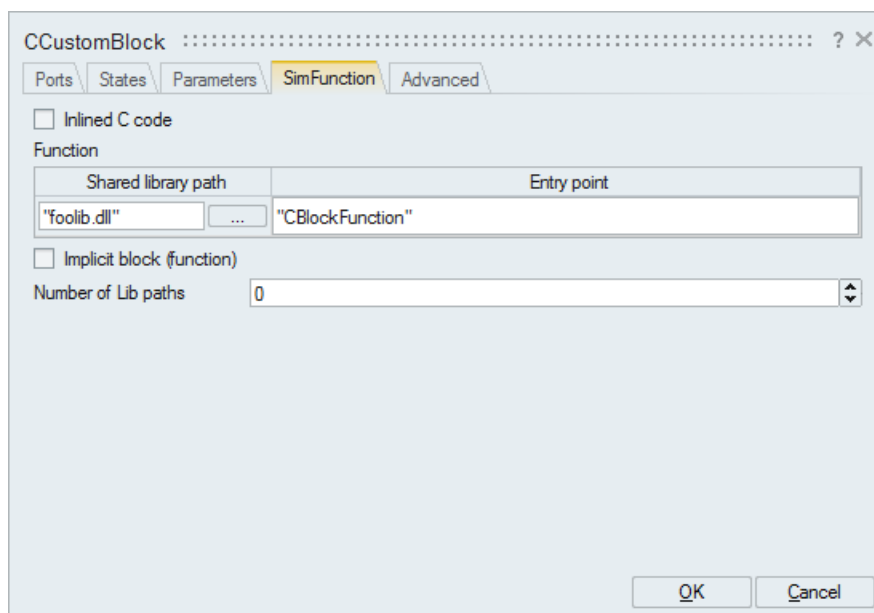
Figure 3.21: SimFunction tab used to specify the simulation function. In this case the function is not inlined.

A convenient way to create a **CCustomBlock** block without inlined code is first to create one with an inlined code. This way, the skeleton generator facility can be used and the code can be tested.

Moreover, the shared library is automatically created by **Twin Activate**. Once the code is validated, the block may use the shared library instead of the inlined code. This can be done automatically using the convert to non inlined facility. In this case the inlined checkbox is unchecked and the library information is provided in the SimFunction tab. Note that the "Function code" becomes an invisible parameter but its content, the original **C** code, can be retrieved by marking the inlined checkbox. So if the objective is to hide the code (in particular for IP protection), make sure to erase the content of the "Function code" parameter.

If the shared library of the non-inlined **CCustomBlock** block has dependency on other shared libraries, then the location of these libraries must be provided. These libraries may include both external dll files that must accompany the model and **Twin Activate** dll files already available in the software distribution. In particular if the **CCustomBlock** block is created using the **Twin Activate** code generation facility, the libraries associated with the blocks used in the diagram are included in the list of libraries.

Bibliography

[1] Nicolas Devillard, Fast median search: an ANSI C implementation, 1998. http://ndevilla.free.fr/median/median/.

Chapter 4

Simulation restarts: implementing iterations

In some applications, a single time simulation does not provide a solution to the problem. This is particularly the case where multiple simulations are needed to solve problems such as parameter fitting, solving nonlinear systems, optimizing, etc. Time simulation iterations in **Twin Activate** are implemented through the **End** block, which can either stop the simulation for good, or stop and restart a new time simulation. These actions are controlled by activation signals in **Twin Activate**.

Activation signals represent the key element in **Twin Activate** formalism. They determine in particular when, how and in what order the block simulation functions are called during simulation. The ability to explicitly manipulate Activation signals, gives users modeling capabilities that are seldom present in other modeling and simulation tools.

The use of Activation signals go beyond the framework of a single simulation run. Within the formalism of **Twin Activate**, there exists the possibility of creating models where their simulations lead to multiple time simulations. Indeed in **Twin Activate** an activation signal can end or restart a time simulation. This ability to conditionally restart simulations is particularly useful for parameter optimization, strategy assessment, Monte Carlo simulation, etc. Traditionally, such applications are developed using scripts calling the simulator in batch mode. By integrating the complete formulation within the simulation model (which actually is a misnomer since the model contains more than a simulation model), the implementation becomes more concise and easy to maintain. There is of course always the possibility to use **OML** scripts to code the optimization part and run **Twin Activate** simulations in batch mode.

To illustrate possible applications of conditional restart in **Twin Activate**, an example is presented here. In particular the implementation of the shooting method for solving boundary value systems is considered. Boundary value systems are for example obtained when solving trajectory optimization problems. A boundary value system is often a system of ODEs defined over a time interval $[T_i, T_f]$ where the states of the system are not fully specified at the initial time of the interval. The problem is well posed thanks to conditions specified on the states at the final time of the interval.

The simplest implementation of the shooting method for solving boundary value systems consists of guessing values for the missing states of the system at initial time, simulating the system up to the final time, comparing the final states with the constraints imposed on them and based on this comparison generate new guess values for the initial states. The iterations are continued until the final states satisfy the constraints up to desired precision.

To implement the shooting method in **Twin Activate**, it is then necessary to model the system and implement a mechanism to compute new guess values and, until the desired precision is not achieved, to restart the simulation after changing the initial conditions. The system modeling is done as usual without any special consideration for the shooting method. The system model should provide a function of the final state of the system representing the constraints (the value of the function is zero if the constraints are satisfied).

For well-posed boundary value problems, the number of free initial states is equal to the number of final constraints. Let p denote the subset of initial state which are free and f the final constraint function, i.e.,

$$f(x(T_f)) = 0 \tag{4.1}$$

where x represents the states of the system. The final state $x(T_f)$ depends of course on the initial state $x(T_i)$ and in particular on p: $x(T_f) = H(T_f, p)$. The function H can only be evaluated numerically by simulating the system from T_i to T_f. For a given T_f, the constraint (4.1) can be expressed as

$$f(x(T_f)) = f(H(T_f, p)) = g(p) = 0.$$

The solution p of the nonlinear system $g(p) = 0$, which has as many equations as unknowns, can be obtained using various methods. For example the **OML** function `fsolve` is provided specifically for solving such problems. This function can be used in an **OML** script to solve the boundary value problem by running the simulation model programmatically. THe objective however here is to include the solution of the problem within the model, obtaining a self contained formulation of the problem in terms of a **Twin Activate** diagram without any dependency on external code. So a nonlinear solver similar to `fsolve` must be made available in **Twin Activate**, as a block.

4.1 Nonlinear solver block

A nonlinear solver block cannot function the way `fsolve` does. In `fsolve`, the nonlinear solver iterates by calling an external function provided by the user evaluating the function $g(p)$ for the given p until an end condition is satisfied, in which case the function returns the solution p, or an error message. `fsolve` is the main and the function evaluating $g(p)$ the secondary. This main-secondary configuration is not appropriate for implementing a solver block in **Twin Activate**. In **Twin Activate**, the **Twin Activate** simulator is the main solver and the blocks are secondary elements. For that, the solver algorithm must be implemented in reverse-communication mode. Many nonlinear solver and optimization codes are available both in normal and reverse-communication modes.

4.1.1 A reverse-communication solver

In reverse-communication mode, the solver is provided with the values of p and $g(p)$ and returns a new value of p indicating also whether this value is the solution or a value to be used for the next iteration. If we make the assumption that the main solver calls back systematically the solver function with p provided by solver at the previous iteration, then it is fairly straightforward to reprogram any solver in reverse-communication mode. The structure of the program must be changed so that the internal state of the solver is maintained from one call to the next. The solver algorithm however is not modified.

If the solver function is not recalled with the value it returns, then specific reverse-communication algorithms are required. This is the case considered here for the implementation of a nonlinear solver block. The function used here is a simple nonlinear solver implemented as toy code to illustrate the

approach. The internal state of the function contains a number of previous $(p, g(p))$ pairs and a weight function associated with each one. At each iteration, the new pair $(p, g(p))$ replaces the "least interesting" (having the smallest weight) pair and the eights are updated. The weight associated to a pair $(p, g(p))$ depends on the norm of $g(p)$ and how long ago the pair has been received. The weight is reduced at every iteration.

Let p be close to the solution p^* satisfying $g(p^*) = 0$), then by neglecting higher order terms,

$$g(p) = -J(p - p^*) \tag{4.2}$$

where $J = \frac{\partial g}{\partial p}(p^*)$ is the Jacobian matrix. This relation is the basis of the Newton's method when J is known. Using this relation for the pairs $(p_i, g(p_i))$, $i = 1, \cdots, N$, yields

$$p^* = p_i - J^{-1}g(p_i), \quad i = 1, \cdots, N. \tag{4.3}$$

The unknowns are p^* and J so the system has enough equations if $N = n + 1$ where n is the size of p, and it is over-determined if $N > n + 1$. In this latter case, a least-squared problem is formulated as follows

$$p^* = p_i - J^{-1}g(p_i) + v_i, \quad i = 1, \cdots, N, \tag{4.4}$$

with a cost function to minimize:

$$\mathcal{J} = \sum_{i=1}^{N} w_i \|v_i\|^2. \tag{4.5}$$

w_i^{-1}'s are weights associated with each equation.

The set of equations (4.4) can be expressed in matrix form as follows

$$p^*T = P - J^{-1}G + V \tag{4.6}$$

where

$$T = \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix}, \quad P = \begin{bmatrix} p_1 & p_2 & \cdots & p_N \end{bmatrix}$$

and

$$V = \begin{bmatrix} v_1 & v_2 & \cdots & v_N \end{bmatrix}, \quad G = \begin{bmatrix} g(p_1) & g(p_2) & \cdots & g(p_N) \end{bmatrix}.$$

.

The value of interest here is p^*, so the matrix J can be removed from the system by multiplying (4.6) on the right by $G^\perp$, a maximum rank right-annihilator of G:

$$p^*TG^\perp = PG^\perp + VG^\perp \tag{4.7}$$

which can be expressed as

$$Ap^* = b + C\nu \tag{4.8}$$

where $A = (TG^\perp)^T \otimes I$,[1] b is the vector obtained by concatenating the columns of the matrix $PG^\perp$ and $C = (G^\perp)^T \otimes I$. The vector ν is obtained by concatenating the v_i's.

The cost $\mathcal{J}$ in (4.5) can be expressed as follows

$$\mathcal{J} = \nu^T W \nu \tag{4.9}$$

---

[1]The $\otimes$ operation represents the Kronecker product.

where

$$W = \begin{bmatrix} w_1 I & & & \\ & w_2 I & & \\ & & \ddots & \\ & & & w_N I \end{bmatrix}.$$

The solution p^* minimizing $\mathcal{J}$ in (4.9) subject to (4.8) satisfies

$$\begin{bmatrix} A & -C^T W^{-1} C \\ 0 & A^T \end{bmatrix} \begin{bmatrix} p^* \\ \lambda \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}$$

from which the solution can be explicitly obtained as follows

$$p^* = \begin{bmatrix} I & 0 \end{bmatrix} \begin{bmatrix} A & -C^T W^{-1} C \\ 0 & A^T \end{bmatrix}^{-1} \begin{bmatrix} b \\ 0 \end{bmatrix}.$$

This algorithm can be coded as an **OML** function as follows:

```
function [pnext,TT]=nsolve(p,gp,TT)
  n=length(p);N=size(TT,2);
  G=TT(1:n,:);P=TT(1+n:2*n,:);wi=TT(2*n+1,:);
  wi=wi*(1+1/n);
  [_x,idx]=max(wi);
  G(:,idx)=gp;P(:,idx)=p;wi(idx)=gp'*gp;
  [Q,R]=qr(G');r=rank(R);Gperp=Q*[zeros(N-r,r);eye(r)];
  At=ones(1,N)*Gperp;Bt=P*Gperp;Ct=Gperp;
  A=kron(At',eye(n));b=Bt(:);C=kron(Ct',eye(n));
  Wi=kron(diag(wi),eye(n));
  m=size(A,1);
  pnext=[eye(n),zeros(n,m)]*...
    inv([A,-C*Wi*C';zeros(n,n),A'])*[b;zeros(n,1)];
  if norm(p-pnext)<eps
    pnext=pnext+2*eps*rand(n,1);
  end
  TT=[G;P;wi];
end
```

The arguments of the function are p, the value of $g(p)$ and a matrix _TT used as as internal state of the function used to store the previous values of p, $g(p)$ and the associated weights (more specifically the inverse of the weight).

The function returns a new value of p to be used at the next iteration and the updated internal state matrix _TT. This matrix could be initialized and retained by the function itself (and not be part of its input output arguments) from one call to the next be defining it as persistent. In the way this function will be used in the construction of a **Twin Activate** block, it is easier to have the calling function keep the value of _TT.

The stopping tests based the value of $g(p)$ and the maximum number of iterations are not implemented in this function and are handled externally.

4.1.2 Twin Activate block implementation

The nonlinear solver block is implemented as a Super Block where the central piece is an **OmlCustomBlock** block implementing the solver function `nsolve` developed in the previous section. This block has 3 regular inputs and 2 regular outputs corresponding to the arguments of `nsolve`. It also has an activation input port for its activation.

The Super Block should also implement the logic to stop the iterations and propose a user friendly interface for the user. The block is shown in Fig. 4.1. The block is a Super Block, with 2 inputs and 3 outputs: the inputs correspond to the pairs $(p, g(p))$ and the outputs are the new value of p proposed for iteration, a stop signal set to 1 if no further iterations are required and finally an error output, set to 1 if the iterations should stop on an error (the number of iterations exceeding the maximum allowed).

The block is masked as shown in Fig. 4.2. The parameters `ftol, maxiter, sz` are used to parameterize the solver. The last parameter, `count`, may be made invisible; the value of the variable `__counter__` corresponds to the iteration number; the first iteration is numbered one.
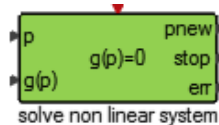
Figure 4.1: The new nonlinear solver block.
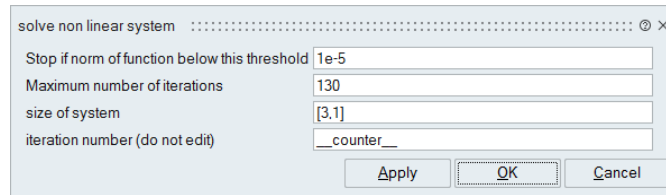
Figure 4.2: The mask of the Super Block.

The content of the Super Block is shown in Fig. 4.3. And its Context contains the following script

```
if count==1
  n=prod(sz);N=2*n;
  G=[eye(n,n),eye(n,n)];G=G(:,1:N);
  P=[eye(n,n),eye(n,n)];P=P(:,1:N);
  wi=ones(1,N)*1e32;
  TT=[G;P;wi];
else
  TT=GetFromBase('_TT',[]);
end
```

So at the first iteration, the value of `TT` is initialized in the Context. In the subsequent iterations, the value is read from the Base environment, in particular the variable `_TT`. This is done thanks to the **OML** function `GetFromeBase`. The second argument of this function is optional and is used only if the variable named in the first argument does not exist in the Base environment. In this case, the variable `_TT` exists at the second iteration because it has been defined by the block **ToBase** block in the diagram. Variables in the Base environment are used to hold and pass values from one time simulation to the next.

Note that the value of N is set to $2n$. The first tab of the parameter GUI of the **OmlCustomBlock** block is shown in Fig. 4.4.
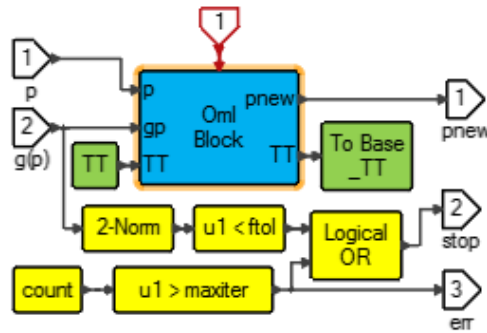
Figure 4.3: The content of the Super Block.
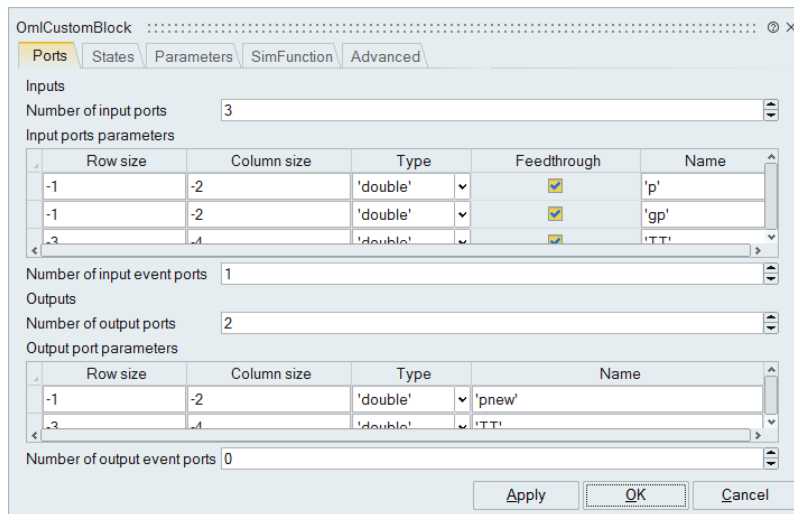
Figure 4.4: The first tab of the parameter GUI of the **OmlCustomBlock** block.

4.1.3 Examples

Static model

The model in Fig. 4.5 is a test example. There is no dynamical system and the simulation consists simply of evaluating a function at time zero, and the objective is to find the vector p for which this function is close enough to zero.

The initial guess value for p is defined in the context of the top diagram:

```
if __counter__==1
p=-ones(4,1);
else
p=GetFromBase('p');
end
```

The variable `__counter__` is provided by the system and represents the number of times the time simulation is performed. At the first iteration, the value of p is initialized in the Context. In the subsequent

iterations, the value is read from the Base environment, in particular the variable `p`. This is done thanks to the **OML** function `GetFromeBase`. The variable exists at the second iteration because it has been defined by the block **ToBase** in the model.

The time simulation restarts when the **End** block with option Restart is activated. This block is activated at the event at time zero only if the start signal has value zero. The simulation is no longer restarted if start has value 1, in which case depending on the value of the signal err, the simulation ends with a success message or an error message.

The use of Base variables and simulation restarts will be further explored when discussing optimization in the Chapter 6.

Shooting model

This example corresponds literally to a shooting problem. Projectile is fired with a given initial velocity V and the objective is to find the firing angle so that the projectile lands at a distance d.

Denoting the position and the velocity of the projectile in 2D by $s = (x, y)$ and $v = (v_x, v_y)$, the equations of motion are given by

$$\dot{s} = v, \quad x(0) = 0,$$
$$\dot{v} = -\alpha(|v|^{\beta-1})v + \begin{bmatrix} 0 \\ -g \end{bmatrix}, \quad v(0) = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} V.$$

The problem is then finding the initial firing angle θ such that when y crosses zero, x is close to d. So the nonlinear function to solve is $f(\theta) = x - d$. The problem is formulated as a **Twin Activate** model in Fig. 4.6. The system parameters are defined in the Initialization script of the model:

```
a=.1;
b=2.5;
g=9.8;
V=1000;
d=9;
```

and the initial guess value of θ is defined in the Context:

```
if __counter__==1
p=pi/3;
else
p=GetFromBase('p');
end
```
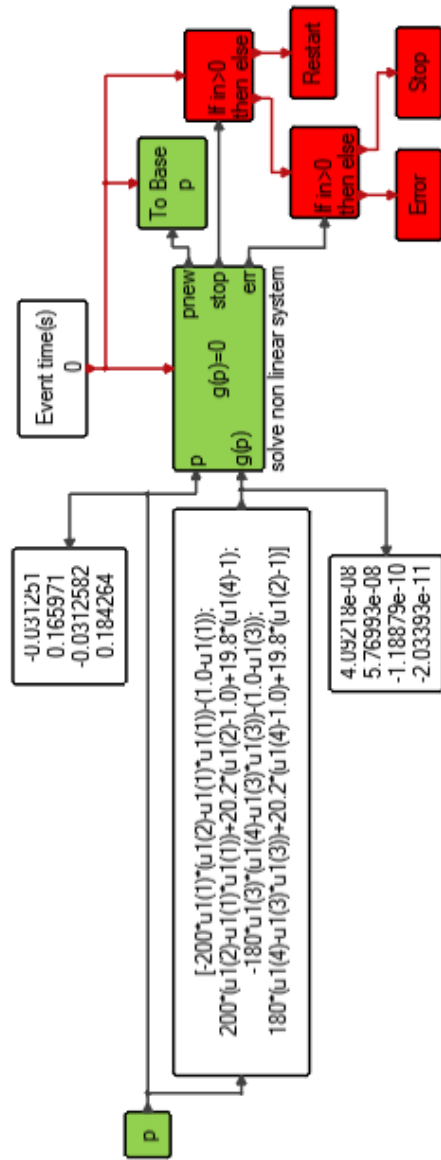
Figure 4.5: The **End** block is used in different configurations to restart the simulation, stop it with a success message and stop it with an error.
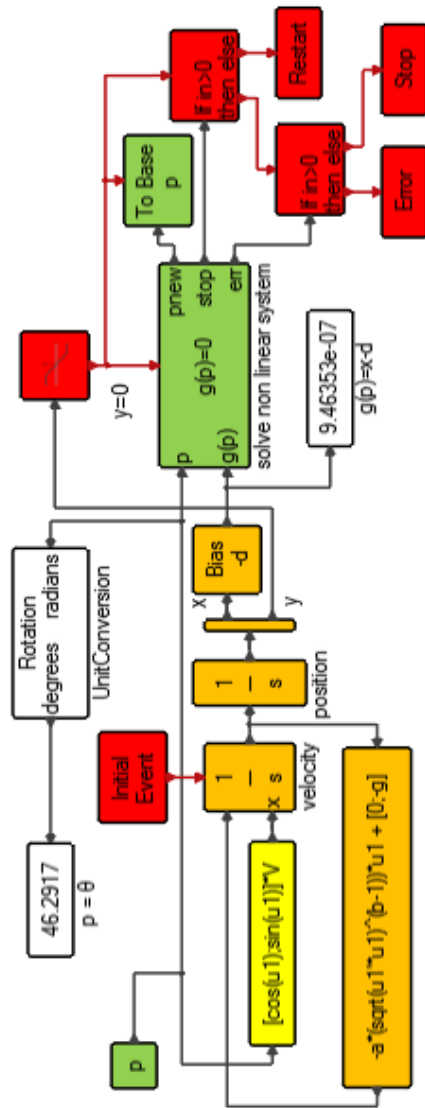
Figure 4.6: The implementation of the shooting method. In this case the final simulation time is not fixed and is determined by the time of the zero-crossing detected by the **ZeroCrossDown** block.

4.1.4 Solving nonlinear system without a specific block

The new nonlinear solver block is convenient to use for solving nonlinear systems in **Twin Activate** but the iterative method presented in the previous sections can be implemented without it. The function `nsolve` introduced earlier can be used directly in the Context of the diagram, as it is shown below. The Model in Fig. 4.7 is an alternative implementation of the model in Fig. 4.6. Assuming the file defining
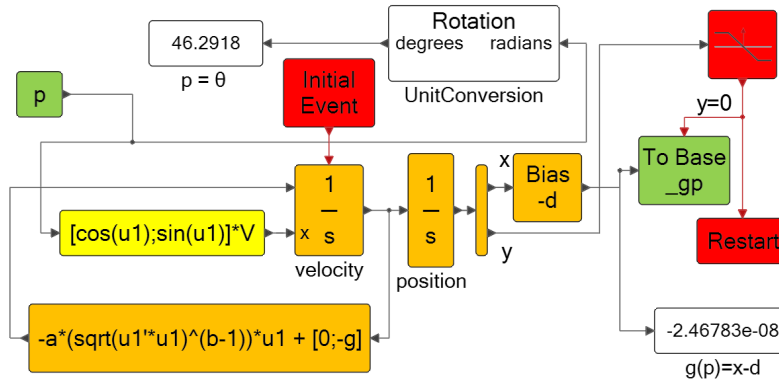
Figure 4.7: The implementation of the shooting method without using a specific block.

`nsolve` is in the path, the solver part is implemented in the Context of the diagram as follows:

```
if __counter__==1
  n=length(p0);N=2*n;
  G=[eye(n,n),eye(n,n)];G=G(:,1:N);
  P=[eye(n,n),eye(n,n)];P=P(:,1:N);
  wi=ones(1,N)*1e32;
  TT=[G;P;wi];
  p=p0;
else
  TT=GetFromBase('_TT',[]);
  p=GetFromBase('_p',[]);
  gp=GetFromBase('_gp',[]);
  if norm(gp)<1e-5
    printf('succesfull convergence, p=%g, g(p)=%g',p,gp);
    return
  end
  [p,TT]=nsolve(p,gp,TT);
  if __counter__>100
    error('Max nb. of iterations reached.');
  end
end
AddToBase('_p',p);
AddToBase('_TT',TT);
```

Example

The problem considered here is an optimal control problem where the objective is to find a function optimizing a cost function. The example consists of the computation of geodesics for the Grusin's

metric. See [1] for details. The problem considered here is the following:

$$J = \min \int_0^1 u_1^2 + u_2^2 \, dt$$

subject to

$$\dot{x}_1 = u_1 \tag{4.10}$$
$$\dot{x}_2 = u_2 x_1 \tag{4.11}$$

with boundary conditions $x_1(0) = x_2(0) = x_1(1) = 0$ and $x_2(1) = 1/(2\pi)$.

The Lagrangian is defined as follows

$$\mathcal{L} = \frac{1}{2}(u_1^2 + u_2^2) + \lambda(\dot{x}_1 - u_1) + \mu(\dot{x}_2 - u_2 x_1)$$

from which the following optimality conditions are obtained

$$\dot{\lambda} = -\mu^2 x_1 \tag{4.12}$$
$$\dot{\mu} = 0 \tag{4.13}$$

with optimal controls $u_1 = \lambda$ and $u_2 = \mu x_1$.

This system can be solved analytically but here it is used to illustrate the usage of the shooting method. Since μ is a constant, the unknowns are considered to be $p = (\lambda(0), \mu)$ and the system of equations to solve is

$$g(p) = \begin{bmatrix} x_1(1) \\ x_2(1) - 1/(2\pi) \end{bmatrix}$$

where $x_1(1)$ and $x_2(1)$ are obtained by integrating the system

$$\dot{x}_1 = \lambda$$
$$\dot{x}_2 = \mu x_1^2$$
$$\dot{\lambda} = -\mu^2 x_1$$

with initial conditions $x_1(0) = x_2(0) = 0$. These equations are obtained from (4.10), (4.11), (4.12) and using the optimal controls $u_1 = \lambda$ and $u_2 = \mu x_1$.

The solution is implemented as the **Twin Activate** model in The Initialization script defines the final conditions of x and an initial guess for the unknown p:

```
xT=[0;1/(2*pi)];
p0=[1;1];
```

The Context is the following script:

```
n=length(p0);N=2*n;
if __counter__==1
  done=false;
  G=[eye(n,n),eye(n,n),eye(n,n)];G=G(:,1:N);
  P=[eye(n,n),eye(n,n),eye(n,n)];P=P(:,1:N);
  wi=ones(1,N)*1e32;
```
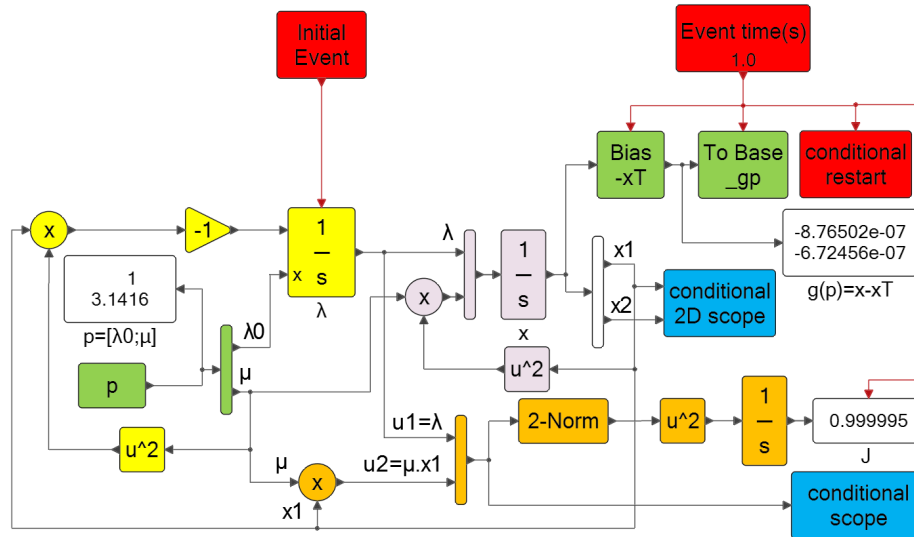
Figure 4.8: Model implementing the computation of geodesics for the Grusin's metric.

```
    TT=[G;P;wi];
    p=p0;
else
    TT=GetFromBase('_TT',[]);
    p=GetFromBase('_p',[]);
    gp=GetFromBase('_gp',[]);
    if norm(gp)<1e-5
        printf('succesfull convergence, p=%g, g(p)=%g',p,gp);
        done=true;
    elseif __counter__>100
        error('Max nb. of iterations reached.');
    else
        done=false;
        [pn,TT]=nsolve(p,gp,TT);
al=(__counter__<=N)*0.9;
p=al*p+(1-al)*pn;
    end
end
AddToBase('_p',p);
AddToBase('_TT',TT);
```

The simulation resutls are shown in Fig. 4.9 and 4.10.

4.2   Repeating activations

The **RepeatActivation** block in **Twin Activate** can be used to implement iterative computations at single event times. A simple example is presented to show its usage. In particular, the computation of the solution of a nonlinear equation is implemented using the Newton's method.
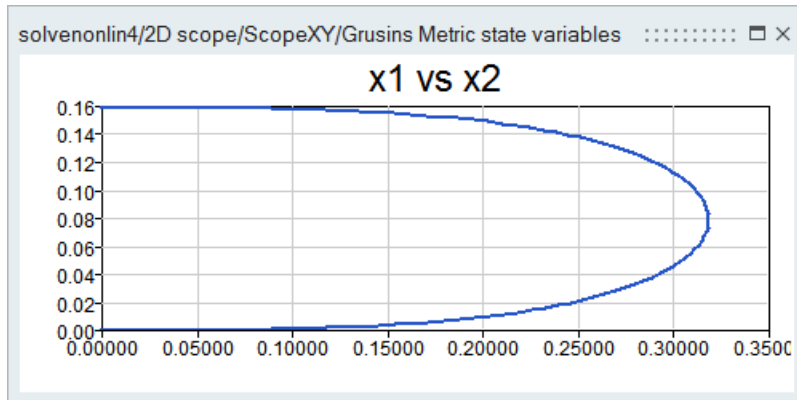
Figure 4.9: Optimal solution x_1 versus x_2.
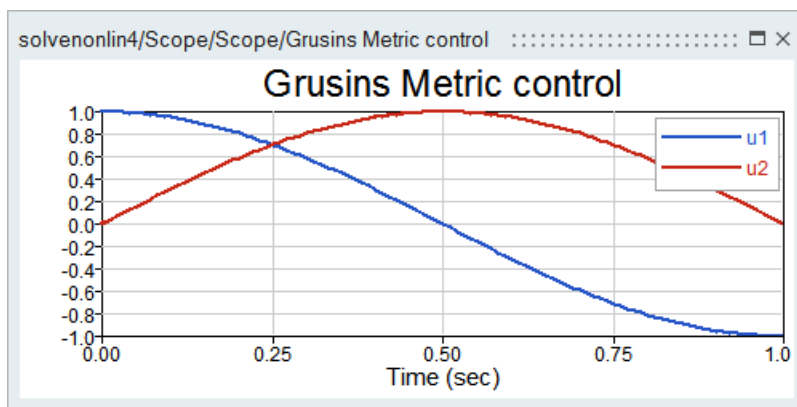
Figure 4.10: Optimal controls.

The **RepeatActivation** block plays also an important role in the implementation of state machines in **Twin Activate**.
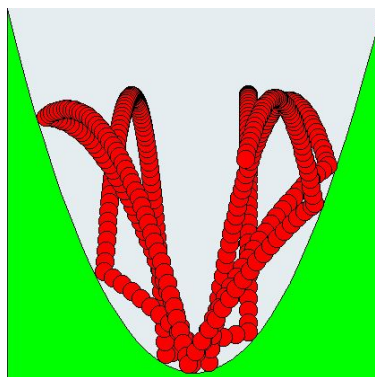
4.2.1 Problem setup

The problem considered here is a 2D version of a ball bouncing on a not-necessarily-flat surface. The surface is defined by a curve

$$y = f(x). \tag{4.14}$$

User defines the function f and its derivative f'. The ball is supposed to be a point-mass, and the collision with the surface elsastic.

The objective is to model and simulate this system in **Twin Activate**.

4.2.2 System equations

The state of the system contains the position (x, y) and the velocity (w, v) of the ball. The ball is subject to gravity, so its movement can be expressed as follows

$$
\begin{align}
\dot{x} &= w \tag{4.15a} \\
\dot{w} &= 0 \tag{4.15b} \\
\dot{y} &= v \tag{4.15c} \\
\dot{v} &= -g. \tag{4.15d}
\end{align}
$$

The collisions occur when (x, y) hits the curve, i.e, when (4.14) is satisfied.

The collision of the ball with the surface creates a jump in the velocity of the ball. The new velocity can be obtained using the law of conservation of momentum and the assumption of the conservation of energy (elastic collision), resulting in the following

$$
\begin{pmatrix} w \\ v \end{pmatrix} \leftarrow \begin{pmatrix} 1 & f'(x) \\ -f'(x) & 1 \end{pmatrix}^{-1} \begin{pmatrix} 1 & f'(x) \\ f'(x) & -1 \end{pmatrix} \begin{pmatrix} w \\ v \end{pmatrix}.
$$

For this particularly simple example, it is possible to explicitly solve the differential equations. Given the current position and velocities of the ball, its new position and velocity, t seconds in the future, assuming no collisions occur during this period, can be expressed as follows

$$
\begin{align}
w &\leftarrow w \\
v &\leftarrow v - gt \\
x &\leftarrow x + wt \\
y &\leftarrow y + vt - \frac{1}{2}gt^2.
\end{align}
$$

The time delay t is the time to the next collision. So t is the smallest positive value satisfying

$$
h(t) \triangleq y + vt - \frac{1}{2}gt^2 - f(x + wt) = 0. \tag{4.16}
$$

The solution to this nonlinear equation is obtained using Newton's iterations

$$
t_{k+1} = t_k - \frac{h(t_k)}{h'(t_k)}. \tag{4.17}
$$

The iterations are ended when $|h(t_k)|$ falls below a given threshold. The function $h'(t)$ can be expressed as follows

$$
h'(t) = v - gt - f'(x + wt)w. \tag{4.18}
$$
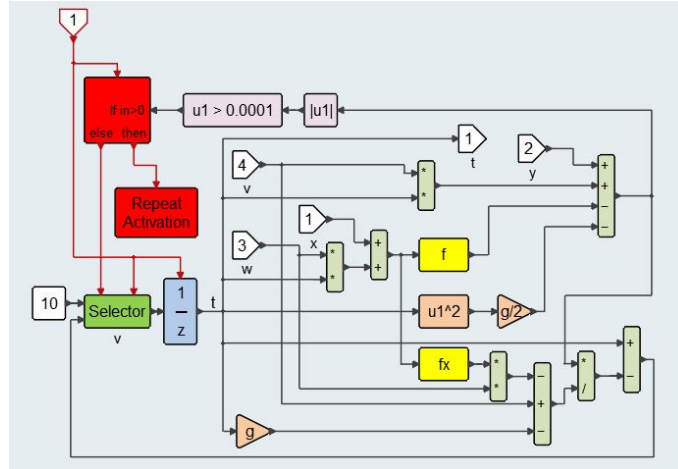
Figure 4.11: Atomic super block, `next time`, implementing the Newton iterations (4.17).

4.2.3 Implementation in Twin Activate

The implementation of the Newton iterations (4.17) is realized using a **RepeatActivation** block. When this block is activated by an event, once all the blocks activated by this event are processed, the same event is re-activated. This is done at the same synchronous time instant. For the rest of the model, the result of all the iterations are seen as a single event activation. The re-activated event is the primary event associated with the event that activated the **RepeatActivation** block. To confine the re-activation to the desired area of the model, it is convenient to use an Atomic super block. The implementation of the Newton iterations (4.17) can be seen in Figure 4.11.

The top diagram of the model `ball_bounce_.scm` is shown in Figure 4.12. For the sake of animation, the next event time is considered to be the minimum of the next collision time and a constant time step dt.

Note that the objective here has been to illustrate the use of the **RepeatActivation** block. The model is a naive implementation. For example the next collision time is computed at every dt; this can be optimized. And there is no guarantee that the Newton's iterations converge to a positive solution. So the simulation may fail for some surfaces $y = f(x)$.
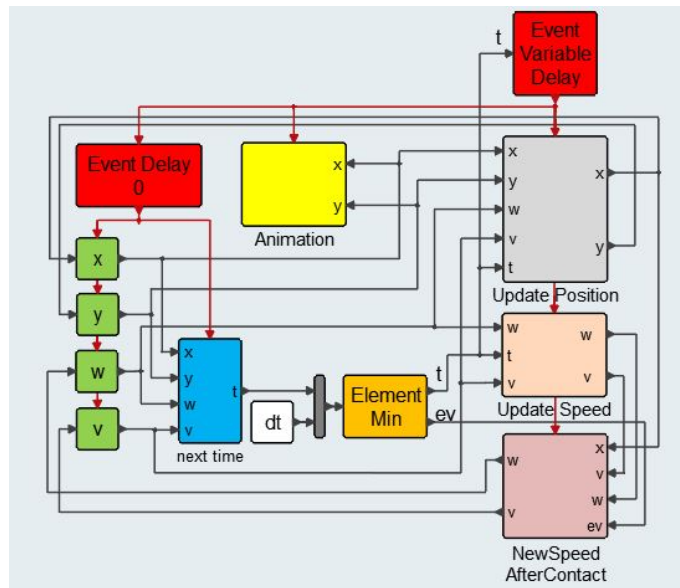
Figure 4.12: Top diagram of the model.

Bibliography

[1] Ugo Boscain and Benedetto Piccoli, "An introduction to optimal control" in Contrôle Non Linéaire et Applications, January 2005. http://www.cmapx.polytechnique.fr/~boscain/AUTOMATICS/introduction-to-optimal-control.pdf

Chapter 5

Example: Implementation of an Extended Kalman Filter in Twin Activate

Kalman filter is an algorithm that uses several measurements which are polluted by noise and produces estimates of the unknown signal. The estimated signal tends to be more precise than those based on a single measurement. The Kalman filter has numerous applications and is a widely applied concept in time series analysis used in fields such as signal processing, navigation and econometrics. The extended Kalman filter (EKF) is the nonlinear version of the Kalman filter which linearizes about an estimate of the current mean and covariance. In the case of well defined transition models, the extended Kalman filter is the de-facto method of state estimation for non-linear systems. This technique essentially linearizes a non-linear system at an operating point to be able to use the linear Kalman Filter methods of state estimation.

In this chapter, the implementation of an Extended Kalman Filter (EKF) in **Twin Activate** is illustrated through an example. The problem considered has been presented in [1]. It consists of constructing an estimator for the airspeed of a projectile using the measurements of its axial acceleration. The dynamics of the projectile is modeled as a system of nonlinear differential equations and noisy observations are supposed to be available periodically. The estimator is implemented using an EKF.

The model of the projectile is the following:

$$\begin{pmatrix} \dot{V} \\ \dot{z} \\ \dot{\theta} \end{pmatrix} = f \begin{pmatrix} V \\ z \\ \theta \end{pmatrix} = \begin{cases} \dfrac{1}{2m}\rho V^2 S_{\text{ref}} C_d - g\sin(\theta) \\ -V\sin(\theta) \\ -\dfrac{g}{V}\cos(\theta) \end{cases}$$

where the model states V, z and θ are air-relative speed, altitude, and angle of attack of the projectile respectively. The model parameters m, ρ, S_{ref}, C_d, and g are the projectile mass, the air density, the reference surface of the projectile, the drag coefficient, and the gravity acceleration respectively. The sensor provides the axial acceleration of the projectile. Thus the measurement equation used as the measurement equation of the Kalman filter is:

$$h(V, z) = a_x = \frac{1}{2m}\rho V^2 S_{\text{ref}} C_d.$$

The value of C_d is usually provided through a table as a function of the Mach number M. Here the Mach number is given by the formula $M = V/a$ where a is the speed of sound. The speed of sound is

a function of the air density ρ, which in turn is given by the following empirical equation given in [2]:

$$\rho(z) = (\rho_0 + 0.00651 \frac{z}{T_0})^{4.256} \qquad (5.1)$$

where ρ_0 and T_0 represent the air density and air temperature at see level.

5.1   Twin Activate model
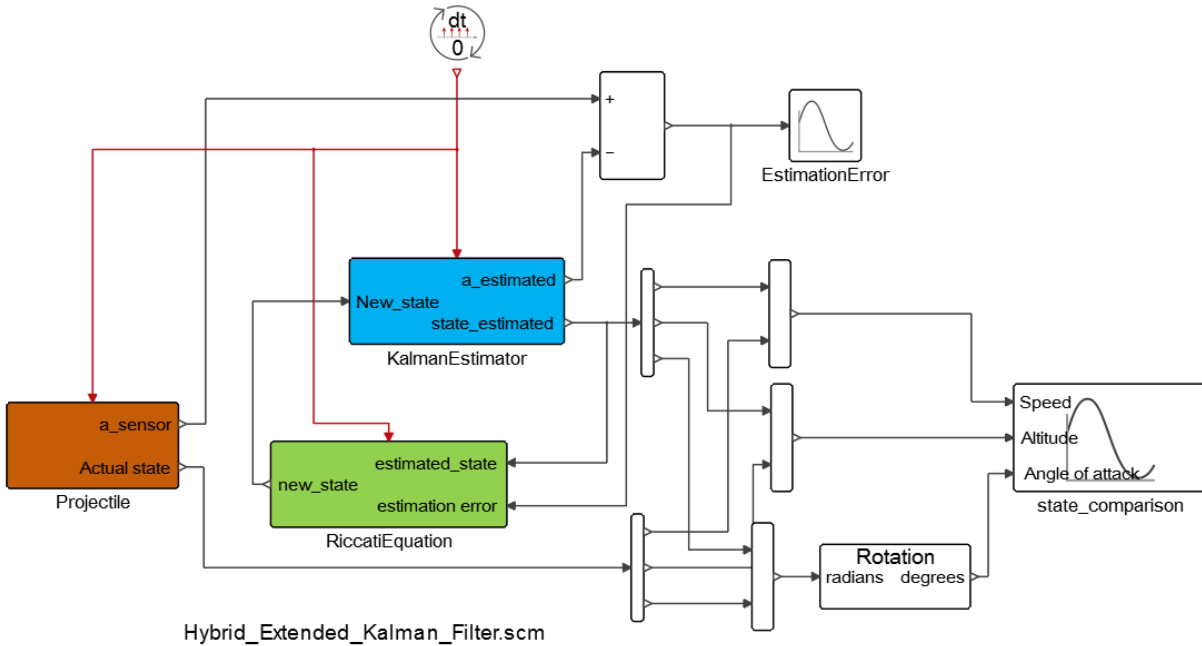
Figure 5.1: Projectile speed estimation model (Hybrid_Extended_Kalman_Filter.scm)

The above model is implemented in **Twin Activate** as illustrated in Figure 5.1. This diagram contains the whole system including the projectile model and the EKF. The parameters of the model are defined in the Context of the diagram as shown in Figure 5.2:

In the Context Editor of **Twin Activate**, the projectile parameters, the drag coefficient of the projectile as a function of the Mach number $C_d(M)$ as well as expressions for its derivative $\frac{dC_d}{dM}$ are given. In the projectile model, in order to compute the Mach number, the air density and the sound speed are first computed as a function of altitude and the projectile speed. These values are computed using two `MathExression` blocks. Once the Mach number is computed, a lookup table is used to find out the drag coefficient of the projectile. Two noise sources with Normal distribution are used to simulate the process and the measurement noises. The Covariance of theses sources are defined in the context editor.

5.2   EKF model

The Kalman filter is essentially a set of equations that implement an optimal predictor-corrector type estimator. The optimality is in the sense that it minimizes the estimated error covariance. Although
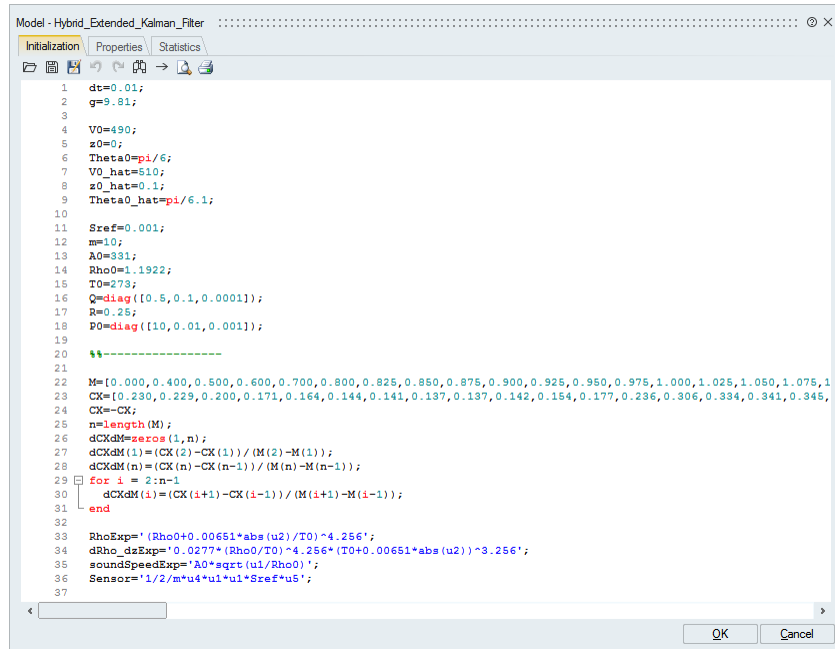
Figure 5.2: Parameter definition in the **Twin Activate** context editor

conditions to reach optimality rarely exist, the filter is used in many applications. The standard Kalman filter is limited to a linear system, the projectile model is nonlinear and the nonlinearity is associated with both the process model and the observation model. As a result we need to choose an extended Kalman filter for state estimation. Unlike standard Kalman filter which is discrete-time, this systems is represented as continuous-time models and discrete-time measurements are frequently taken for state estimation and state update. Therefore, we used a hybrid Extended Kalman filter where the state dynamics is represented by a nonlinear system and the state updated is done through a nonlinear discrete-time update. The system model and measurement model are given by

$$
\begin{cases}
\dot{\hat{x}}(t) & = f(\hat{x}, t) + w(t) \\
y(t_k) & = h(\hat{x}) + v(t_k)
\end{cases}
$$

where

$$
\hat{x} = \begin{pmatrix} \hat{V} \\ \hat{z} \\ \hat{\theta} \end{pmatrix}
$$

$\hat{x}$ is the Kalman filter estimation for actual process state x. In the Figure 5.4, the Kalman filter has been implemented in **Twin Activate**. The estimator model is very similar to the projectile model with the difference that the state of the integral block is updated at each new sensor reading. The Kalman estimator equation itself is integrated using a numerical solver. In **Twin Activate**, any numerical solver available in the solver pane can be used. The solver integrates between each two measurements. At the end of each integration, the predicated state is used for computing estimated error.
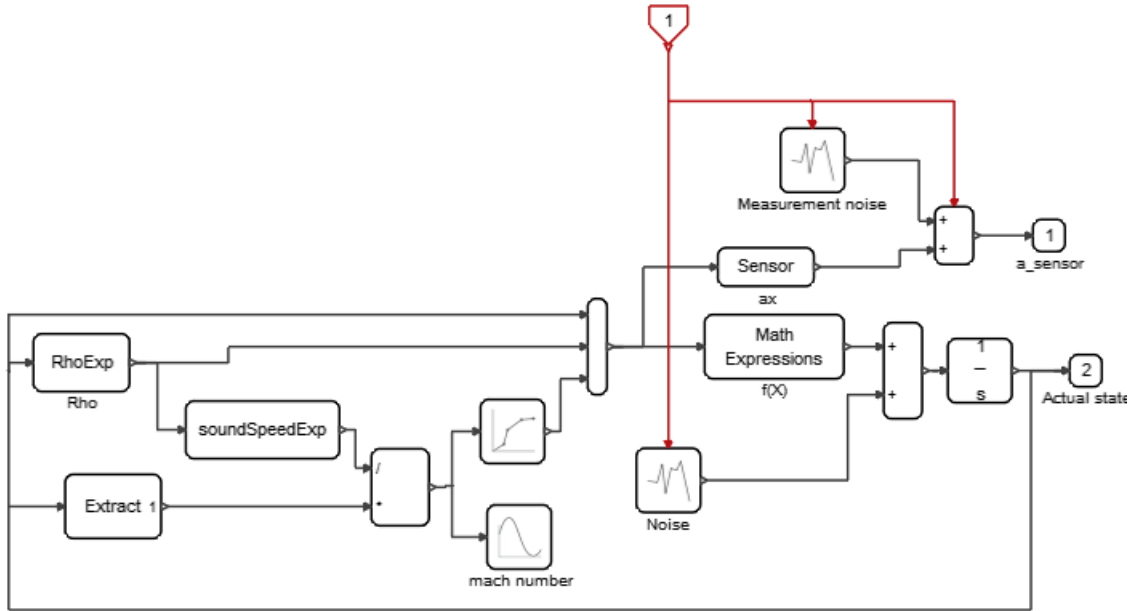
Figure 5.3: Model of the projectile

5.3 Covariance prediction

In Kalman filter the error covariance matrix (P) containing the filter's estimated error covariance for the Kalman states ($\hat{x}$). The following relationship defines the way this matrix is evaluated as a function of the system Jacobian (A) and process noise (Q).

$$\dot{P} = AP + PA^T + Q$$

The system process noise is a way to tell the filter how much the system dynamic model should be trusted. The system Jacobian matrix A is the partial derivative of $f(x)$ with respect to x. In computing the matrix $A(x)$, we need to compute $\dfrac{\partial C_d}{\partial z}$ and $\dfrac{\partial C_d}{\partial V}$. These two values can be computed as follows:

$$\begin{cases} \dfrac{\partial C_d}{\partial V} &= \dfrac{\partial C_d}{\partial M}\dfrac{\partial M}{\partial V} \\[2mm] \dfrac{\partial C_d}{\partial z} &= \dfrac{\partial C_d}{\partial M}\dfrac{\partial M}{\partial z} \end{cases}$$

$\dfrac{\partial C_d}{\partial M}$ is a table as a function of M whose values are be computed by brute force differentiating of the table $C_X(M)$. The value of $\dfrac{\partial M}{\partial V}$ is simply $\dfrac{1}{a}$, where a is the speed of the sound. The value of $\dfrac{\partial M}{\partial z}$ is however more tricky to compute. The sound speed (a) as a function of air density (ρ), see level air-speed (a_0), and see level air density (ρ_0) is given by

$$a = a_0\sqrt{\dfrac{\rho}{\rho_0}}$$

Thus, we can get

$$\frac{\partial M}{\partial z} = -\frac{V}{a^2}\frac{\partial a}{\partial z} = -\frac{V a_0}{2a^2\sqrt{\rho\rho_0}}\frac{\partial \rho}{\partial z}$$

The value of $\dfrac{\partial \rho}{\partial z}$ can be obtained by differentiating equation (5.1) with respect to z. The computation of these values in **Twin Activate** are illustrated in Figure 5.6.
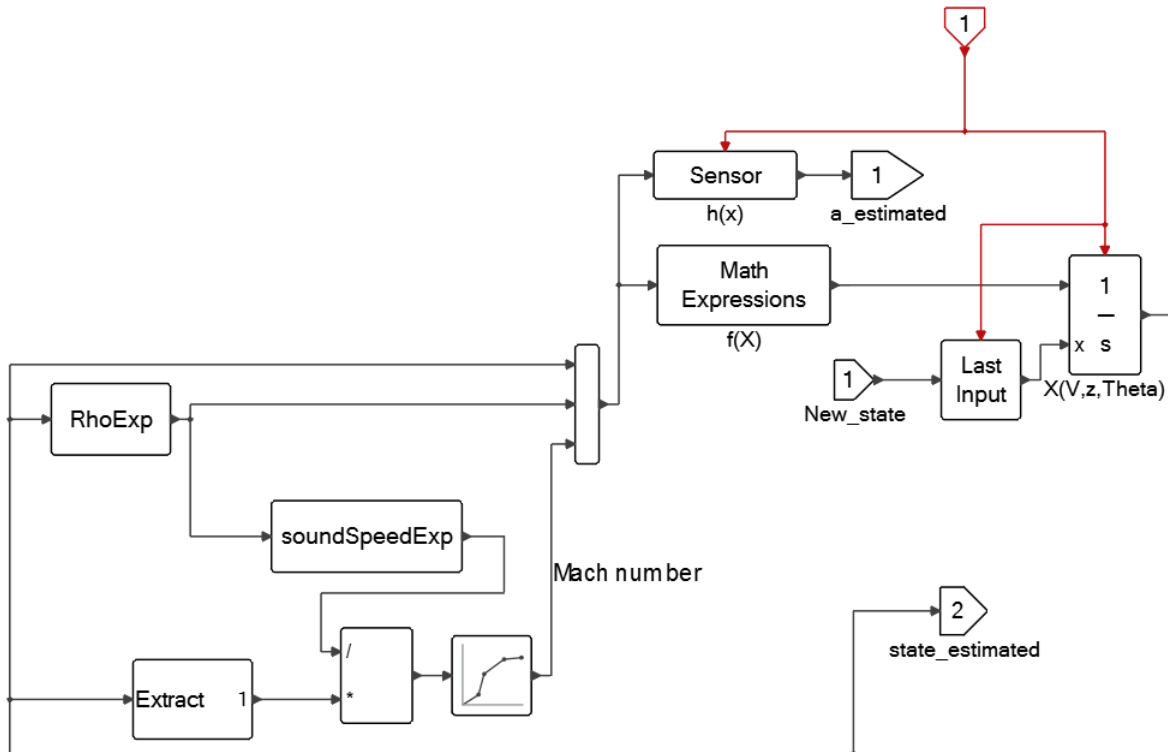
Figure 5.4: Kalman estimator

5.4 State update

At each sensor observation (an event activation in **Twin Activate**), the `a posteriori` Kalman state and the error covariance state are updated with the following relation:

$$\begin{cases} \hat{x}^+ &=& \hat{x}^- + K(a_{\mathrm{measurement}} - h(\hat{x}^-)) \\ P^+ &=& (I - KH)P^- \end{cases}$$

where $a_{\mathrm{measurement}}$ is the measured value at t_k. The gain K is computed as follows:

$$K = P^- H^T (H P^- H^T + R)^{-1}$$

where $H = \dfrac{\partial h(x)}{\partial x}$. In Figure 5.5, the implementation of the error covariance matrix and the state update computation in **Twin Activate** are illustrated.
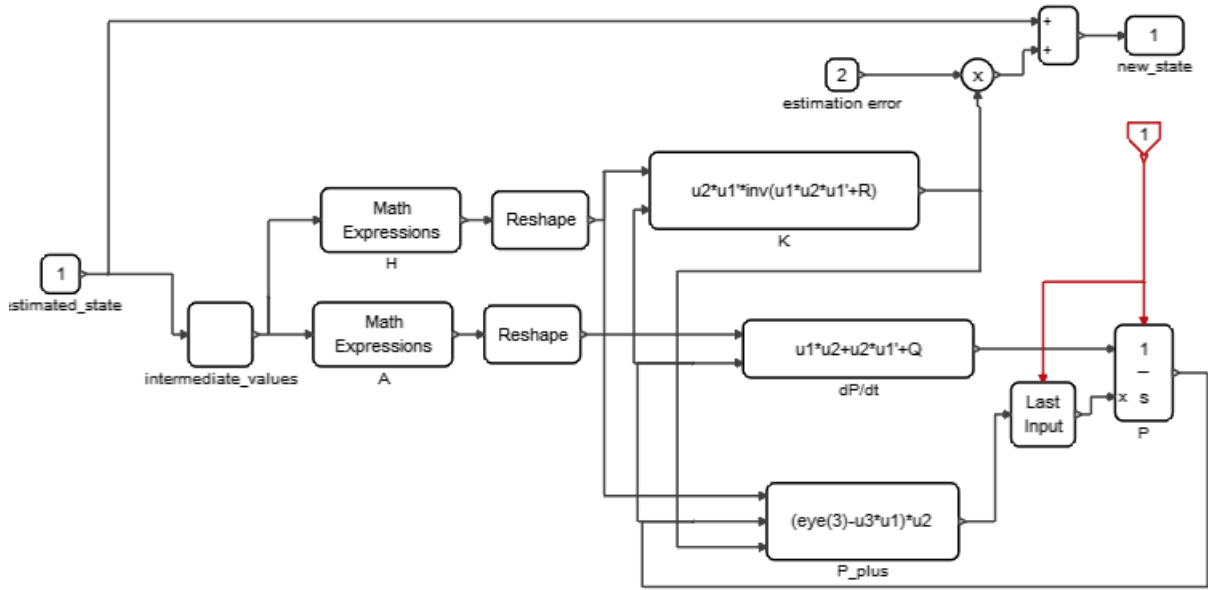
Figure 5.5: Kalman state and error covariance state Update

5.5 Simulation

In Figure the actual process states (in red) as well as the Kalman estimation of the states are shown. In Figure , the Kalman estimation error is shown.
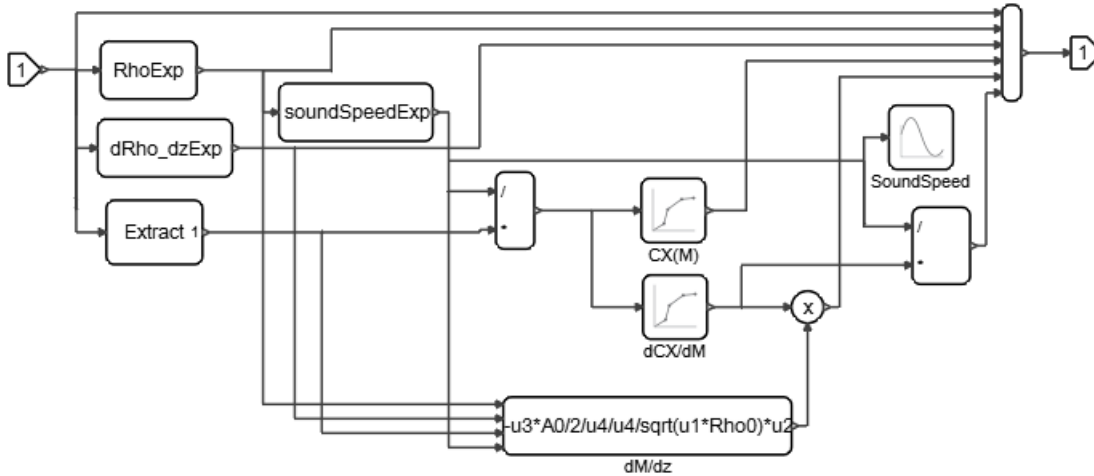
Figure 5.6: Computing required partial derivatives
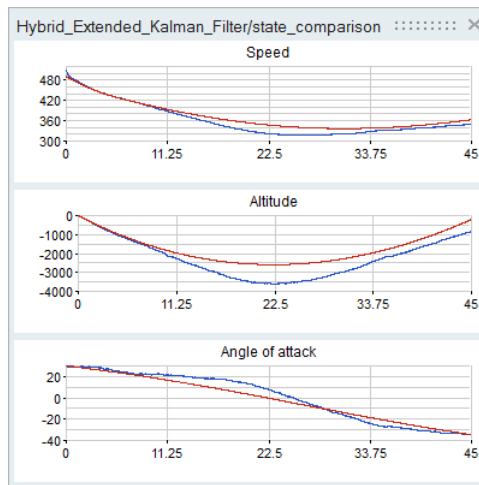
Figure 5.7: Kalman state and error covariance state Update
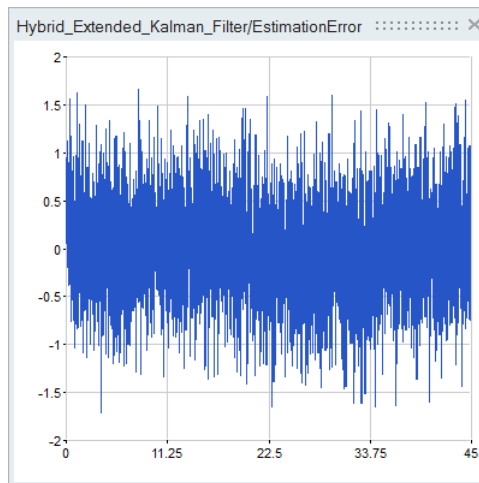
Figure 5.8: Kalman state and error covariance state Update

Bibliography

[1] Thomas Recchia, Projectile velocity estimation using aerodynamics and accelerometer measurements: a kalman filter approach, US Army research center, New Jersey, 2010.

[2] Barnes W. McCormick, Aerodynamics Aeronautics and Flight Mechanics. 2nd ed., John Wiley & Sons, Inc., New York, NY, 1995.

Chapter 6

Optimization

In many applications, modeling and simulation are used to find optimal values for system parameters or optimal control strategies. This can be done by formulating an optimization problem with proper cost function associated with the simulation model. There are various ways to define and solve such optimization problems in **Twin Activate**. This document describes some of the available methods.

The outline of this chapter is as follows: first the **Optimization** block is presented. This block may be used to formulate optimization problems fully integrated within **Twin Activate** models. Then batch simulation from the **OML** environment is presented and its application to optimization is illustrated via several examples. Finally, a graphical tool specifically developed for optimization in **Twin Activate** is presented. The graphical tool is probably the simplest way to formulate and solve optimization problems in **Twin Activate** but understanding its operations requires some knowledge of batch simulation that is why it is presented last.

6.1   Use of Optimization block in Twin Activate

The **Optimization** (**BobyqaOpt**) block, available in the Optimization palette, may be used to solve optimization problems using the BOBYQA software written by Prof. Powell, implemented in reverse communication mode.
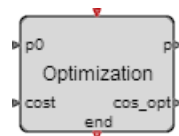
Figure 6.1: BOBYQA block

Unlike the usual optimization procedures where the optimization code is the main code calling the simulator for cost evaluation by providing the value of the variable, in reverse communication mode the simulator runs the main program and calls the optimizer with the cost value. The optimizer then returns a new value of the variable indicating whether the optimization should be continued or that this value corresponds to the optimal value of the variable.

The **Optimization** block parameters are the optimization parameters and the inputs and outputs of the block are used to exchange variable values (vectors in general), cost and control signals between the optimizer and the simulator. The block restarts the simulation as long as necessary until the optimal

solution is found.

This method requires a way to maintain the value of the optimization variable from one simulation run to the next. This may be done using the **ToBase** and **FromBase** blocks. The **ToBase** block, when activated, copies the value of its input into a variable in the base **OML** workspace. The name of the variable is given by the block parameter. The **FromBase** block, when activated does the opposite: it copies the value of the variable to its output. If the variable name specified by the **FromBase** block parameter does not exist in the base **OML** workspace, a default value is copied.

The first input of the **Optimization** block is used to pass the initial/current value of the variable to the optimizer. The second input is used to pass the cost associated with the value produced by the **Optimization** block on its first output in the previous simulation run. The block is activated when a new value of the cost is available, usually at the end of every simulation run. The block then produces the optimization variable on its first output, and either restarts the simulation or generates an event indicating that the optimization has ended and that the variable provided on the output is the solution to the optimization problem. In general the activation output is connected to an **End** block with parameter "Stop" but other configurations are possible. The optimal cost is provided by the second output of the block.

6.2 Example

The following is a simple example to illustrate the use of the **Optimization** block. The problem considered is that of finding the optimal gear ratios for a 5 speed vehicle given its torque curve as a function the engine RPM. This should be considered a toy example: the model is very simple and in particular does not consider the clutch mechanism. Multiple optimization criteria will be examined.

The simple model used for the dynamics of the vehicle is given by the following equations:

$$
\begin{aligned}
\dot{\omega} &= g_i^2 T(\omega)q/c - a\omega - bv^2 \\
v &= c\omega/g_i \\
\dot{x} &= v.
\end{aligned}
$$

The variable ω designates the engine angular velocity, g_i is the gear ratio for gear number i, v is the speed of the vehicle and x the traveled distance. $T(\omega)$ is the torque as a function of the angular velocity, and a and b are constant friction coefficients. q and c are also constant coefficients depending on vehicle characteristics. At the time of gear shift, ω jumps so that v remains a continuous-time function. The system is modeled as a Super Block named car the content of which is illustrated in Fig. 6.2.

The 5 activation links entering the diagram on the top are used to model the shifting events; the Selector block outputs the corresponding gear ratio. The Torque curve is modeled using a Lookup table block.

The first optimization problem consists of finding the maximum traveled distance in 30 seconds starting with a given small initial speed. The optimization variables are the gear ratios and the shift times. Even though given the gear ratios it is possible to find the up-shift times without optimization in this simple case, we consider them as optimization variables for simplicity. So there are 9 optimization variables: a vector T of size 4 representing the time lapses from one up-shift to the next, and a vector gear of size 5 that represents the gear ratios associated with the gearbox.

The **Optimization** block is used with the configuration described in Fig. 6.3.

At $t = 30$, the **Optimization** block is activated. When this happens, the **Optimization** block reads the cost value and generates a new optimization vector p. Vector p is the concatenation of T and

`gear`. The values of `T` and gear are saved in the base environment from one simulation run to the next. **SetSignal** blocks are used to make their values available in the rest of the diagram without having to draw long links through the model.

The parameters of the **Optimization** block are given below (see Fig. 6.4). Note that the bounds are set very large but the lower bounds guarantee that the variables cannot take zero or negative values. The model is not valid if they do.

The shift times vector `T` generates events using the Event Variable Delay block. The Event Variable Delay block delays its input activation by the value given at its regular input port at the time of activation. Note that time values given in `T` represent incremental times, i.e., the time from one gear shift to the next, not absolute shift times.

The usage of Event Variable Delay block may seem unnecessary since the amount of delay does not change during a single simulation run. In such cases, the Event Delay block is commonly used where its block parameter is set to the desired delay. But since in this case the amount of delay comes from the base, the Event Variable Delay block is used and the value of delay is obtained from the base using a **FromBase** block. This is not however the only possible implementation. Later, an alternative method for obtaining variables from the base will be presented allowing the usage of the Event Delay block, simplify considerably this diagram.

The model parameters are defined in the Initialization script of the diagram:

```
v0=10/3.6;
a=.004;
b=0.005^2;
c=.05;
q=.003;
Torque=[0,600,800,1000,2000,3000,4000,5000,6000,8000;
        0,10,40,100,200,280,360,350,200,0];
```

Running the simulation actually starts the optimization process. The optimum gear ratios and the maximum traveled distance are seen through the use of **Display** blocks in the model. The traveled distance and speed as a function of time are also given below along with trajectory followed on the Power/RPM curve which shows, as expected, that the optimal gear shifting times correspond to the strategy where the engine RPM is beyond the maximum power RPM and the up-shift reduces the engine RPM to a point that has the same power output as before the shift (Fig. 6.5):
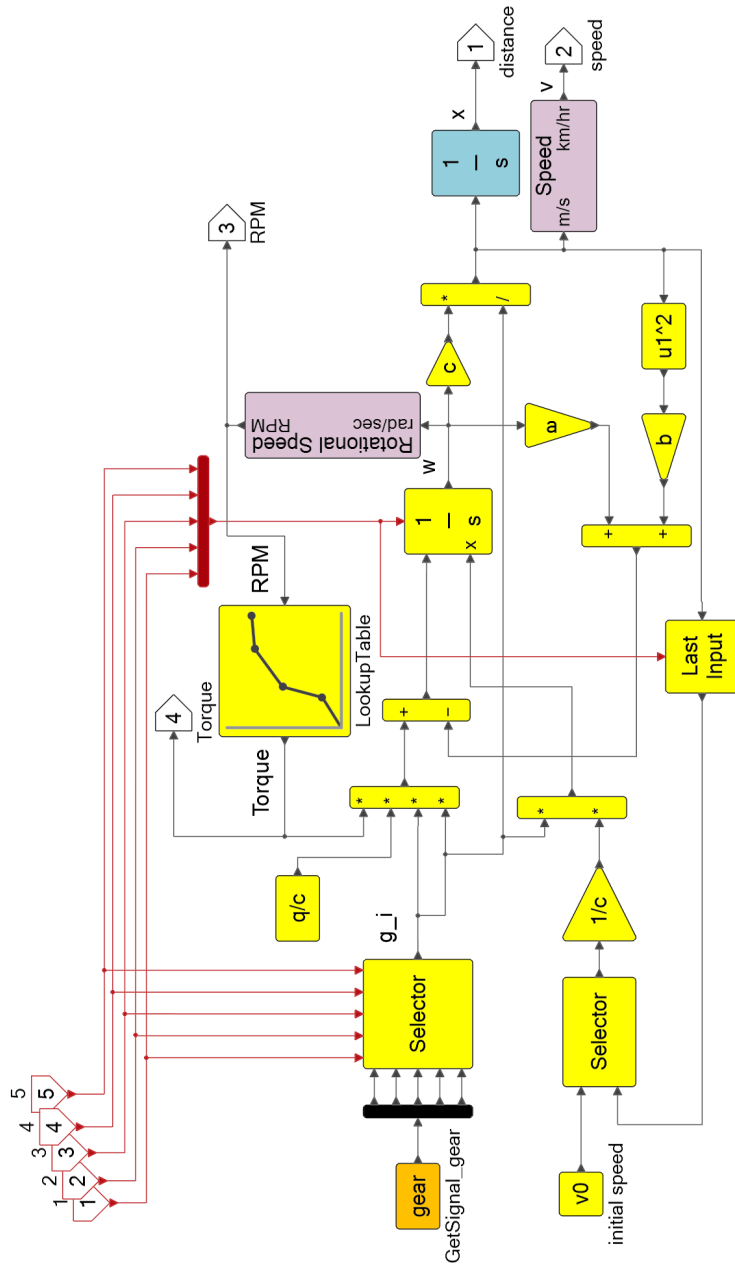
Figure 6.2: Superblock in Car model
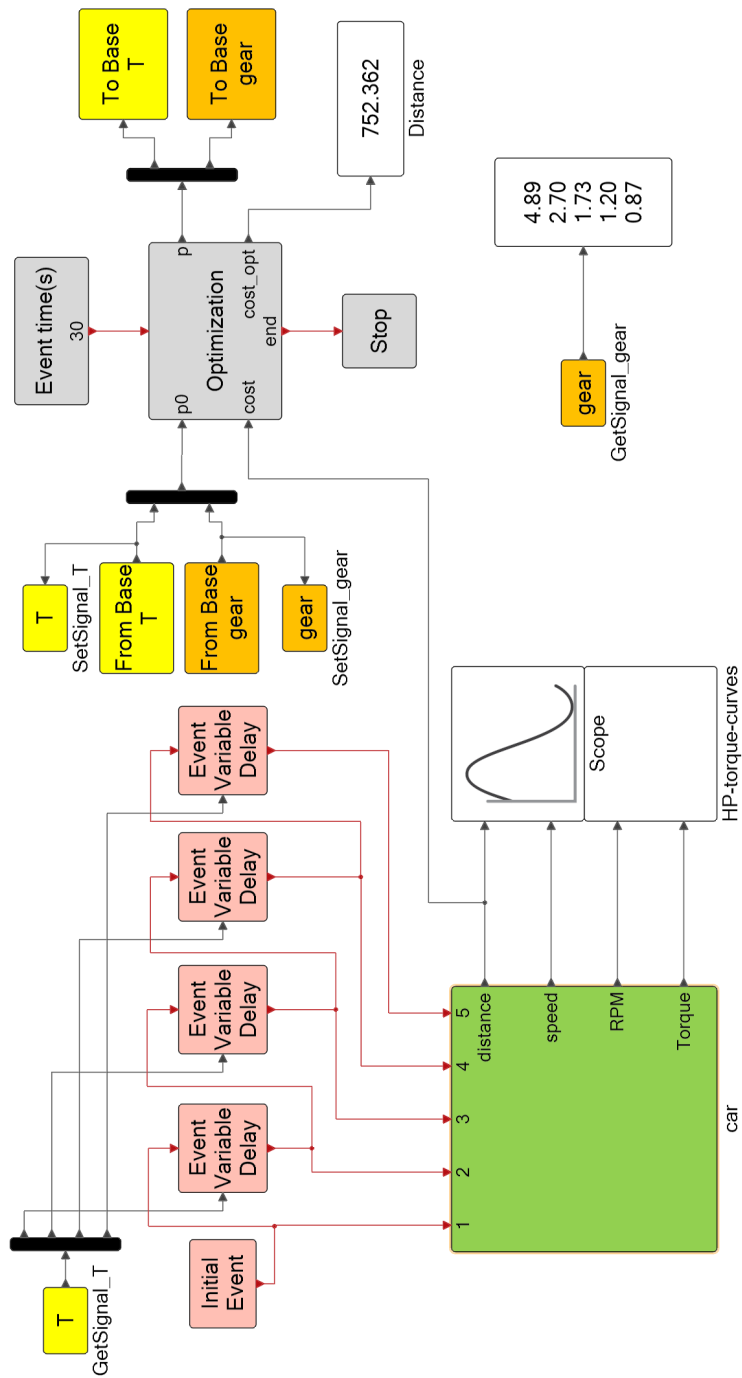
Figure 6.3: Optimization block used in Car model
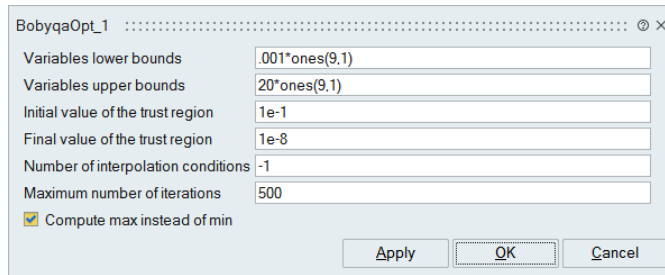
Figure 6.4: BobyQA block dialog
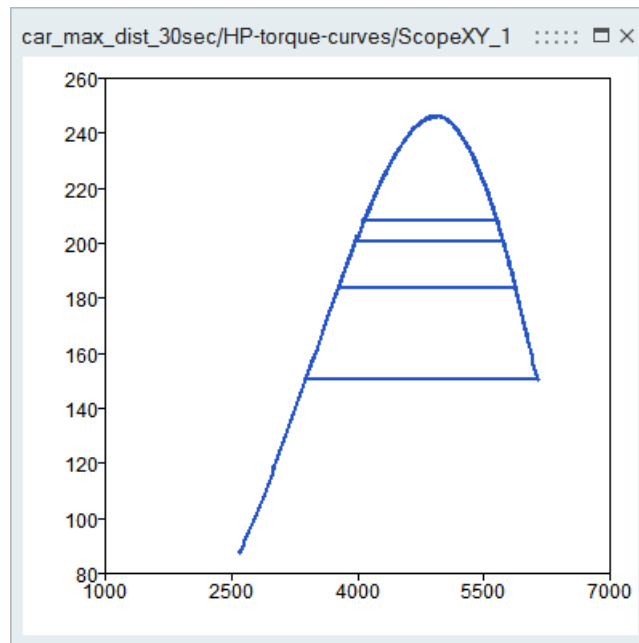
Figure 6.5: Optimal path illustrated on the curve representing the engine power versus RPM
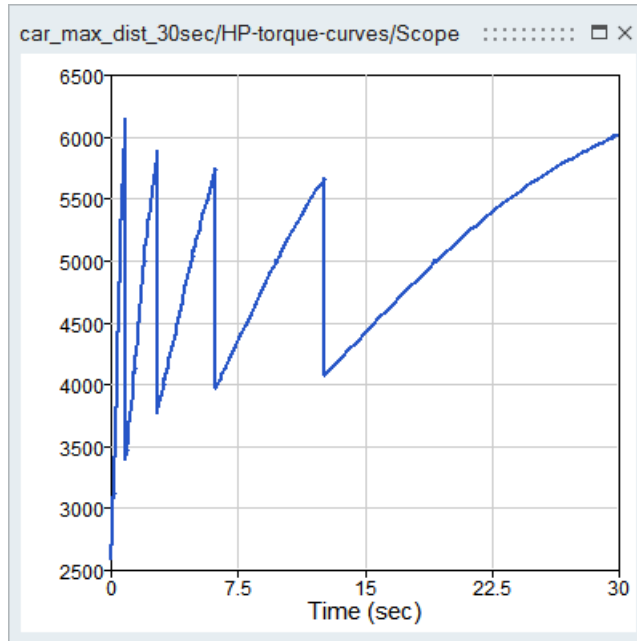
Figure 6.6: The RPM as a function time.
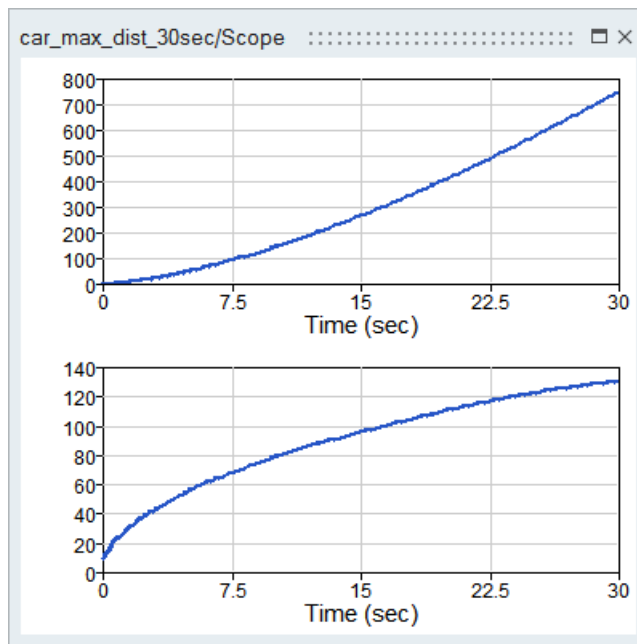
Figure 6.7: Distance and Speed results for the model car_max_dist_30sec.scm

The second optimization problem consists of minimizing the time to reach 1000m. In this case each simulation run is ended when the traveled distance goes beyond 1000m. The zero-crossing block **ZeroCrossUp** generates an event when this happens, activating the **Optimization** block, which reads the cost value (in this case the time) and generates a new optimization vector `p`. Vector `p` is again the concatenation of `T` and `gear`. The full model is shown in Fig. 6.8.

The optimization result shows that the vehicle reaches 1000m in 36.5 seconds. The optimal gear ratios are different from the solution of the first problem.

In another problem formulation, reaching a given speed in the shortest possible time is considered. This problem is similar to the previous problem and the corresponding model is found in the file `car_min_time_speed_vtarget.scm` and is illustrated in Fig. 6.9.

Note that an event at time 100 (final simulation time) is added to the event corresponding to reaching `vtarget` since depending on the gear ratios, the car may never reach the speed of `vtarget`. This problem is considered again later to illustrate the use of an **OML** function to read data from the base workspace. Using this function, leads to simpler models as explained in the following section.

Optimal gear-change timing

It is interesting to note that the optimal gear chaning instants are the time instants where before and after the gear change, the engine power has the same value, i.e., the transition is horizonal (see Fig. 6.5). This of course is not a coincidence and can be shown to hold under very general assumptions, as it is shown below. The solution found by the optimizer having this property is an indication that the optimization process has succeeded in finding the optimal solution.

To show that the optimal solution corresponds to a horizontal transition, consider the simpler problem where the car has only two gears, so the problem is to find the timing of the only possible gear change. The extension to the case of more than two gears is then straighforward.

The following image compares two gear shifting strategies corresponding to different gear changing times:
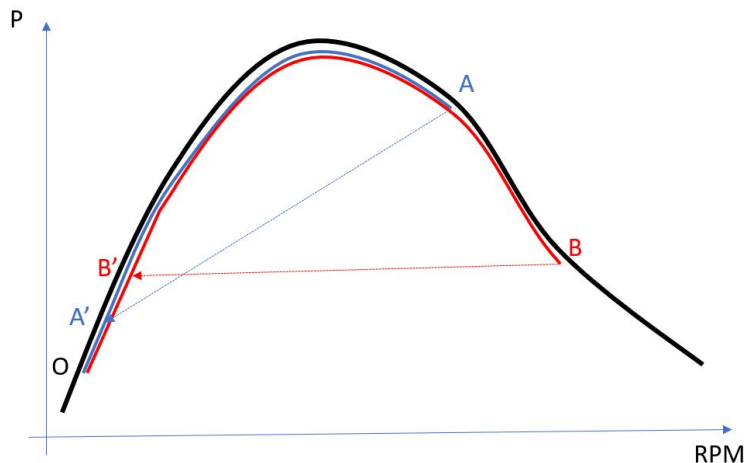
For both strategies (blue and red) the starting point is O. As time goes on, the RPM goes up with the power following the black curve (representing the maximum engine horse power as a function of RPM). For the blue strategy the gear is shifted at point A transitioning the state of the system to point A'. For the red strategy, the gear shifting happens later, at point B, transitioning to B'. So both strategies follow the exact same path except over the intervals $[A, B]$ and $[A', B']$: in the red strategy, the path includes

$[A, B]$, whereas the blue strategy includes $[A', B']$.

To compare the two strategies, the time it takes to go from A to B must be compared to the time to go from A' to B'. In both cases the starting and ending speeds are the same, so the time is shorter if the power over the time interval is higher. In the image above, the part of the black curve from A to B is clearly higher than that between A' and B'. So more power is supplied over $[A, B]$ compared to $[A', B']$, which means that the time for going from A to B is shorter. Thus the time for the red strategy is smaller. Now by considering the limiting case as A gets closer to B (and so A' to B'), it is easy to see that the optimal transition occurs when A, B, A' and B' are aligned.

6.3 `GetFromeBase` and `AddToBase` **OML functions**

The usage of the **Optimization** block requires a method for keeping the values of the optimization variables from one simulation run to the next. The **FromBase** and **ToBase** blocks were used to communicate with the base. The use of **FromBase** however implies that the optimization variables can only be used as input to other blocks and not as block parameter values. To be able to use optimization variables in the definition of block parameter values, the **OML** function **GetFromBase** may be used either directly in the definition of parameters or, in the Initialization script or a Context.

To illustrate the use of this function consider again the problem of reaching `vtarget` in minimum time but in this case the values of `T` and `gear` are retrieved from the base in the Initialization script:

```
v0=10/3.6;
a=.004;
b=0.005^2;
c=.05;
q=.003;
Torque=[0,600,800,1000,2000,3000,4000,5000,6000,8000;
        0,10,40,100,200,280,360,350,200,0];
vtarget=100;
T=GetFromBase('T',ones(4,1)*4);
gear=GetFromBase('gear',[3;2;1.5;1;.8]);
```

`T` and `gear` can now be used in the definition of block parameter values or in diagram Contexts. Here for example `T` is used to obtain the absolute times of gear shifts in the Context of the top diagram:

```
engine_par=[a,b,c,q];
T2=T(1);
T3=T2+T(2);
T4=T3+T(3);
T5=T4+T(4);
```

The values of `T2`, `T3`, `T4` and `T5` represent the shift times and may be used as block parameters. This is used to simplify the model as shown in Fig. 6.10.

Variables `gear` and `T` are available in the scope of the diagram so there is no need to use **FromBase** blocks to obtain their values. The latter are replaced with **Constant** blocks. The car Super Block may now be masked (see Fig. 6.11).

The masking is done by the auto masking operation followed by mask edition

Note that the optimization variables remain in the base after an optimization is run. So in order to restart

an optimization process with the original initial values, the optimization variables must be removed from the base. This can be done using the `clear` function.

The **OML** function `AddToBase` may be used in the Initialization script or the Context to define or redefine a variable in the base.
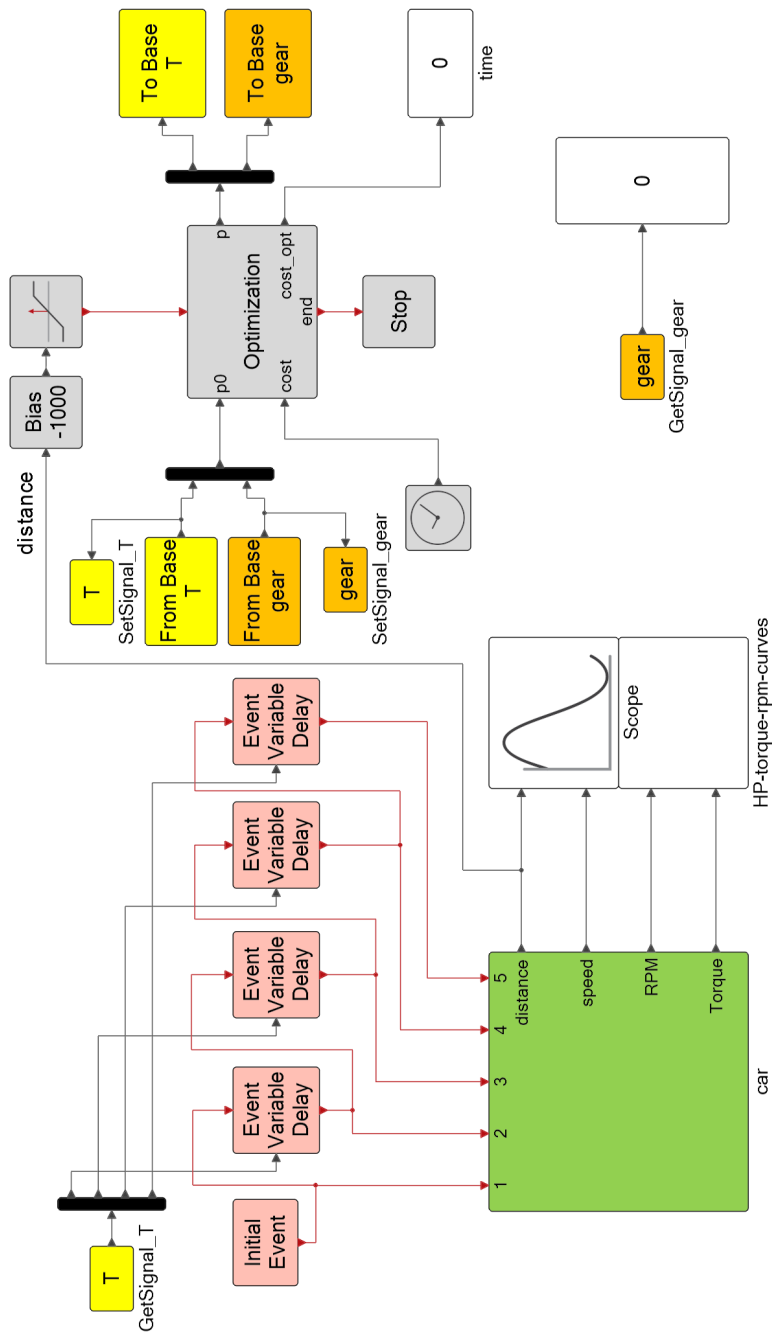
Figure 6.8: Car model for minimizing time to distance (car_min_time_dist_1000m.scm)
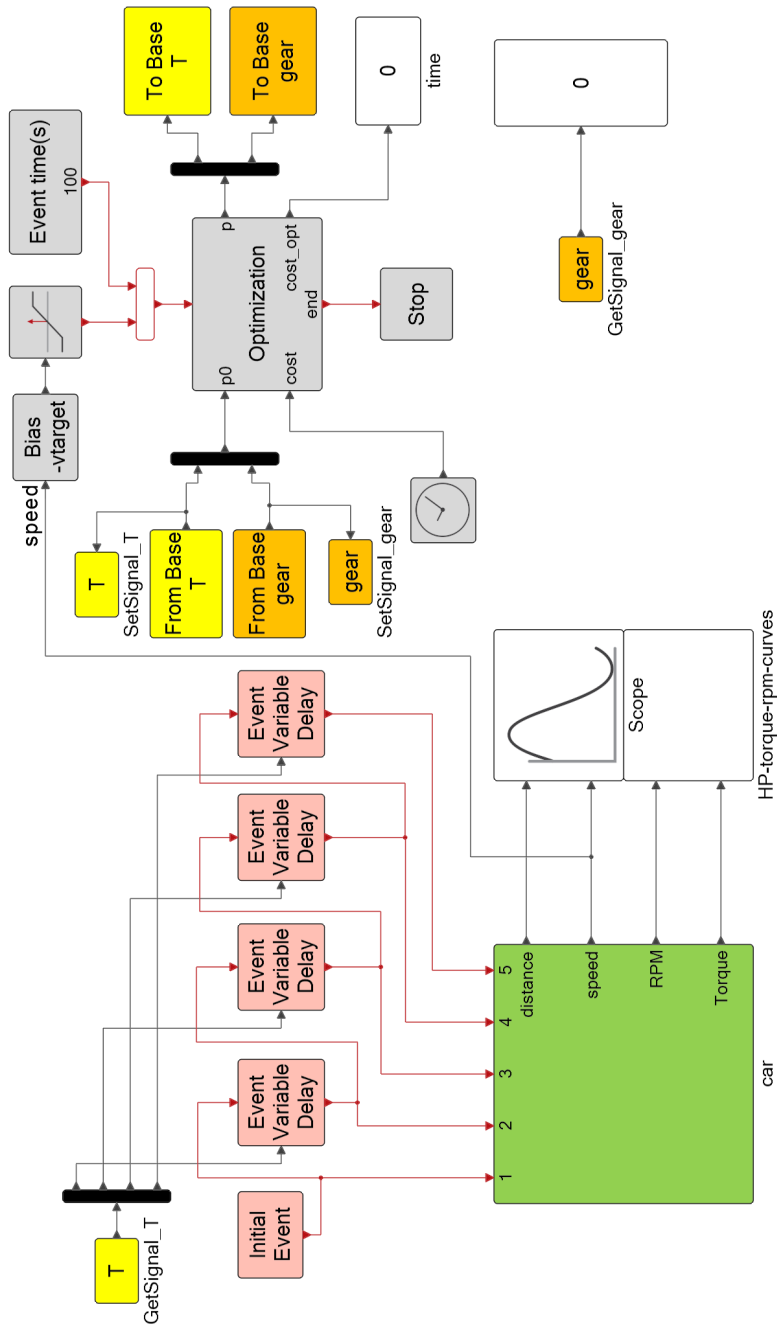
Figure 6.9: Car model for reaching speed in shortest time (car_min_time_speed_vtarget.scm)
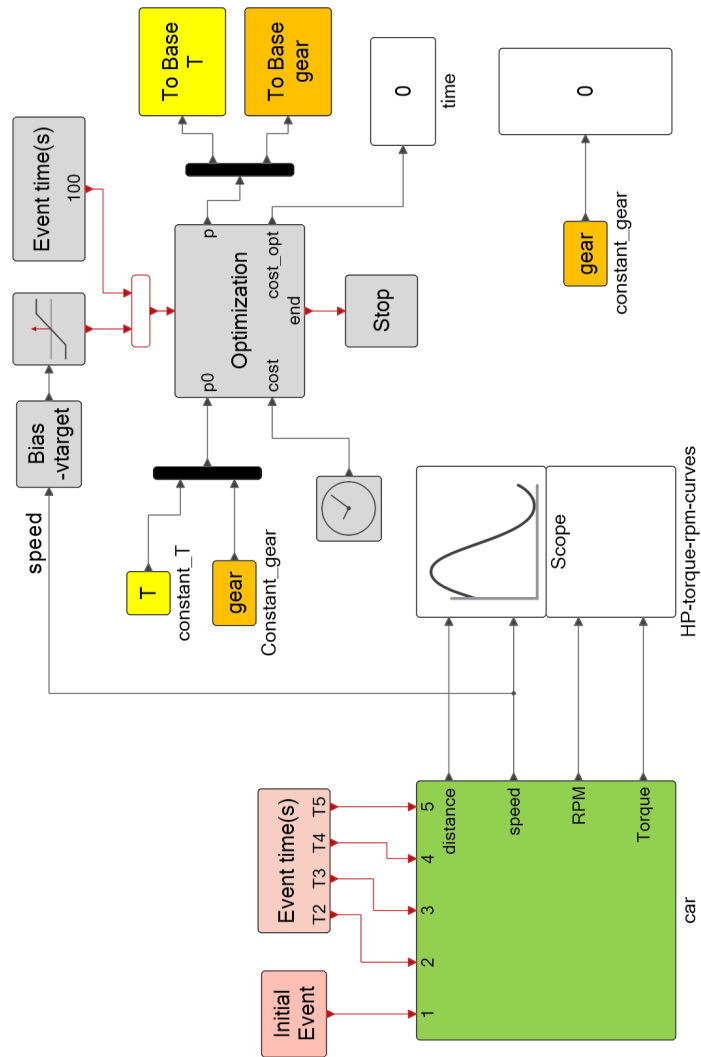
Figure 6.10: Model variant (car_min_time_speed_vtarget_2.scm)
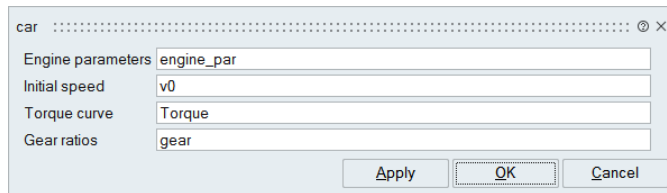
Figure 6.11: Masked "Car" superblock

6.4   Script based optimization

Script based optimization provides a powerful mechanism for solving very general optimization problems where the cost and constraints may be obtained from a combination of **Twin Activate** simulation results and an **OML** code. But in many cases, the cost is a simple function of signals generated by one or more **SignalOut** blocks and the optimization problem formulation is straightforward yielding virtually always the same **OML** script. For these simple cases, **Twin Activate** provides a graphical tool for formulating an optimization problem associated with a given model. This tool, which generates transparently the required optimization script, is presented in Section 6.5. But first direct scripting is presented.

6.4.1   Direct scripting of optimization code in OML

The full power of **OML** may be used to solve optimization problems associated with **Twin Activate** models. The code in **OML** would be the same as that for solving any optimization problem; the only specificity is that the cost would depend entirely or partially on the result of a **Twin Activate** simulation run. **OML** scripts used for such optimizations may also be included inside the **Twin Activate** model. See Chapter 21 for details.

Running a simulation from **OML** (in batch mode) requires creating a simulation object for the model. This object is needed to activate the graphical outputs of the Scopes present in the model. If graphical outputs are not needed, the simulation object is not needed.

The model evaluator is an **OML** function that is generated specifically for a given model. The evaluator evaluates all model parameters and creates the flat model structure, which is used by the compiler to create the simulation structure. The model evaluator must be created before running the simulator but it can be created only once for a given model. As parameters are changed during simulation runs, the evaluator function is re-evaluated but the evaluator function itself does not need to be recreated.

Finally once the simulation object and the evaluator are created, the model may be simulated.

The **OML** functions corresponding to the above operations are:

- `bdeCreateSimulationObject`: This function creates a simulation object for a given model specified as an scm model file.

- `vssCreateEvaluator`: This function creates an evaluator for the given model.

- `vssRunSimulation`: This function is used to run a **Twin Activate** simulation. The arguments include the evaluation function of the model and various workspaces.

There are various ways to exchange data between **OML** and the **Twin Activate** model. The most direct way to change model parameters from **OML** is to use the `vss_context` workspace. Variables used in a model for defining block parameters are defined in a workspace resulting from the execution of the Initialization script and various diagram Contexts following the semi-global scoping rule. When a simulation is run in batch mode, the workspace `vss_context` is injected in the workspace resulting from the execution of the Initialization script. Variables in `vss_context` may redefine variables defined in the Initialization script. In that case, the original values of the variables in the Initialization script can be considered as default or initial values, which can be modified during batch simulation. Note also that the presence of these default/initial values are needed to make the model self-contained.

To get data back to **OML** from the simulation, the **SignalOut** block may be used. This block returns a Signal object in the working environment. This environment is passed to the batch simulation function as optional argument.

6.4.2 Example

Consider again the optimization problem where the objective was to minimize the time to attain 1000m mark. The model is modified (Fig. 6.12) by removing the **Optimization** block and its supporting blocks. Variables `T` and `gear` are now defined in the Initialization script, and the simulation end time is returned in Signal **Tf**.
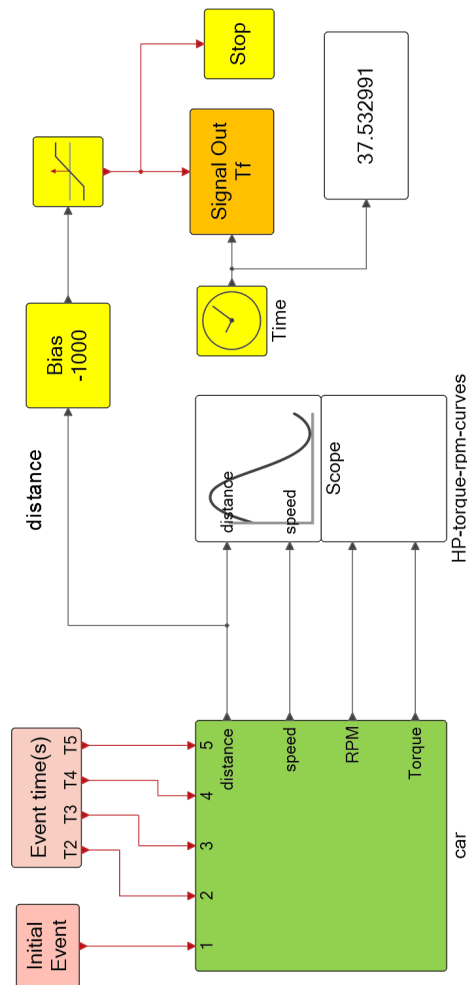
Figure 6.12: Modified model for batch use with OML (car_min_time_distance_1000_batch.scm)

The Initialization script includes now the definition of `T` and `gear`:

This model can be simulated since all the required variables, in particular `T` and `gear`, are defined. For optimization purpose however the values of these variables must be overwritten. This will be done using the `vss_context` workspace as shown below.

The optimization script may use any of the existing **OML** optimization functions or coded directly in **OML**. The following script, `car_min_time_distance_1000_batch.oml`, uses the `fminuncon` function.

```
function e=eval_cost(p,ev,env)
  vss_context=struct();
  vss_context.T=p(1:4);
  vss_context.gear=p(5:9);
  vssRunSimulation(ev,vss_context,env);
  Tf=getenvvalue(env,'Tf');
  e=Tf.ch{1}.data;
end

me=omlfilename('fullpath');
p=fileparts(me);
model=[p,'/car_min_time_distance_1000_batch.scm'];
T0=4*ones(4,1);
gear0=[3.2;1.8;1.3;1;.8];
p0=[T0;gear0];
simobj=bdeCreateSimulationObject(model);
ev=vssCreateEvaluator(model,simobj);
env=getnewenv;
cost=@(p) eval_cost(p,ev,env);
options=optimset('MaxFunEvals',1000);
p = fminunc(cost,p0,options);
T=p([1:4])
gear=p([5:9])
```

In the above example, the `SignalOut` block is used to return a single value (simulation time) but it can be used to return the whole time trajectory of signals and be used to implement very complex optimization scenarios.

In the following version of the gear ratio optimization problem, a constraint is imposed on the engine RPM. The problem is that of reaching a given speed as fast as possible without revving the engine above a maximum RPM (for example 5500 RPM). The optimization methods used so far, **BobyqaOpt** and `fminuncon`, did not allow the specification of such constraints. Bobyqa allowed bounds on the optimization variables but in this case, the engine RPM is not directly an optimization variable. To solve such constrained problems, the **OML** function `fmincon` may be used. The function `fmincon`, in addition to a cost value, takes a vector of constraint values. This function minimizes the given cost subject to the constraint being below specified values.

To use `fmincon`, the **Twin Activate** model must be extended to return the engine RPM. This is done inside the signal `omega` as shown in Fig. 6.13

The optimization script is then as follows:

```
function e=ev_cost(p,ev,env)
  vss_context=struct();
  vss_context.T=p([1:4]);
  vss_context.gear=p([5:9]);
  vssRunSimulation(ev,vss_context,env);
  Tf=getenvvalue(env,'Tf');
  e = Tf.ch{1}.data;
  if isempty(e) e=100; end
```
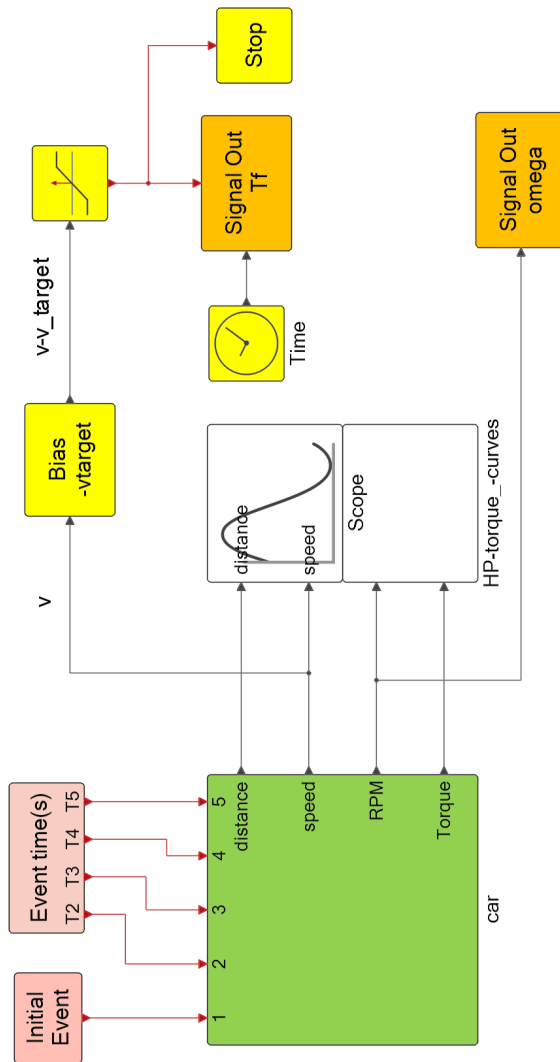
Figure 6.13: Model extended to return the `omega` signal
(car_min_time_speed_vtarget_w_constr_batch.scm)

```
end

function c=ev_constr(p,ev,env)
  vss_context=struct();
  vss_context.T=p([1:4]);
  vss_context.gear=p([5:9]);
  vssRunSimulation(ev,vss_context,env);
  omega=getenvvalue(env,'omega');
  w = omega.ch{1}.data;
  c=max(w)-5500;
end

me=omlfilename('fullpath');
```

```
p=fileparts(me);
model=[p,'/car_min_time_speed_vtarget_w_constr_batch.scm'];

T0=5*ones(4,1);gear0=[3.2;1.8;1.3;1;.8];p0=[T0;gear0];
lb=.1*ones(9,1);ub=20*ones(9,1);
simobj=bdeCreateSimulationObject(model);
ev=vssCreateEvaluator(model,simobj);
env=getnewenv;
cost=@(p) ev_cost(p,ev,env);
constr=@(p) ev_constr(p,ev,env);
options=optimset('MaxFunEvals',1000);
p = fmincon(cost,p0,[],[],[],[],lb,ub,constr,options);
T=p([1:4])
gear=p([5:9])
```

After execution of the optimization script, the optimal time evolution of the RPM is obtained as shown in Fig. 6.14.
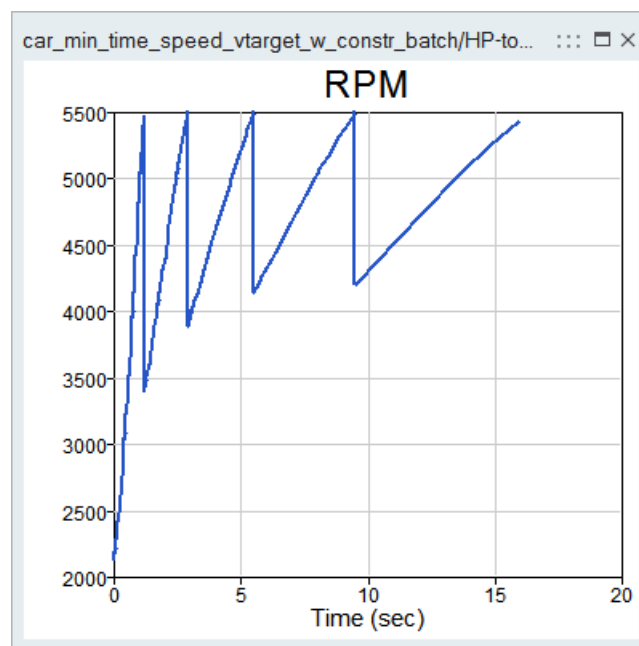
Figure 6.14: RPM vs. time after optimization.

The vehicle reaches 100Km/h in 15.9 seconds. Without the 5500 RPM constraint, the time was 15.76sec.

In this example, the constraint value, i.e., the maximum over the RPM signal was computed inside the **OML** function `ev_constr`. For that, the complete signal time history, `omega`, was returned to **OML**. The maximum value could have also been computed in the model and simply its final value returned to **OML** but that would have required adding a few more blocks to the diagram. In general the computation of the cost function and constraint values could be done partly in the model, partly in the **OML** code.
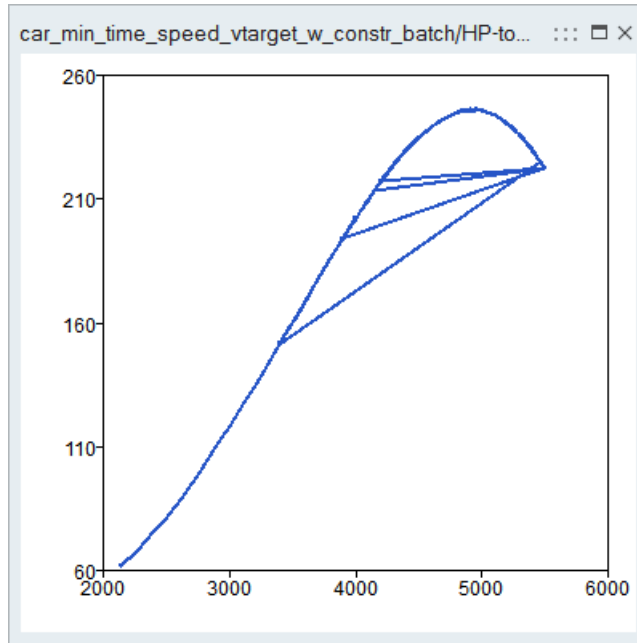
Figure 6.15: Horse power as a function of RPM after optimization.

6.5 Twin Activate Graphical optimization tool

The graphical interactive optimization tool generates and runs automatically the **OML** scripts for solving **Twin Activate** optimization problems. The tool can be used to associate optimization criteria to an existing **Twin Activate** model. In particular it provides a simple to use interface for defining optimization variables and defining cost and constraints.

The tool should be seen as a graphical interface for the **OML** optimization functions `fminuncon`, `fmincon` and `ga`. The interface is associated with a **Twin Activate** model, or more specifically with an scm file (the distinction is important since the modifications to the model are not taken into account until the model is saved). The information that needs to be provided in the interface is as follows:

- Model path: The association is done by entering the path of the scm file containing the **Twin Activate** model.

- Initialization: An **OML** code must be provided. Any variable defined in this code overwrites variables defined in the Initialization script of the model. The initial (guess) values of the optimization variables should not be defined in this code; they will be defined later. Variables needed to define them however may be defined here.

- Number of variables: The number of optimization variables must be given here. Depending on this number, the user is presented with a list of widgets were he should provide the name, the initial value and the range (min and max values) of each variable. The initial value may be defined as an expression using variables defined in the Initialization code. If the variable is a vector or matrix, the min and max values may be given as scalars, otherwise their sizes should match that of the variable.

- Number of constraints: The number of inequality constraints must be specified. This number, which may be defined as an expression using variables defined in the Initialization code, must match the size of the `_constraint` vector defined in the Optimization script when the optimiza-

tion problem contains constraints.

- Optimization script: An **OML** code used to define the variable `_cost` and, if the problem has constraints, the vector `_constraint`. The code may assume that all the variables defined in the Initialization code, all the signals associated with **SignalOut** blocks present in the model, in addition to the current values of the optimization variables produced by the optimizer are in its workspace. The optimizer searches the optimization variables within the specified bounds such that all the elements of `_constraint` are negative and the `_cost` is minimal.

- Optimization method: SQP and Genetic algorithm are proposed. More methods will be included in the future. SQP uses the **OML** functions `fmincon` or `fminbnd` functions depending on the presence or absence of constraints. The genetic algorithm uses `ga`.

- Depending on the selected optimization method, different optimization parameters such as Max number of iterations, Convergence tolerance, Constraint tolerance, Population size, etc., must be provided. These parameters are passed to the underlying **OML** optimization routine. See **OML** documentation for details.

The graphical interface provides, at the end of the simulation, the optimal cost, the number of iterations and gives access to the generated **OML** script. The cost value history versus the iteration number may also be plotted.

The solution to the problem is returned in the base environment inside the **OML** table `_optim`. In addition to the optimal values of the optimization variables, the table includes the optimal cost and their histories for all the iterations performed during optimization.

The optimization may be started using the run button on top of the graphical interface once all the required information is provided. At every step, the proposed values for the optimization parameters and the associated costs are displayed. The optimization process may be stopped at any time using the stop button.

The use of the tool is quite intuitive and is illustrated here through an example.

6.5.1 Example

The constrained optimization problem considered previously and solved using **OML** scripting is considered again, and formulated and solved with the graphical optimization tool. The problem is to find the best gear ratios to reach a given speed as fast as possible with constraint on the top RPM. The same **Twin Activate** model is used. The top RPM will be parameterized without modifying the model.

The graphical optimization tool corresponding to this problem may be created as shown in Fig. 6.16 and run using the Run button on top.

The Initialization code used in the optimization tool is virtually identical to the model Initialization script (see Fig. 6.17).

But instead of `T` and `gear`, it defines their initial values. Note that all model parameters may be modified directly in the tool without requiring any modifications of the **Twin Activate** model. So the end user does not even have to see the simulation model to run optimizations for different sets of parameters. For example optimization may be performed for different values of target speed and top RPM. All vehicle parameters, including the Torque curve can also be modified.

The bounds on the optimization variables `T` and `gear` are set as scalars. The time increment is always positive, so is the gear ratio which is assumed to be arbitrarily greater than 0.1 (0 value would be
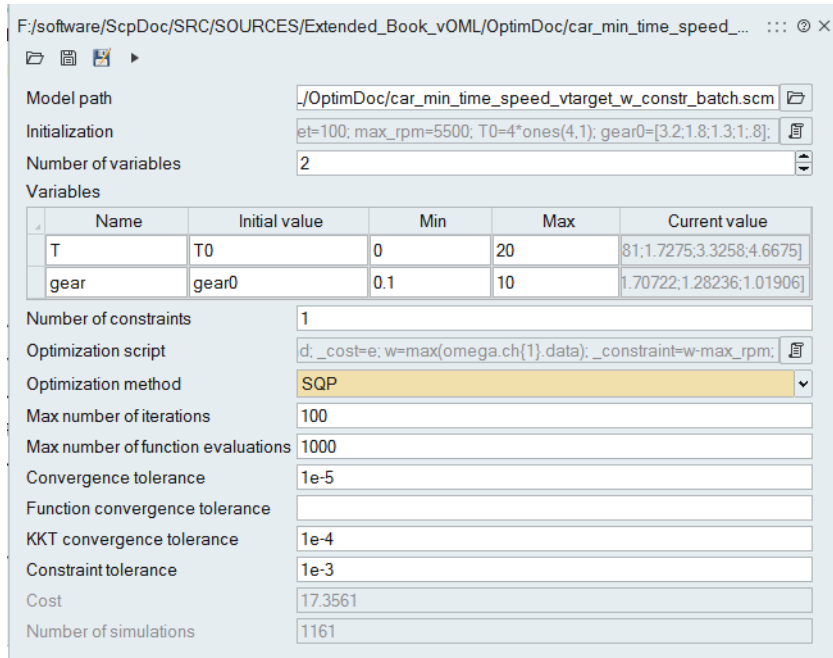
Figure 6.16: Optimization Tool
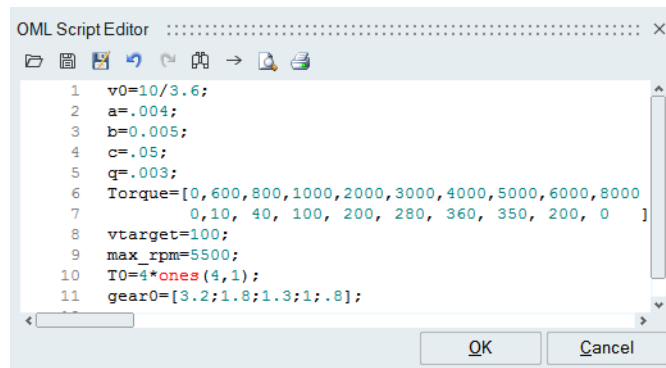
Figure 6.17: Initialization script used in the Optimization Tool

meaningless).

The number of constraints is set to 1 and the variables `_cost` and `_constraint` are defined as follows (Fig. 6.18):

Note the use of the signals `Tf` and `omega` produced by the **SignalOut** blocks of the model. The `Tf` signal however may be empty since the vehicle does not always reach the target speed before the end of the simulation. If that happens the cost is set to infinity so that the optimization may proceed. The optimization variables could also be used in defining the cost and constraints. Note however that they would have their current values as decided by the optimizer at the current step of the optimization and not necessarily their original values defined in the Initialization code.

Running the simulation gives the same result obtained previously with direct scripting. The optimal values of `T` and `gear`, and the optimal cost in addition to other information are available in Table `_optim` subsequent to running the optimization.
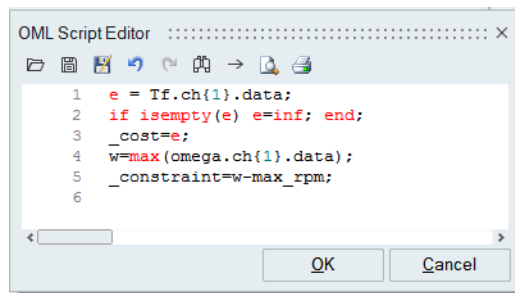
Figure 6.18: Constraint definition in the Optimization Tool

Chapter 7

Linearization

7.1 Linearization function

Twin Activate provides functions to create linear models from a Super Block by linearization. Linearization consists of constructing an approximate linear model around a given equilibrium (operating) point. Linearized models are constructed in particular for analysis and, controller and filter design.

The equilibrium point for the system

$$\begin{aligned} \dot{x} &= f(x, u) \\ y &= h(x, u) \end{aligned}$$

is defined as the triplet $(x_\mathsf{e}, u_\mathsf{e}, y_\mathsf{e})$ satisfying

$$\begin{aligned} 0 &= f(x_\mathsf{e}, u_\mathsf{e}) \\ y_\mathsf{e} &= h(x_\mathsf{e}, u_\mathsf{e}). \end{aligned}$$

The linearization of this system around the equilibrium point $(x_\mathsf{e}, u_\mathsf{e}, y_\mathsf{e})$ is a linear state-space system specified in terms of 4 matrices A, B, C and D:

$$\begin{aligned} \dot{\delta x} &= A \delta x + B \delta u \\ \delta y &= C \delta x + D \delta u \end{aligned}$$

where $\delta x = x - x_\mathsf{e}$, $\delta u = u - u_\mathsf{e}$, $\delta y = y - y_\mathsf{e}$, and

$$A = \frac{\partial f}{\partial x}(x_\mathsf{e}, u_\mathsf{e}), \quad B = \frac{\partial f}{\partial u}(x_\mathsf{e}, u_\mathsf{e}), \quad C = \frac{\partial h}{\partial x}(x_\mathsf{e}, u_\mathsf{e}), \quad D = \frac{\partial h}{\partial u}(x_\mathsf{e}, u_\mathsf{e}).$$

The main function to perform linearization in **OML** is `vssLinearizeSuperBlock`: and is used as follows

```
[A,B,C,D] = vssLinearizeSuperBlock(model,sblk,inps,outs,tf,ctx);
```

The first argument, `model`, representing the model in which linearization is to be performed, can either be a path of an `scm` file or a model object. The latter in general is the current model which can be obtained by the `bdeGetCurrentModel` function.

The second argument is the full name of the Super Block being linearized. The linearization is not necessarily performed with respect to all the inputs and outputs. Some of the inputs, in general corresponding to discrete-time signals, may have to be excluded. Similarly some outputs may need to be excluded. The arguments `inps` and `outs` are vectors of indices representing respectively the inputs and outputs to be considered for linearization.

The time `tf` is the time up to which the simulation is performed (in general to reach an equilibrium point) before linearization is done. The value of `tf` can be set to zero if the model is already in an equilibrium point (possibly thanks to the following argument `ctx` described below).

The final argument, `ctx`, is an external context, used to overwrite some of the variables defined in the Initialization script of the model. This is done in general to put the model in an equilibrium point without modifying it.

The function returns the (A, B, C, D) matrices of the linearized model.

7.1.1 Example: inverted pendulum

The classical problem of controlling an inverted pendulum mounted on a cart is considered. The objective is to control the force applied to the cart in such a way that the pendulum is stabilized pointing upwards. The force moves the cart on a ramp.
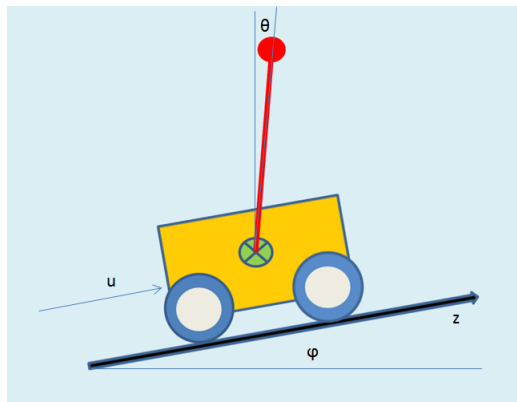
Figure 7.1: Inverted Pendulum on a ramp

Two sensors provide the position of the cart and the angle of the pendulum to the controller.

The equations of motion of the system may be expressed as follows:

$$\begin{cases} (M+m)\ddot{z} + ml\ddot{\theta}\cos(\theta-\phi)/2 - ml\dot{\theta}^2\sin(\theta-\phi)/2 &= u(t) - (M+m)g\sin(\phi) \\ (J+ml^2/4)\ddot{\theta} + ml\ddot{z}\cos(\theta-\phi)/2 - mgl\sin(\theta)/2 &= 0 \end{cases}$$

M represents the mass of the cart, and m the mass of the pendulum. The moment of inertia of the pendulum, considered to be a homogeneous bar, is $J = ml^2/12$. The cart moves on a rail placed on a ramp with an angle ϕ.

The system variables z and θ represent respectively the position of the cart on the ramp and the angle of the pendulum. The angle zero corresponds to the vertical position, pointing up.

The system may be re-expressed in matrix form as follows:

$$\begin{pmatrix} M+m & ml\cos(\theta-\phi)/2 \\ ml\cos(\theta-\phi)/2 & J+ml^2/4 \end{pmatrix} \begin{pmatrix} \ddot{z} \\ \ddot{\theta} \end{pmatrix} = \begin{pmatrix} ml\dot{z}^2\sin(\theta-\phi)/2 + u - (M+m)g\sin(\phi) \\ mgl\sin(\theta) \end{pmatrix}$$

The matrix on the left is square and invertible (provided $m > 0$), so $\ddot{z}$ and $\ddot{\theta}$ may be computed by using a simple matrix formula. So the cart pendulum system may be implemented in **Twin Activate** using the **MatrixExpression** block as shown in Fig. 7.2.
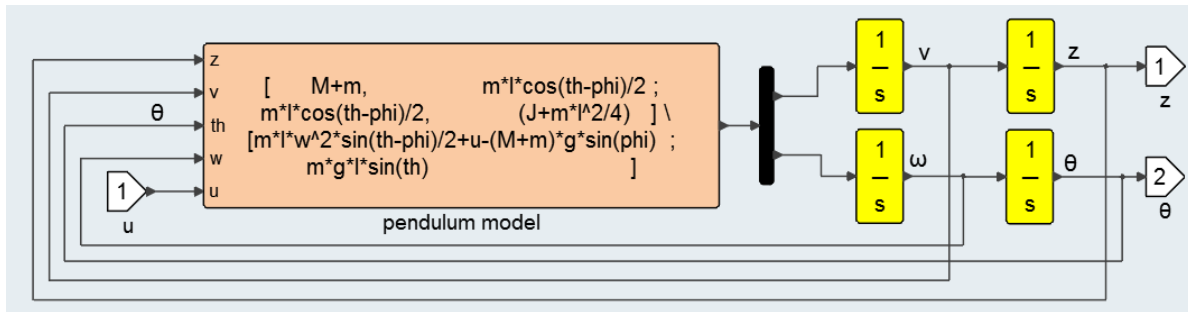
Figure 7.2: Pendulum superblock inside Inverted Pendulum model (inv_pendulum0.scm).

The parameters of the system are defined in the Super Block Context as follows:

```
m=1;
M=4;
l=.7;
g=9.8;
J=m*l^2/12;
```

The complete model is illustrated in Fig. 7.3. Note that the controller is a state space linear system with parameters `At, Bt, Ct`.
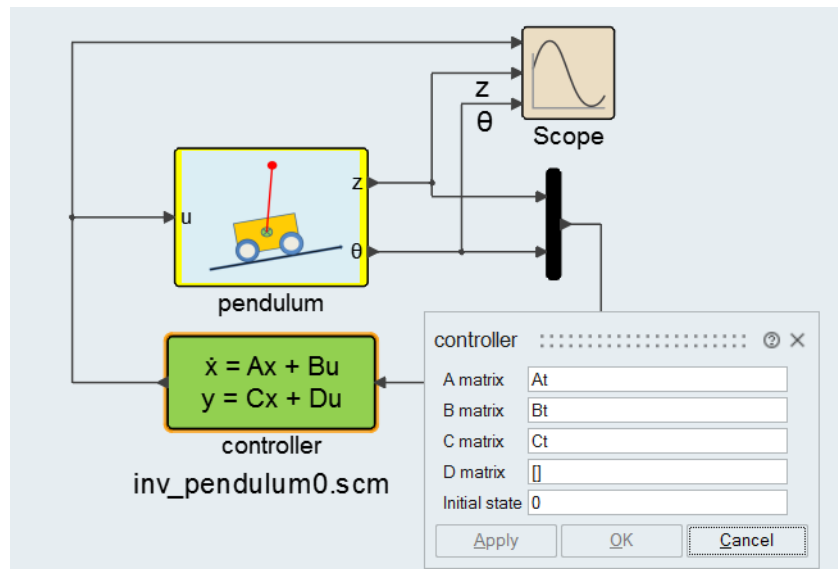
Figure 7.3: Inverted Pendulum model (inv_pendulum0.scm).

The model is designed in such a way that the simulation can be preformed both with and without a linearized model. In the absence of a linearized model, a "zero" controller (controller producing zero as output) is used. This is done in the Context of the top diagram as follows:

```
_syslin=GetFromBase('_syslin',[]);
```

```
if isempty(_syslin)
  At=[];
  Bt=zeros(0,2);
  Ct=zeros(1,0);
else
  A=_syslin.a;
  B=_syslin.b;
  C=_syslin.c;
  D=_syslin.d;
  n=size(A,1);
  Kc=-place(A,B,-1*ones(1,n));
  Kf=-place(A',C',-2*ones(1,n));
  Kf=Kf';

  At=A+B*Kc+Kf*C+Kf*D*Kc;
  Bt=-Kf;
  Ct=Kc;
end
```

So if a structure called `_syslin` is defined in the base environment, it is used as the linearized model to design an observer-based controller. If not, the controller is set to zero. This way, when the model is linearized, the linearization is performed in the "open-loop" case (i.e., the plant alone).

The linearization can be performed with the following script:

```
clear('_syslin');
% vector of input port indices considered for linearization
inps = 1;
% vector of output port indices considered for linearization
outs = [1,2];
% Modified context
ctx=struct;
ctx.z0 = 0;
ctx.th0 = 0;
ctx.phi = 0;

model=bdeGetCurrentModel;
% Selected Super block to linearize
superblock = 'pendulum';

[A,B,C,D]=vssLinearizeSuperBlock(model,superblock,inps,outs,0,ctx);
_syslin.a=A;
_syslin.b=B;
_syslin.c=C;
_syslin.d=D;
```

The linearization is done around the point $z = 0$, $\theta = 0$, $v = 0$ and $\omega = 0$. This is an equilibrium point assuming $\phi = 0$. The resulting controller stabilizes the system for small values of ϕ but with a bias in the final position of the cart. For example in the case (Initialization script):

```
phi=0.001;
z0=-0.01;
th0=.002;
```

The simulation result is shown in Fig. 7.4. Note that to obtain this result, first the script must be run so that `_syslin` is made available in the base, then the model simulated.
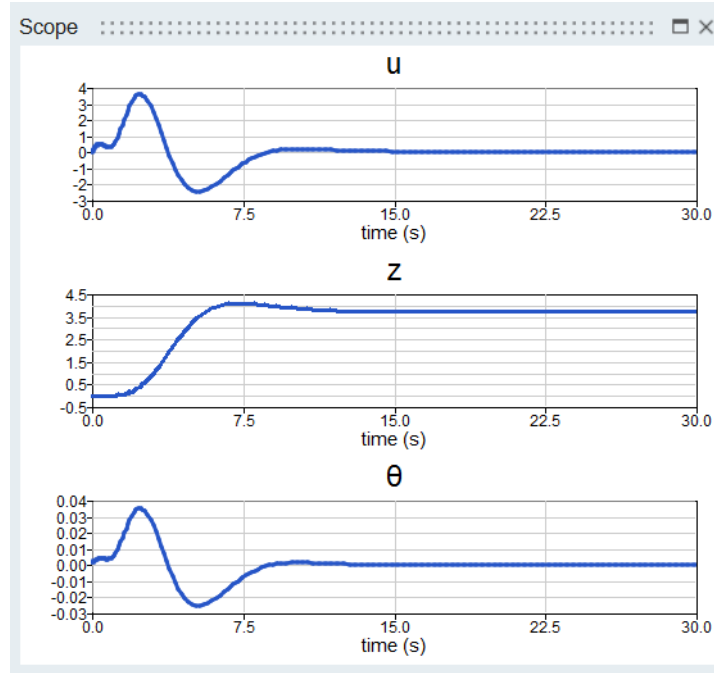
Figure 7.4: Simulation result for (inv_pendulum0.scm). Note that z does not converge to zero since ϕ is not zero in this case.

Changing the model   The final bias in z can be removed either by taking into account the value of ϕ and linearizing the plant accordingly (this requires the computation of a new equilibrium point and will be addressed in the next section) or by introducing an integral in the feedback loop (of z). This can be done by changing the model as illustrated in Figs. 7.5 and 7.6.
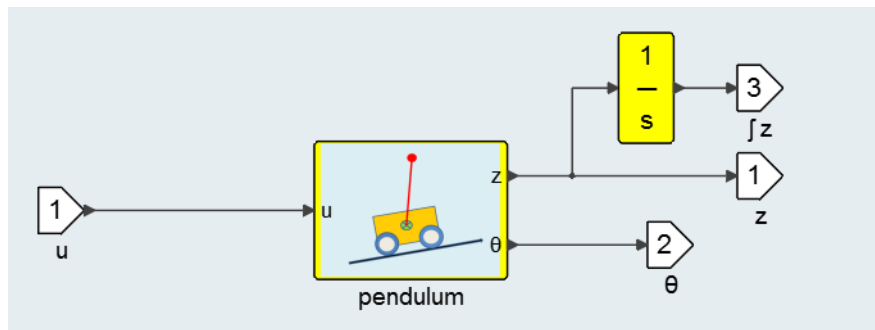
Figure 7.5: Pendulum_integral superblock inside the Inverted Pendulum model (inv_pendulum_integral.scm). The integral of z is added as an output to be used for feedback control.

The linearization is performed using the following script:
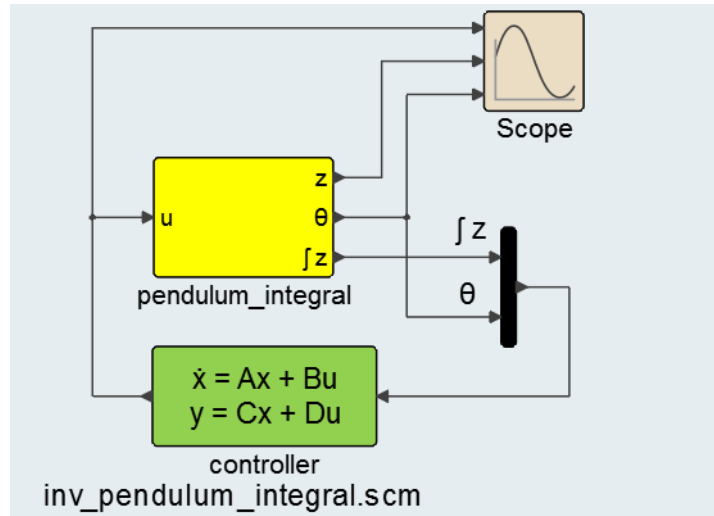
Figure 7.6: Inverted Pendulum model (inv_pendulum_integral.scm).

```
clear('_syslin');
% vector of input port indices considered for linearization
inps = 1;
% vector of output port indices considered for linearization
outs = [3,2];
% Modified context
ctx=struct;
ctx.z0 = 0;
ctx.th0 = 0;
ctx.phi = 0;

model=bdeGetCurrentModel;
% Selected Super block to linearize
superblock = 'pendulum_integral';

% Linearization
[A,B,C,D]=vssLinearizeSuperBlock(model,superblock,inps,outs,0,ctx);
_syslin.a=A;
_syslin.b=B;
_syslin.c=C;
_syslin.d=D;
```

Note that only the second and third outputs of the Super Block (in the reverse order) are used in the definition of the Plant to be linearized. The simulation result is shown in Fig. 7.7. The bias is removed thanks to the integral in the loop.

To simplify the use of the function `vssLinearizeSuperBlock` for a given Super Block, the menu `Linearize` is provided. By selecting a Super Block and using the contextual menu `Linearize`, an **OML** script is automatically generated containing the proper call to the function `vssLinearizeSuperBlock` for linearizing the selected Super Block. The script contains also information on how to calculate the equilibrium point. This functionality will be discussed later in this section.
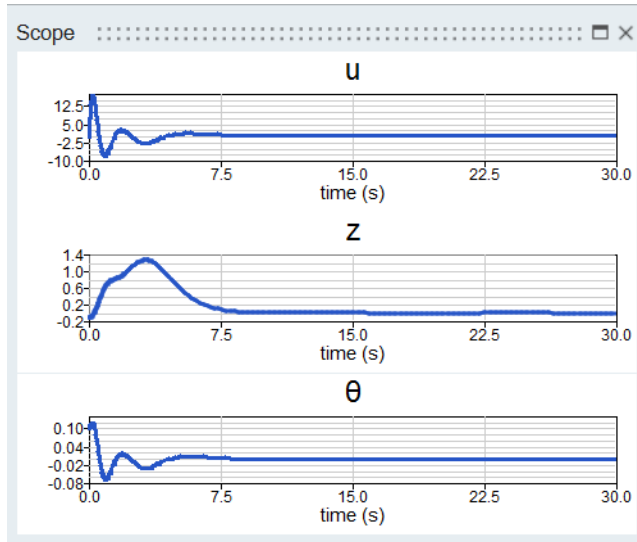
Figure 7.7: Simulation result for (inv_pendulum_integral.scm). Note that z does converge to zero even though ϕ is not zero.

The linearization script can be included inside the model using the **ExecOMLScript** block presented in Chapter 21.

The `vssLinearizeSuperBlock` function can also be used inside the Initialization script of the model to integrate controller design within the simulation model. This powerful mechanism is illustrated in the following example.

7.1.2 Example: Cart on a beam

A cart of mass m is placed on a beam. The cart can move freely along the beam without any friction. The beam pivots around a fixed point O. Its moment of inertial is denoted by J and its angle of rotation by θ. The angle $\theta = 0$ corresponds to the horizontal position.
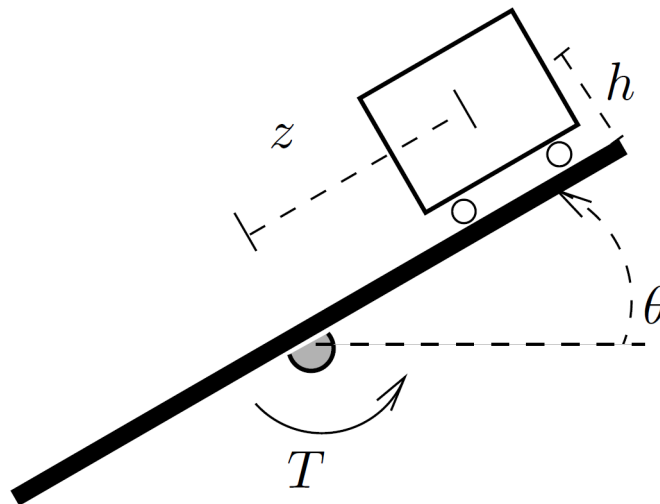
The variable z denotes the projection of the center of mass of the cart on the beam; the zero position corresponds to the point O. The height of the center of mass of the cart (its distance to the beam) is

denoted h.

The mechanism is controlled by applying a torque T to the beam around the point O. The objective of the control is to stabilize the cart on the beam.

The equations of motion for this system can be expressed as follows:

$$
\begin{aligned}
m\ddot{z} &= mz\dot{\theta}^2 - mg\sin(\theta) \\
(J + mz^2)\ddot{\theta} &= T - 2mz\dot{z}\dot{\theta} - mgz\cos(\theta) + mgh\sin(\theta)
\end{aligned}
$$

The objective is to design an observer-based controller for the system. This is done by first linearizing the system, then designing an observer-based controller for the linearized model. The measured outputs are considered to be the angle θ and the position z:

$$
y = \begin{pmatrix} z \\ \theta \end{pmatrix}.
$$

The control input is $u = T$. So the system has four states, one input and two outputs.

The system is modeled in Modelica language as follows

```
model cart
  //parameters
  parameter Real h = 1;
  parameter Real m = 2;
  parameter Real J = 10;
  parameter Real th0 = 0;
  parameter Real z0 = 0;
  //input variables
   Real T;
  //output variables
   Real z(start=z0), th(start=th0);
   Real zd(start=0), thd(start=0);
   Real g=9.81;
equation
  der(th)=thd;
  der(z)=zd;
  der(zd) = z*thd*thd-m*g*sin(th);
  (J+m*z*z)*der(thd) = T-2*m*z*zd*thd-
      m*g*z*cos(th)+m*g*h*sin(th);
end cart;
```

which is implemented using a Custom Modelica block. The system could also be implemented using standard **Twin Activate** blocks.

The **Twin Activate** model is given in Fig. 7.8 and 7.9.

The initial position of the cart and the initial angle of the beam, `z0` and `th0`, are not necessarily zero for simulation but must be set to zero for linearization. This is done in the Initialization script of the model as follows:

```
h=1; m=2; J=10;
nx=4;nu=1;ny=2;
```
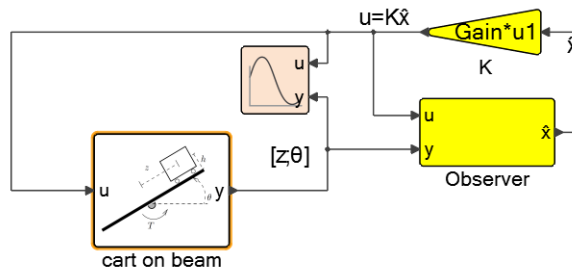
Figure 7.8: The model of the cart on beam system with an observer-based controller.

$$
\begin{aligned}
m\ddot{z} &= mz\dot{\theta}^2 - mg\sin(\theta) \\
(J + mz^2)\ddot{\theta} &= T - 2mz\dot{z}\dot{\theta} - mgz\cos(\theta) + mgh\sin(\theta)
\end{aligned}
$$
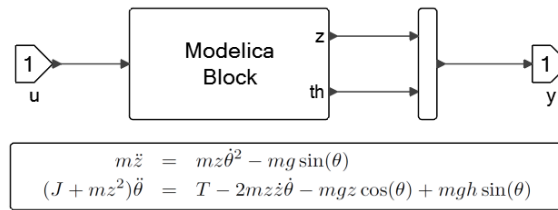
Figure 7.9: Inside the 'cart on beam' Super Block being linearized.

```
th0=pi/12;
z0=-1;

if ~vssIsInLinearization
  ctx=struct;
  ctx.K=zeros(1,nx);
  ctx.obs=ss([],zeros(0,nu+ny),zeros(nx,0),zeros(nx,nu+ny));
  ctx.th0=0;ctx.z0=0;
  [A,B,C,D]=vssLinearizeSuperBlock(bdeGetCurrentModel,...
    'cart on beam',[1],[1],0,ctx);
  L=-place(A',C',-3*ones(nx,1))';
  obs=ss(A+L*C,[B+L*D,-L],eye(nx),zeros(nx,nu+ny));
  K=-place(A,B,-3*ones(nx,1));
```
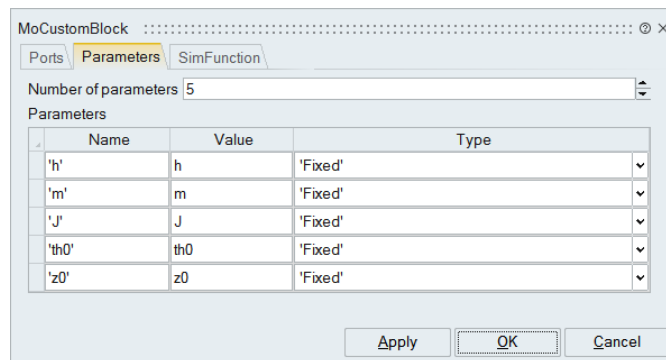

Figure 7.10: The parameters tab of the Custom Modelica block dialog GUI.

```
end
```

The variable `vssIsInLinearization` is false in general; it is true only when the Initialization script is executed in the phase of linearization (the model is evaluated and simulated for linearization). As it can be seen in the script, when the model is in the linearization stage, the controller gain `K` is set to zero and the observer `obs` similarly to a trivial linear system. This is done through the external context `ctx`. `ctx` is used also to set to zero the initial states for linearization.

When the script is not used in the linearization stage, the observer-based controller is constructed and used for simulation. To construct an observer-based controller for a linear system (A, B, C, D), a feedback gain matrix K is found such that $A + BK$ is Hurwitz (all eignvalues have strictly negative real parts) and an observer gain L is found such that $A + LC$ is Hurwitz. This is possible as long as the system is stabilizable and detectable. Here the pole placement **OML** function `place` is used to compute K and L.

So when this model is simulated, the Super Block 'cart on beam' is linearized, the result of linearization is used to construct the observer-based controller, and the simulation of the resulting model for non zero initial conditions is performed. The simulation result is shown in Fig. 7.11.
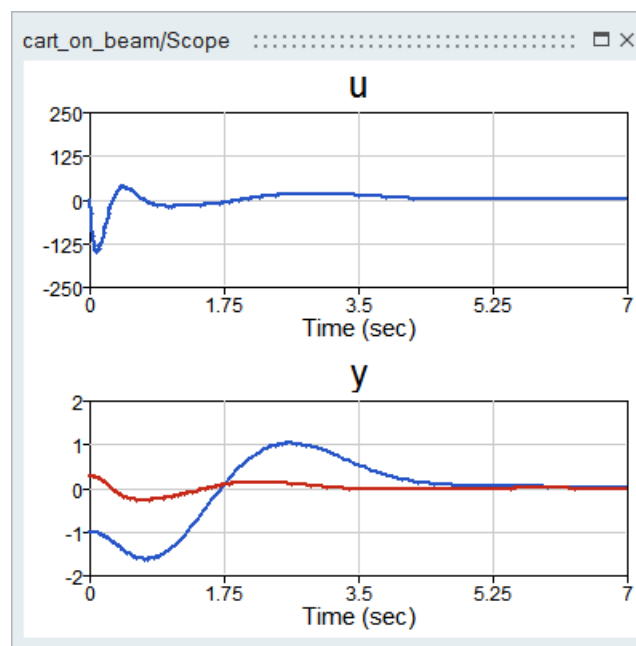
Figure 7.11: The simulation result shows that the system is stabilized.

Changing the system   The model used above uses the model of a cart on a beam with the control being the torque applied to the beam. The model however, and in particular the Initialization script does not use any information related to the underlying system defined inside the Super Block that it controls except input, output and state sizes and certain system parameters. So this construction is very generic and can be applied to any other system placed inside the Super Block by bringing small adjustments to the Initialization script.

Consider here the following modification of this example: instead of the control being a torque applied to the beam, it is now a force applied to the cart at the level of its wheels (along the beam). The equations

of motion the become:

$$
\begin{aligned}
m\ddot{z} &= mz\dot{\theta}^2 - mg\sin(\theta) + T \\
(J + mz^2)\ddot{\theta} &= -2mz\dot{z}\dot{\theta} - mgz\cos(\theta) + mgh\sin(\theta)
\end{aligned}
$$

where T represents now the applied force. This new system can now be placed inside the Super Block and no other modification is required. This is done in Model `cart_on_beam_force.scm`. The simulation shows that the controller is properly constructed and the system is stabilized.

Stabilizing the cart off center Consider now the problem of stabilizing the cart not at the center of the beam but at a different location. A possible approach would be to use the exact same model but introduce a bias in the output of the system before feeding it to the controller, `y0`. See Fig. 7.12. The idea being that the controller will drive `y-y0` to zero thus imposing the desired output.

The variable `y0` is defined in the Initialization script as follows

```
zobj=0.2;y0=[zobj;0];
```

The value of `zobj` corresponds to the location where the cart is to be stabilized. The simulation result is shown in Fig. 7.13. It can be seen that even though the use of bias has moved the location of the cart, the desired value `zobj` is not achieved. The reason is that the steady-state gain of the controller is not infinite.
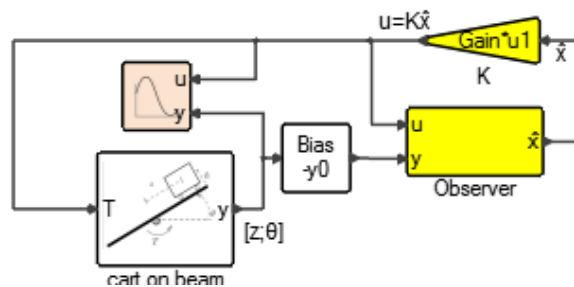
Figure 7.12: The model of the 'cart on beam' system with an observer-based controller.

An alternative solution for stabilizing the cart off center is to linearize the system off center. For that, an equilibrium point must be found. The solution which consists of running the simulation for certain time cannot be used in this case because the system is unstable in open loop. The equilibrium point should then be computed differently.

7.2   Computation of the equilibrium point

The function `vssEquilibriumPointSuperBlock` is provided for finding equilibrium points by modifying model parameters defined by the user. The simple usage of this function is as follows

```
[ctx,err]=vssEquilibriumPointSuperBlock(model,sblk,ctx,params);
```

The first three input arguments of the function are identical to that of `vssLinearizeSuperBlock` function seen above. The cell array `params` contains the list of variable names in the `ctx` that can be modified by the function in order to set the state derivatives of all the continuous-time states of the blocks inside the Super Block `sblk` to zero. `ctx` is returned with the value of these variables modified.
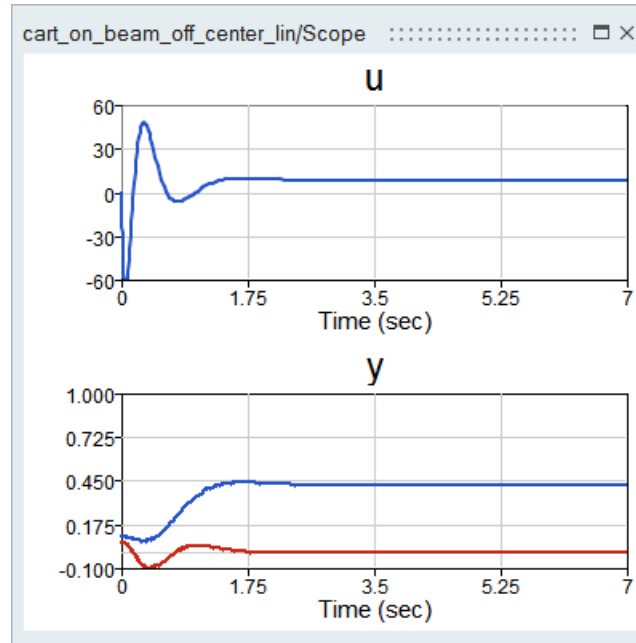
Figure 7.13: The simulation result shows that the cart is stabilized off the center of the beam but not at the desired location.

In the cart on the beam example, the equilibrium state is known `x0=[zobj;0;0;0]`, so is the equilibrium output `y0=[zobj;0]`, which is imposed. What is unknown is the equilibrium input `u0`.

The **Twin Activate** model using this function is given in Fig. 7.14 This is done in the Initialization script
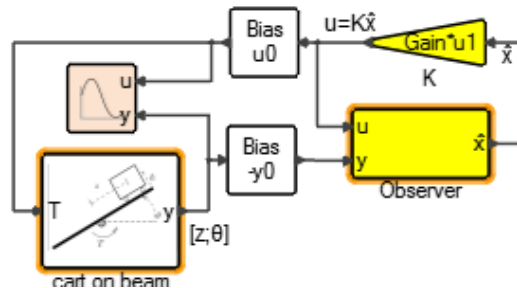
Figure 7.14: The model of the cart on beam stabilized with a controller linearized off center.

of the model as follows:

```
h=1; m=2; J=10;
nx=4;nu=1;ny=2;
th0=pi/50;
z0=0.5;
zobj=1;y0=[zobj;0];

if ~vssIsInLinearization
  ctx=struct;
  ctx.K=zeros(1,nx);
```

```
  ctx.obs=ss([],zeros(0,nu+ny),zeros(nx,0),zeros(nx,nu+ny));
  ctx.th0=0;ctx.z0=zobj;ctx.u0=0;
  model=bdeGetCurrentModel;
  ctx=vssEquilibriumPointSuperBlock(model,...
'cart on beam',ctx,{'u0'});
  [A,B,C,D]=vssLinearizeSuperBlock(model,...
   'cart on beam',[1],[1],0,ctx);
  L=-place(A',C',-7*ones(nx,1))';
  obs=ss(A+L*C,[B+L*D,-L],eye(nx),zeros(nx,nu+ny));
  K=-place(A,B,-7*ones(nx,1));
  u0=ctx.u0;
end
```

The simulation results shows that the desired location is achieved. See Fig. 7.15
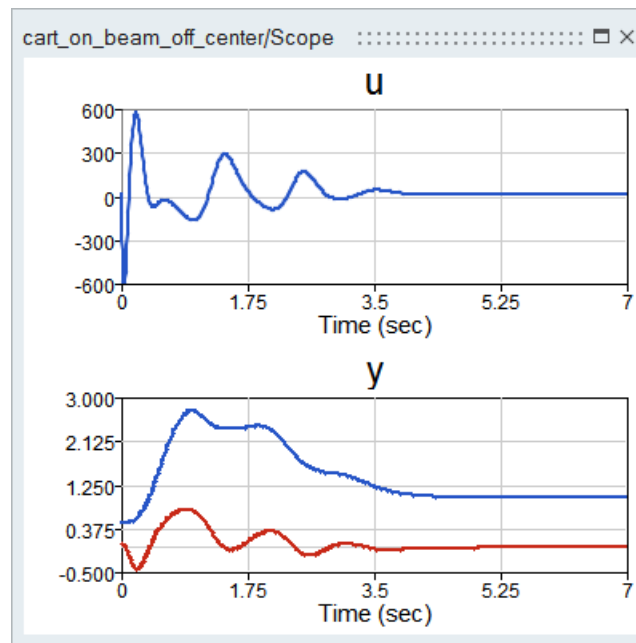
Figure 7.15: The simulation result shows that the objective is achieved.

The `vssEquilibriumPointSuperBlock` function has additional optional arguments:

`[ctx,err]=vssEquilibriumPointSuperBlock(model,sblk,ctx,params,inps,outs,L,M,verb);`

The optional argument `inps` is used to set the time-dependency of the inputs of the superblock to consider for the computation of the equilibrium point. This time dependency can also be set directly inside the diagram by checking the time-dependency parameter of the corresponding input port blocks. The input sizes can also be defined by setting the parameters of these blocks. This may be required since in some cases the compiler may not be able to determine port sizes by considering only the diagram inside of the Super Block. `inps` is a vector containing port numbers and its default value is `[]`.

The optional argument `outs` specifies the outputs of the Super Block to take into account in the construction of the output vector y. The default is `[]`.

The `L` and `M` matrices specify the vector which is set to zero by `vssEquilibriumPointSuperBlock`. This vector is: `[L*xdot;M*y]`. The default values are 1. If `L` is set to `[]`, `L*xdot` is excluded from the vector. Similarly, If `M` is set to `[]`, `M*y` is excluded from the vector.

If the optional argument `verb` is set to the string `'v'`, the verbose mode is turned on. In this mode, the values tried by the solver and the corresponding vector `[L*xdot;M*y]` are displayed at each iteration. This mode is useful for debugging.

So in general the `vssEquilibriumPointSuperBlock` function can be used to use the specified parameters to set parts or all of the state derivatives `xdot` and the outputs `y` to zero. A systematic approach to finding the equilibrium point of a system is to construct the corresponding Super Block so that the inputs and output are respectively $\delta u = u - u_\mathrm{e}$ and $\delta y = y - y_\mathrm{e}$. These modifications can be introduced using Bias blocks. The bias values u_e and y_e may be partially imposed, partially free for finding the equilibrium point.

Consider again the cart on the beam example. To obtain the equilibrium point previously, the value of the equilibrium state x_e was "guessed" from the value of the y_e, and the only free variable was u_e. In particular it was noted that the values of z_e and θ_e are equal to the values specified in y_e.

Here, instead of guessing the values of z_e and θ_e, a systematic approach is used. This is done by creating the Super Block by including bias terms so that the input and output are $\delta u = u - u_\mathrm{e}$ and $\delta y = y - y_\mathrm{e}$. See Fig. 7.16.
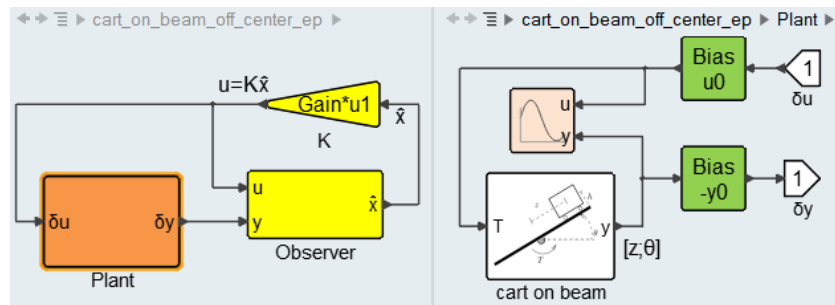
Figure 7.16: The Plant Super Block is used for computing the equilibrium point and linearization. See demo model cart_on_beam_off_center_ep.scm.

7.2.1 Use of infinite steady-state gain controller

It was seen that simply by introducing a bias on the output of the controller designed for stabilizing the cart at the center stabilizes the cart off center. However the location of the stabilization does not correspond to the desired location because of the finite steady-state controller gain. If a controller with infinite gain is used, stabilization at the desired location can be achieved.

A way to obtain infinite steady-state gain is to introduce explicitly an integrator in the system. This is done in the model shown in Fig. 7.17 and 7.18.

The Initialization script in this case is

```
h=1; m=2; J=10;
nx=5;nu=1;ny=2;
th0=pi/50;
z0=0.01;
```
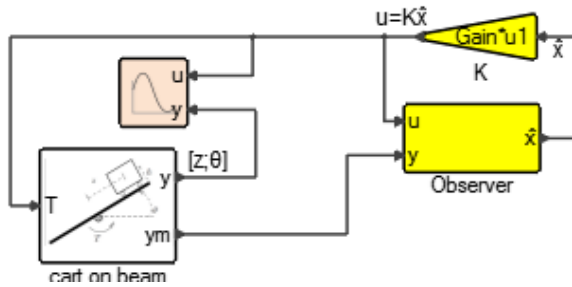
Figure 7.17: The model of the cart using an infinite steady-state gain controller.

$$m\ddot{z} = mz\dot{\theta}^2 - mg\sin(\theta)$$
$$(J + mz^2)\ddot{\theta} = T - 2mz\dot{z}\dot{\theta} - mgz\cos(\theta) + mgh\sin(\theta)$$
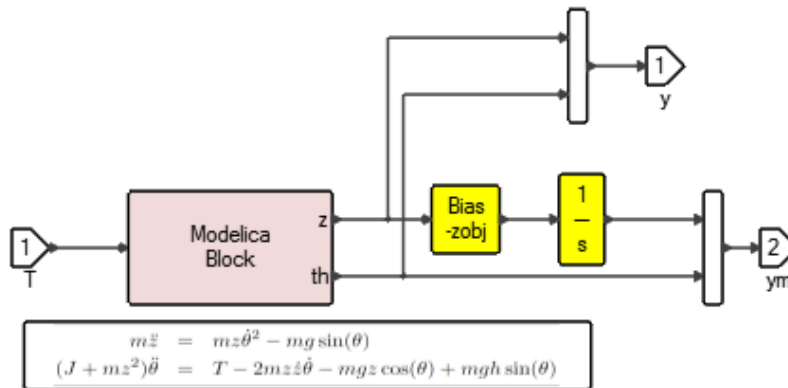
Figure 7.18: Inside the cart on beam Super Block. An integrator is included to force infinite steady-state gain.

```
zobj=0.025;y0=[zobj;0];

if ~vssIsInLinearization
  ctx=struct;
  ctx.K=zeros(1,nx);
  ctx.obs=ss([],zeros(0,nu+ny),zeros(nx,0),zeros(nx,nu+ny));
  ctx.th0=0;ctx.z0=0;ctx.zobj=0;
  model=bdeGetCurrentModel;
  [A,B,C,D]=vssLinearizeSuperBlock(model,...
    'cart on beam',[1],[2],0,ctx);
  L=-place(A',C',-5*ones(nx,1))';
  obs=ss(A+L*C,[B+L*D,-L],eye(nx),zeros(nx,nu+ny));
  K=-place(A,B,-5*ones(nx,1));
end
```

Note that for the linearization only the second output is taken into account. This is the output which is connected to the controller. The first output corresponding to the actual outputs of the system is used for display only.

The simulation result is shown in Fig. 7.19.
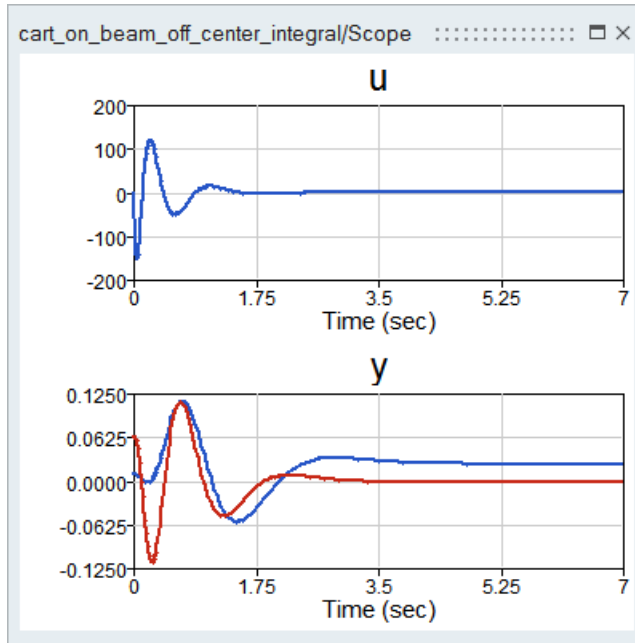
Figure 7.19: The simulation result shows that the system is stabilized at the desired location.

The advantage of using this approach is that the value of `zobj` is not used in the design process and it can be changed during the simulation. The model in Fig. 7.20 is similar to the previous model except that instead of the desired location being defined by a constant `zobj`, it is a time varying signal (a square wave in particular).
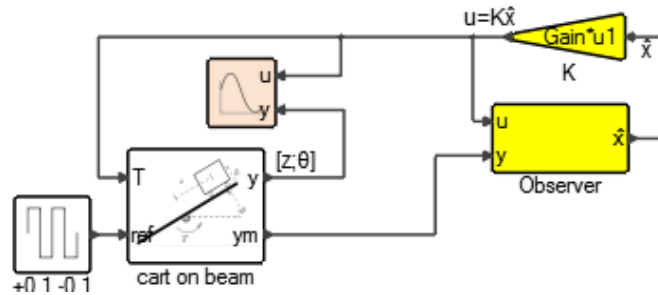
Figure 7.20: The inside of the 'cart on beam' Super Block is shown in Fig. 7.21.

For the linearization in this case the first input and the second output are taken into account.The simulation result is shown in Fig. 7.22.

7.2.2 Equilibrium point for the inverted pendulum example

Consider again the inverted pendulum problem considered in the previous section. Here the assumption is made that the value of ϕ is known so the Plant can be linearized around the equilibrium point corresponding to the given value of ϕ. In this case the input is not linearized around 0 but a value U, which must be determined so that the system is in an equilibrium point. The equilibrium state here is zero.

$$m\ddot{z} = mz\dot{\theta}^2 - mg\sin(\theta)$$
$$(J + mz^2)\ddot{\theta} = T - 2mz\dot{z}\dot{\theta} - mgz\cos(\theta) + mgh\sin(\theta)$$
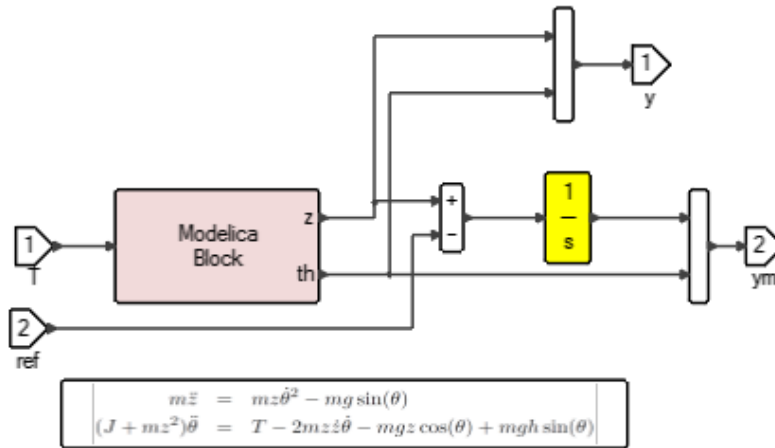
Figure 7.21: Inside the cart on beam Super Block. An integrator is included to force infinite steady-state gain.

The input bias `U` is introduced as shown in Fig.7.23. The value of `U` is not known and must be determined by the computation of the equilibrium point.

The computation of the equilibrium point and linearization is performed with the following script:

```
clear('_syslin');
% vector of input port indices considered for linearization
inps = 1;
% vector of output port indices considered for linearization
outs = 1:2;
% Modified context
ctx=struct;
ctx.z0 = 0;
ctx.th0 = 0;
ctx.U = 0;

model=bdeGetCurrentModel;
% Selected Super block to linearize
superblock = 'pendulum';
% variables in ctx allowed to adjust
vars={'U'};
ctx=vssEquilibriumPointSuperBlock(model,superblock,ctx,vars,inps,outs,1,[],'v');
% Linearization
[A,B,C,D]=vssLinearizeSuperBlock(model,superblock,inps,outs,0,ctx);
_syslin.a=A;
_syslin.b=B;
_syslin.c=C;
_syslin.d=D;
_syslin.u0=ctx.U;
```

The verbose mode is used to monitor the progress of the optimization code computing the equilibrium point.
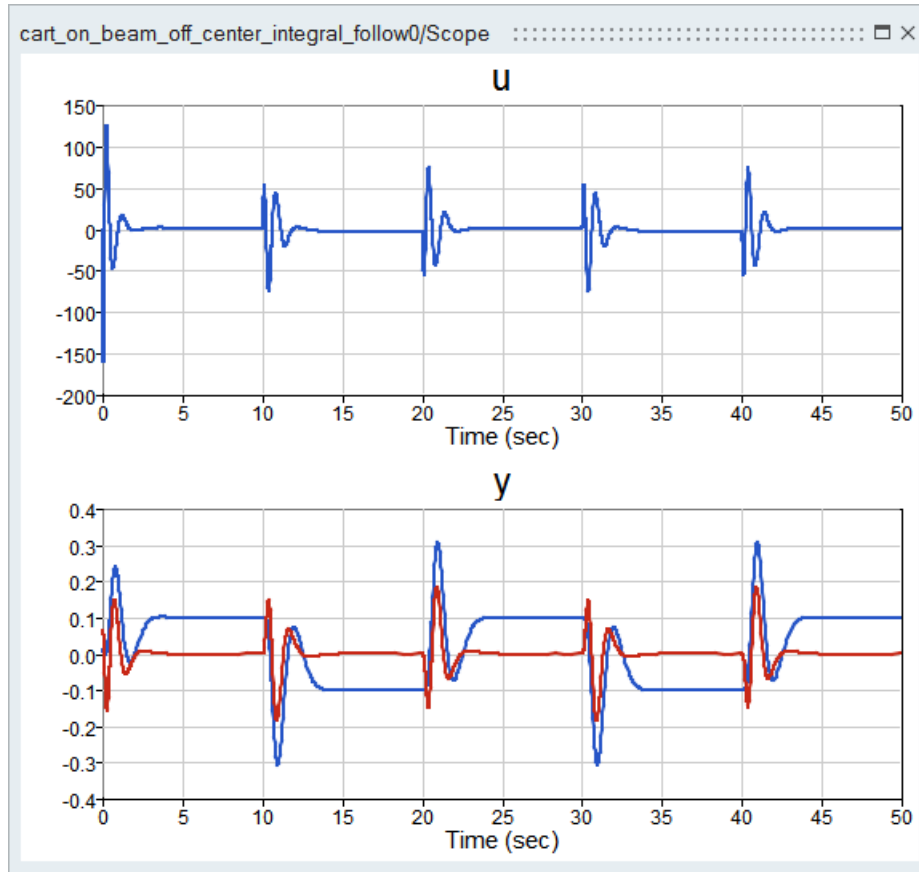
Figure 7.22: The simulation result shows that the location of the cart follows the square wave reference trajectory.

The resulting simulation result is shown in Fig.7.24.
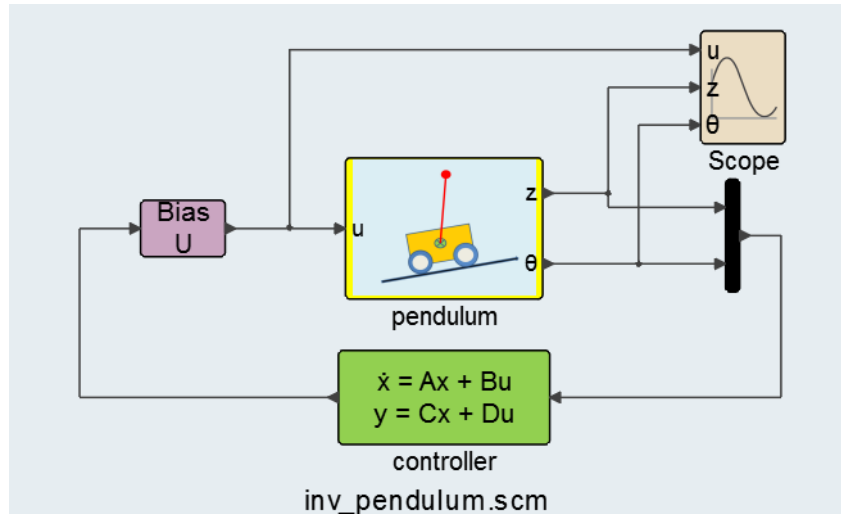
Figure 7.23: Inverted Pendulum model (inv_pendulum.scm).
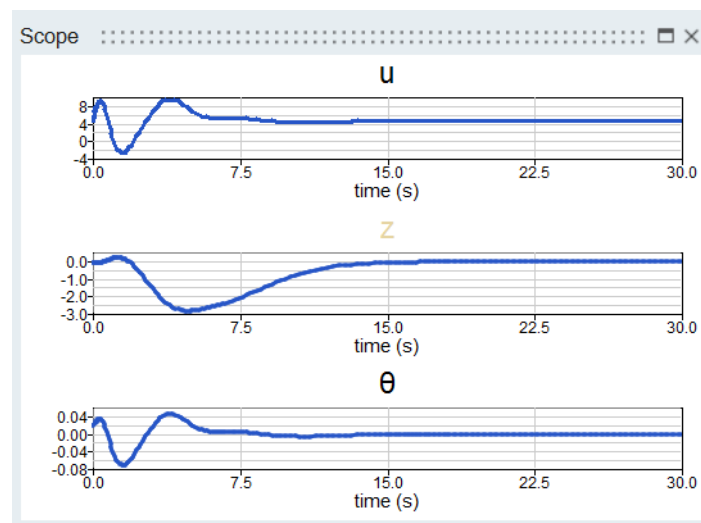
Figure 7.24: Simulation result for (inv_pendulum.scm). Note that z converges to zero.

7.2.3   3D model of a spacecraft for take off and landing control

The model of the spacecraft is made of two bodies connected by a universal joint. The distance from the joint to the center of mass of the top body, (x_0, y_0, z_0), is denoted l_0, and the distance to the bottom body's center of mass, (x_1, y_2, z_3), is dentoed l_1. The top body is the main body of the spacecraft. It has mass m_0 with moments of inertia along the main axes I_{0z} and $I_{0x} = I_{0y}$. The propulsion system is lodged in the second body, which has mass m_1 and inertia I_{1z} and $I_{1x} = I_{1y}$.

The coordinate system used to describe the dynamics of the system is attached to the universal joint. Its center is denoted (x, y, z), the joint axis connected to the bottom body is the x axis of the system and the joint axis connected to the top body is the y axis of the system. The angle between the z axis and the main axis of the top body is denoted α and with the bottom body, β. The two bodies are aligned when α and β are zero.

The spacecraft is controlled via the propulsion force F applied orthogonal to the bottom body, at its

center, and two torques to modify the angles α and β at the universal joint. The torques can be produced by motors mounted on the bodies and acting (for example through a belt system) on the universal joint.

Euler angles   The orientation of the coordinate system attached to the universal joint can be expressed by Euler angles. There are various ways to define Euler angles corresponding to the axis choices and orders in which rotations are applied. The commonly used choice ZXZ (rotate ϕ around Z, then θ around Y and finally ψ around Z) is not appropriate here because it has a singularity corresponding to the vertical position, which obviously is particularly interesting for the spacecraft.

Here the XYZ choice is used. The corresponding rotation matrices are:

$$R_x(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{pmatrix}, \; R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}, \; R_z(\psi) = \begin{pmatrix} \cos\psi & \sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

resulting in the transformation matrix

$$R_z(\psi)R_y(\theta)R_x(\phi) = \begin{pmatrix} \cos\theta\cos\psi & \cos\theta\sin\psi & -\sin\theta \\ \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi & \sin\phi\cos\theta \\ \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi & \cos\phi\cos\theta \end{pmatrix}.$$

Equations of motion   The quations are obtained from the Euler-Lagrange equation and the principle of virtual work. The Lagrangian L is defined as

$$L = T - V$$

where T denotes the kinetic energy of the system and V its potential energy.

The potential energy of the system is straightforward to compute:

$$V = m_0 z_0 + m_1 z_1.$$

Note that the universal joint is assumed to have zero mass. The kinetic energy is made of rotational and linear energies. The angular velocities Ω_0 and Ω_1 of the two bodies can be computed as follows:

$$\Omega_0 = R_y(\alpha)\left(\begin{pmatrix} 0 \\ \dot\alpha \\ 0 \end{pmatrix} + R_z(\psi)\left(\begin{pmatrix} 0 \\ 0 \\ \dot\psi \end{pmatrix} + R_y(\theta)\left(\begin{pmatrix} 0 \\ \dot\theta \\ 0 \end{pmatrix} + R_x(\phi)\left(\begin{pmatrix} \dot\phi \\ 0 \\ 0 \end{pmatrix}\right)\right)\right)\right),$$

$$\Omega_1 = R_x(\beta)\left(\begin{pmatrix} 0 \\ \dot\beta \\ 0 \end{pmatrix} + R_z(\psi)\left(\begin{pmatrix} 0 \\ 0 \\ \dot\psi \end{pmatrix} + R_y(\theta)\left(\begin{pmatrix} 0 \\ \dot\theta \\ 0 \end{pmatrix} + R_x(\phi)\left(\begin{pmatrix} \dot\phi \\ 0 \\ 0 \end{pmatrix}\right)\right)\right)\right)$$

The kinetic energy of the system given by

$$T = \frac{1}{2}(\Omega_0^T I_0 \Omega_0 + \Omega_1^T I_1 \Omega_1 + m_0(\dot x_0^2 + \dot y_0^2 + \dot z_0^2) + m_1(\dot x_1^2 + \dot y_1^2 + \dot z_1^2))$$

where the inerita matrices are defined as follows

$$I_i = \begin{pmatrix} I_{ix} & 0 & 0 \\ 0 & I_{iy} & 0 \\ 0 & 0 & I_{iz} \end{pmatrix}$$

$i = 0, 1$.

The positions of the centers of masses of the two bodies can be expressed as follows

$$\begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} - l_0 N_0$$

$$\begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} + l_1 N_1$$

where N_i's represent the main body axes of the two bodies:

$$N_0 = R_x(\beta) R_z(\psi) R_y(\theta) R_x(\phi) \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

and

$$N_1 = R_y(\alpha) R_z(\psi) R_y(\theta) R_x(\phi) \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

Thanks to these relations, the system Lagrangian can be expressed in terms of the system state variables $q = (x, y, z, \phi, \theta, \psi, \alpha, \beta)$, $q_d = (x, y, z, \dot{\phi}, \dot{\theta}, \dot{\psi}, \dot{\alpha}, \dot{\beta})$; see Appendix.

The virtual work associated with the torques σ and τ and the propulsive force F can be expressed as

$$W = \int F \left(N_1^T \begin{pmatrix} dx \\ dy \\ dz \end{pmatrix} \right) + \sigma d\alpha + \tau d\beta.$$

The equations of motion are then obtained from the Euler-Lagrange equation:

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{x}} - \frac{\partial L}{\partial x} = F(-\sin\theta\cos\phi\cos\psi + \sin\phi\sin\psi)$$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{y}} - \frac{\partial L}{\partial y} = F((\cos\psi\sin\phi + \sin\psi\sin\theta\cos\phi)\cos\beta + \sin\beta\cos\phi\cos\theta)$$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{z}} - \frac{\partial L}{\partial z} = F((\cos\psi\sin\phi + \sin\psi\sin\theta\cos\phi)\cos\beta + \sin\beta\cos\phi\cos\theta)$$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\phi}} - \frac{\partial L}{\partial \phi} = 0$$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\theta}} - \frac{\partial L}{\partial \theta} = 0$$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\psi}} - \frac{\partial L}{\partial \psi} = 0$$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\alpha}} - \frac{\partial L}{\partial \alpha} = \sigma$$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\beta}} - \frac{\partial L}{\partial \beta} = \tau$$

leading to the following second order system

$$M(q)\dot{q}_d + C(q)\dot{q}^2 + R(q, q_d) = U(q, u) \tag{7.1}$$

where u is the control input

$$u = \begin{pmatrix} F \\ \sigma \\ \tau \end{pmatrix}.$$

The expression involved in (7.1) are too complex to be expressed here (hundreds of lines of formulae). They are obtained using the symbolic computational software Maple, which is also used to generate automatically the associated **OML** codes to be used in **Twin Activate** for simulation. They are also used to obtained a linear system by linearization.

7.3   Modeling in Twin Activate

The system states are modeled by two integrator blocks representing q and q_d. The two integrator blocks are used to implement the dynamics as illustrated in Figure 7.25, the highlighted section.
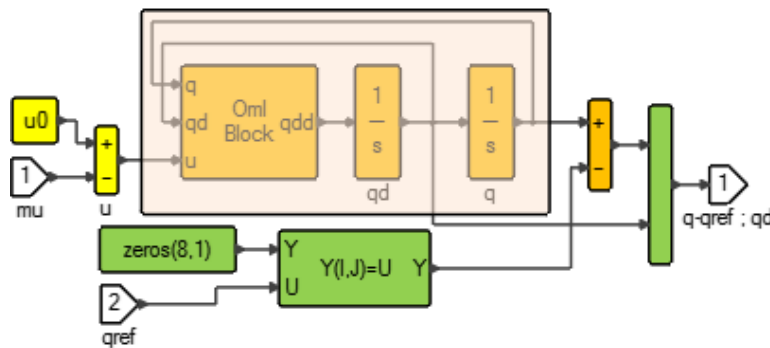
Figure 7.25: **Twin Activate** diagram containing the dynamics of the spacecraft. The system equations are expressed in the **OML** Custom block.

The integrator blocks in this case realize:

$$
\begin{align}
\dot{q} &= q_d \tag{7.2} \\
\dot{q}_d &= M^{-1}(q)(-C(q)q_d^2 - R(q, q_d) + U(q, u)). \tag{7.3}
\end{align}
$$

The expression on the right hand side of (7.3) is realized using an **OML** Custom block. The **OML** codes associated with matrices M, C, and vectors R and U, generated by Maple, are called in this **OML** Custom block.

Using **OML** at run time to perform simulation in this case leads to long simulation times. Maple can be used to generate C code instead and a C Custom block may be used instead for faster simulation if performance becomes an issue.

System linearization   A linear model approximating the behavior of the system around the stationary states of the spacecraft is sought. The stationary points of interest correspond to the state where the spacecraft is pointing straight up (with the two bodies aligned) and not moving. Since the position of the spacecraft, in particular variables x, y and z do not appear explicitly in the system equations[1],

---

[1]The spacecraft dynamics does not depend on the position of spacecraft in space, in this model.

the linearization can be performed around $q = 0$, $q_d = 0$ and $u = u_0$ where the propulsive force $F_0 = (m_0 + m_1)g$ compensates the force of gravity:

$$u_0 = \begin{pmatrix} F_0 \\ 0 \\ 0 \end{pmatrix}.$$

The approximate linear model resulting from the linearization of the nonlinear system (7.2)-(7.3) can be numerically constructed in **OML** using the **OML** code representing the dynamics of the system. But it can also be constructed using the linearization function `rock3dOML_lin\Spacecraft`.

$$\dot{\xi} = A\xi + B\mu \tag{7.4}$$

where $\mu = u - u0$ and

$$\xi = \begin{pmatrix} q \\ q_d \end{pmatrix} - \begin{pmatrix} x^* \\ y^* \\ z^* \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{7.5}$$

7.4   State feedback control

The assumption here is that the controller has access to all the states of the system q and q_d. A constant state feedback controller is sought. Unfortunately the linear model (7.4) is not controllable. The pair (A, B) controllable subspace has dimension 14 (where the system state has dimension 16). The uncontrollable modes are zero and seem to correspond to the spacecraft spin around its main axis. This spin cannot be controlled by linear control, so brought back to zero if perturbed, but this may not be an issue, especially for a short flight.

For a controllable pair (A, B), it is always possible to find a matrix K such that all eigenvalues of the matrix $A - BK$ have strictly negative real parts. Such a K then used as state feedback would make the closed-loop system asymptotically stable. In this case (A, B) not being controllable, the best that can be achieved is to find K such that all the controllable modes are moved so that they ave negative real parts but the uncontrollable modes would remain in place. To find K in this case, the system is decomposed into controllable uncontrollable parts and a state feedback matrix K_1 is found for the controllable part, which is then used to construct K. In particular an orthogonal matrix U is used as a change of coordinate matrix such that

$$\begin{aligned} U^T A U &= \begin{pmatrix} A_1 & A_2 \\ 0 & A_4 \end{pmatrix} \\ U^T B &= \begin{pmatrix} B_1 \\ 0 \end{pmatrix} \end{aligned}$$

where (A_1, B_1) is controllable. Then K_1 is chosen such that the matrix $A_1 - B_1 K_1$ has eigenvalues with strictly negative real parts. Finally K is constructed as follows

$$K = \begin{pmatrix} K_1 & 0 \end{pmatrix} U^T. \tag{7.6}$$

The **OML** code implementing this technique and pole placement for the construction of the feedback K_1, placed in the Initialization script of the model, is given below:

```
path=bdeGetModelFilePath(bdeGetCurrentModel());
path=fileparts(path);
addpath(path);

l0=6;l1=5;m0=2;m1=10;g=10;I0x=60;I0z=1;I1x=40;I1z=20;

St=[0,15,15,30,30,50,100];
Sp=[0,0,1,1,1,1,1;0,0,-1,-1,-1,-1,-1;1,1,1,1,0,0,0];

q0=zeros(8,1);X0=[q0;0*q0];
u0=[g*(m0+m1);0;0];
if ~vssIsInLinearization
  ctx=struct; ctx.K=zeros(3,16);
  model=bdeGetCurrentModel;
  [A,B,C,D]=vssLinearizeSuperBlock(model,...
  'Spacecraft',[1],[1],0,ctx);
  A=C*A*inv(C);B=C*B;C=C*inv(C);
  B=-B;
  %[A,B]=fb(X0,u0,[l0;l1;m0;m1;g;I0x;I0z;I1x;I1z]);
  [nc,U]=contr(A,B);

  A1=U'*A*U;  B1=U'*B;
  nc=nc(1);A1=A1([1:nc],[1:nc]);
  B1=B1([1:nc],:);
  K1=place(A1,B1,-ones(nc,1)*2);
  K=[K1,zeros(3,16-nc)]*U';
end
```

The controllable modes are moved to -2.

Linear state feedback K is used as follows:

$$u = u_0 + K \left(\begin{pmatrix} q \\ q_d \end{pmatrix} - X_{\text{ref}} \right)$$

with $X_{\text{ref}} = (x^*, y^*, z^*, 0, \cdots, 0)$ where (x^*, y^*, z^*) designates the position where the spacecraft is to be stabilized. The reference signals are used to modify the spacecraft model such that the inputs and outputs are zero at equilibrium point.

The top level diagram of the model is illustrated in Figure 7.26. It shows the implementation of the control law.
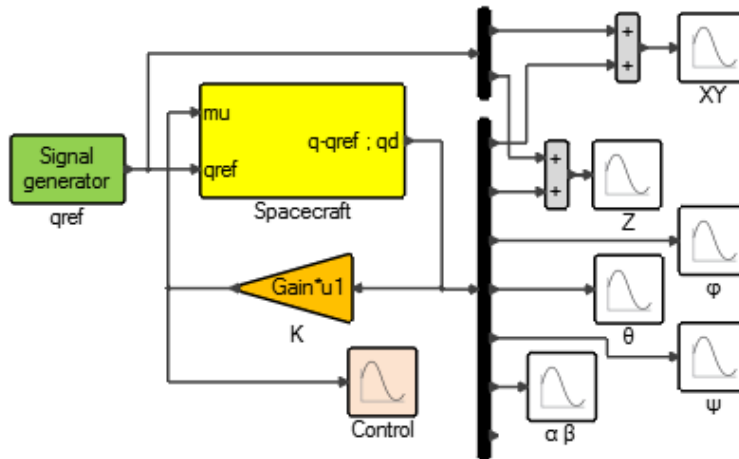
Figure 7.26: **Twin Activate** top diagram shows the complete model.

Simulation results The performance of the control law is assessed by running simulations. In the following test simulation, the reference position (x^*, y^*, z^*) is assumed to be initially $(0, 0, 1)$ then at $t = 15$, $(1, -1, 1)$ followed by $(1, -1, 0)$ at $t = 30$. The spacecraft is initially at point $(0, 0, 0)$ so the objective is to lift the spacecraft up one meter, then move it sideways and then land it.

The simulation results are shown in Figures through .
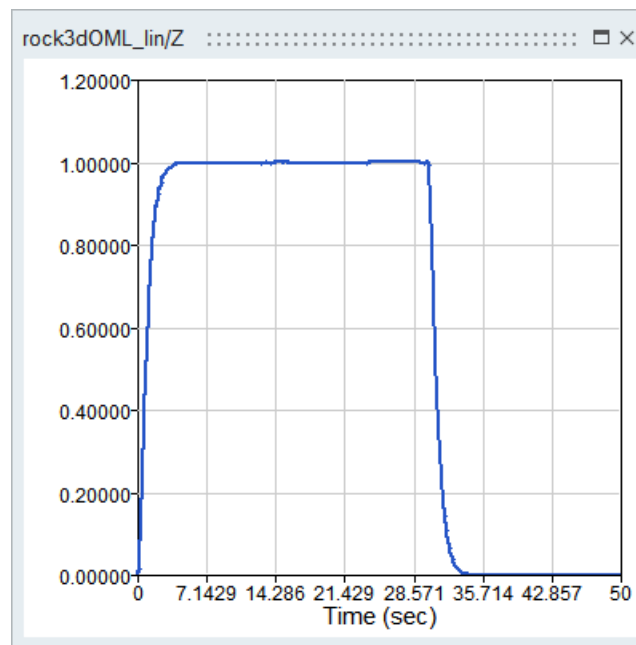
Figure 7.27: The altitude z follows the reference trajectory by going from 0 to 1 and then back to 0.
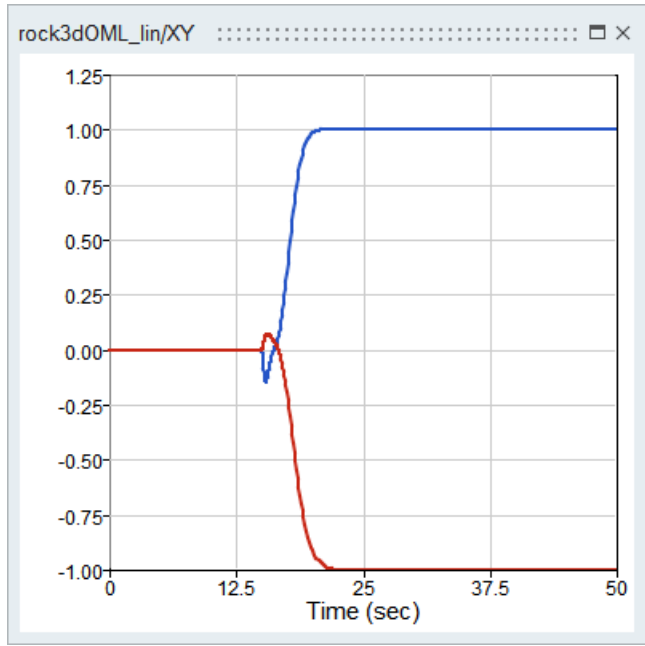
Figure 7.28: The XY position of the spacecraft follows the reference by going from $(0, 0)$ to $(-1, 1)$.
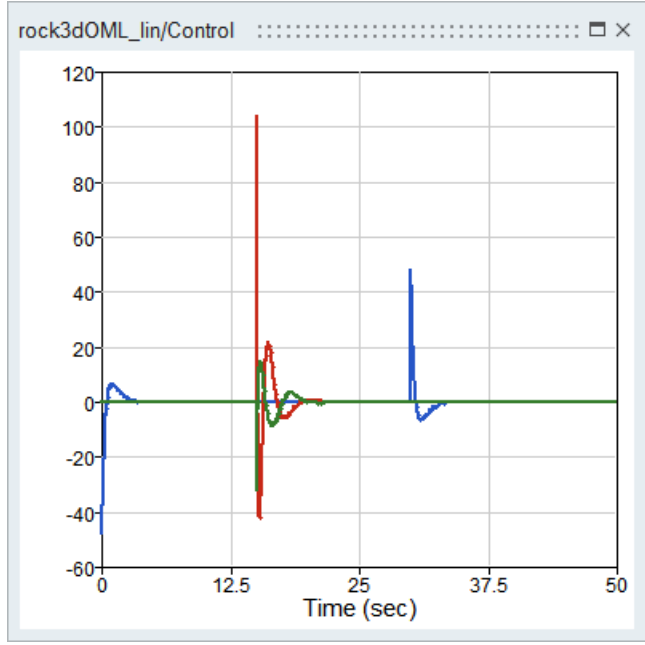
Figure 7.29: The control signals are illustrated here: $F - F_0$, σ and τ.
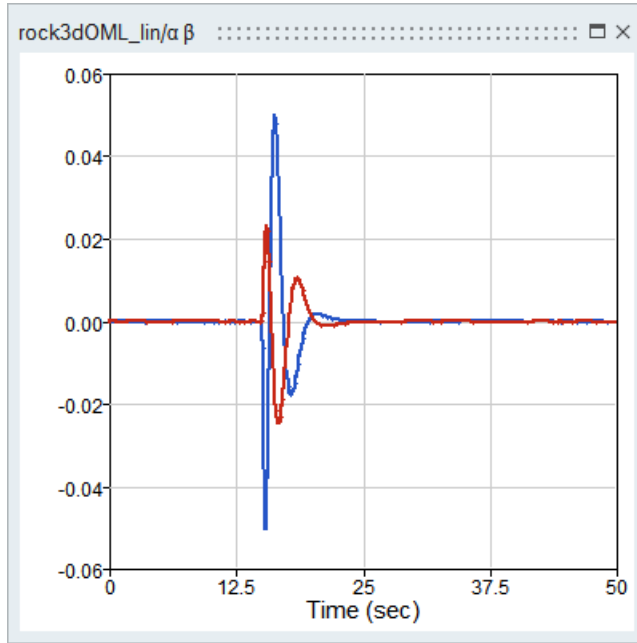
Figure 7.30: The angles α and β corresponding to the universal joint.
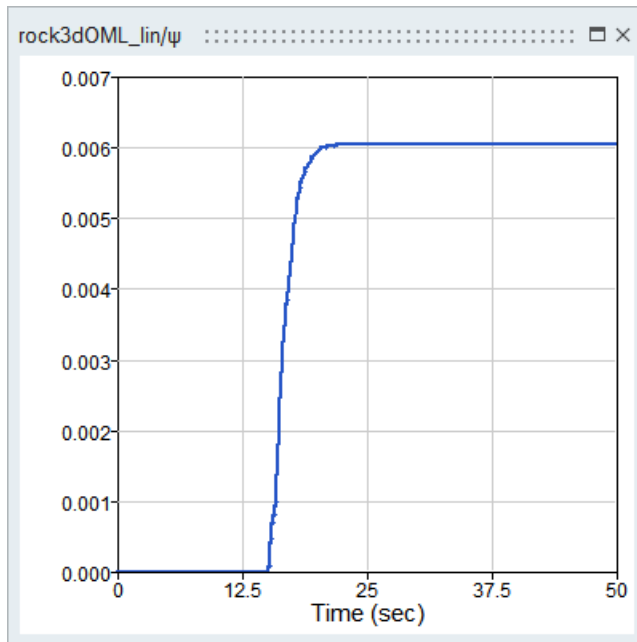
Figure 7.31: The simulation shows a drift in the ψ signal, which is not controllable and thus cannot be brought to zero. This however is not an issue.

Appendix

$$
\begin{aligned}
L \;=\; & \Big(\Big(\Big(-0.5\,\dot\alpha^2\,l_1{}^2\,m_1 + 0.5\,\dot\beta^2\,l_0{}^2\,m_0\Big)\cos(\phi)^2 + (0.5\,I_{0x} - 0.5\,I_{0z})\,\dot\phi^2\,\cos(\alpha)^2 + (0.5\,I_{1z} \\
& - 0.5\,I_{1x})\,\dot\phi^2\,\cos(\beta)^2 + (-0.5\,I_{0x} + 0.5\,I_{0z} - 0.5\,I_{1z} + 0.5\,I_{1x})\,\dot\phi^2\Big)\cos(\theta)^2 + \Big(\dot\alpha^2\,l_1{}^2 \\
& m_1 - \dot\beta^2\,l_0{}^2\,m_0\Big)\cos(\phi)^2 + (-0.5\,I_{0x} + 0.5\,I_{0z})\,\dot\theta^2\,\cos(\alpha)^2 + (-0.5\,I_{1z} + 0.5\,I_{1x}) \\
& \dot\theta^2\,\cos(\beta)^2 - 0.5\,\dot\alpha^2\,l_1{}^2\,m_1 + (-0.5\,I_{0z} - 0.5\,I_{1x} + 0.5\,I_{1z} + 0.5\,I_{0x})\,\dot\theta^2 \\
& + 0.5\,\dot\beta^2\,l_0{}^2\,m_0\Big)\cos(\psi)^2 + \Big(\Big(\cos(\phi)^2\,\sin(\theta)\,\dot\beta\,l_0{}^2\,m_0\,\dot\psi + \Big(-\cos(\alpha)\,l_1\,m_1\,\dot\theta \\
& \dot x - \sin(\alpha)\,l_1\,m_1\,\dot\theta\,\dot z - \sin(\phi)\,\dot\alpha\,l_1{}^2\,m_1\,\dot\psi + l_0\,m_0\,\dot\theta\,\dot x\Big) \\
& \cos(\phi) + \Big((I_{0x} - I_{0z})\,\dot\theta\,\dot\phi\,\cos(\alpha)^2 + (-I_{1x} + I_{1z})\,\dot\theta\,\dot\phi\,\cos(\beta)^2 \\
& + (-I_{1z} + I_{0z} - I_{0x} + I_{1x})\,\dot\theta\,\dot\phi\Big)\sin(\psi) + (I_{0z} - I_{0x})\,\dot\phi^2\,\sin(\alpha)\,\cos(\alpha) \\
& \sin(\theta) + (I_{0z} - I_{0x})\,\dot\psi\,\dot\phi\,\sin(\alpha)\,\cos(\alpha) + \dot\phi\,\dot\beta\,\Big(l_0{}^2\,m_0 + I_{1x}\Big)\Big) \\
& \cos(\theta) + \cos(\phi)^2\,\dot\alpha\,l_1{}^2\,m_1\,\dot\theta + \Big(\Big(-\dot\alpha^2\,l_1{}^2\,m_1 + \dot\beta^2\,l_0{}^2\,m_0\Big)\sin(\phi) \\
& \sin(\theta)\,\sin(\psi) + \Big(-\cos(\alpha)\,\dot\alpha\,l_1\,m_1\,\dot z - \cos(\beta)\,l_0\,m_0\,\dot\psi\,\dot y + \sin(\phi) \\
& \dot\beta\,l_0{}^2\,m_0\,\dot\theta + m_1\,l_1\,(\dot x\,\dot\alpha + g)\,\sin(\alpha) + \dot\psi\,(\dot z\,l_0\,m_0\,\sin(\beta) + m_1 \\
& \dot y\,l_1)\Big)\sin(\theta) - \cos(\beta)\,l_0\,m_0\,\dot\phi\,\dot y + \dot\phi\,(\dot z\,l_0\,m_0\,\sin(\beta) + m_1\,\dot y\,l_1)\Big) \\
& \cos(\phi) + \Big(\cos(\alpha)\,\sin(\phi)\,l_1\,m_1\,\dot\phi\,\dot x + (I_{1x} - I_{1z})\,\dot\theta\,\dot\phi\,\sin(\beta)\,\cos(\beta) \\
& + \Big(\sin(\alpha)\,l_1\,m_1\,\dot\phi\,\dot z - l_0\,m_0\,\dot\phi\,\dot x\Big)\sin(\phi)\Big)\sin(\theta) + \cos(\alpha)\,\sin(\phi)\,l_1\,m_1 \\
& \dot\psi\,\dot x + \Big(\sin(\phi)\,\dot\beta\,l_0\,m_0\,\dot z + (I_{1x} - I_{1z})\,\dot\psi\,\dot\theta\,\sin(\beta)\Big)\cos(\beta) \\
& + \Big(\sin(\alpha)\,l_1\,m_1\,\dot\psi\,\dot z + (-g + \dot\beta\,\dot y)\,l_0\,m_0\,\sin(\beta) - l_0\,m_0\,\dot\psi\,\dot x\Big)\sin(\phi) \\
& + I_{0x}\,\dot\alpha\,\dot\theta\Big)\cos(\psi) + \Big(\Big(\Big(-0.5\,\dot\psi^2 + 0.5\,\dot\alpha^2\Big)l_1{}^2\,m_1 - 0.5\,l_0{}^2\,m_0\,\dot\psi^2\Big) \\
& \cos(\phi)^2 + (0.5\,I_{0x} - 0.5\,I_{0z})\,\dot\phi^2\,\cos(\alpha)^2 + (I_{1x} - I_{1z})\,\dot\phi^2\,\cos(\beta)^2 + (0.5\,I_{1z} - 0.5\,I_{1x}) \\
& \dot\phi^2\Big)\cos(\theta)^2 + \Big(-\sin(\psi)\,\cos(\phi)^2\,\sin(\theta)\,\dot\alpha\,l_1{}^2\,m_1\,\dot\psi + ((-\cos(\beta)\,l_0\,m_0 \\
& \dot\theta\,\dot y - \sin(\phi)\,\dot\beta\,l_0{}^2\,m_0\,\dot\psi + \dot\theta\,(\dot z\,l_0\,m_0\,\sin(\beta) + m_1\,\dot y\,l_1)\Big) \\
& \sin(\psi) + (-g - \dot x\,\dot\alpha)\,l_1\,m_1\,\cos(\alpha) + (g - \dot\beta\,\dot y)\,l_0\,m_0\,\cos(\beta) + \Big(-l_1{}^2\,m_1 \\
& \dot\psi\,\dot\theta - l_0{}^2\,m_0\,\dot\psi\,\dot\theta\Big)\sin(\phi) + \sin(\beta)\,\dot\beta\,l_0\,m_0\,\dot z - \sin(\alpha) \\
& \dot\alpha\,l_1\,m_1\,\dot z)\cos(\phi) + \Big((-I_{1x} + I_{1z})\,\dot\phi^2\,\sin(\beta)\,\cos(\beta)\,\sin(\theta) + (-I_{1x} + I_{1z}) \\
& \dot\psi\,\dot\phi\,\sin(\beta)\,\cos(\beta) + \Big(-\dot\alpha\,l_1{}^2\,m_1 - I_{0x}\,\dot\alpha\Big)\dot\phi\Big)\sin(\psi) \\
& - \cos(\alpha)\,\sin(\phi)\,l_1\,m_1\,\dot\phi\,\dot z + \dot\phi\,\sin(\phi)\,(m_1\,\dot x\,\sin(\alpha)\,l_1 + \dot y\,\sin(\beta)\,l_0\,m_0 \\
& + \dot z\,l_0\,m_0\,\cos(\beta)))\cos(\theta) + \Big(\sin(\psi)\,\dot\beta\,l_0{}^2\,m_0\,\dot\theta + \Big(0.5\,l_1{}^2\,\dot\theta^2 \\
& - 0.5\,\dot\alpha^2\,l_1{}^2\Big)m_1 + 0.5\,\dot\beta^2\,l_0{}^2\,m_0 + 0.5\,l_0{}^2\,m_0\,\dot\theta^2\Big)\cos(\phi)^2 + \Big(\Big(\Big(\cos(\alpha)\,l_1\,m_1\,\dot\psi \\
& \dot x + \cos(\beta)\,\dot\beta\,l_0\,m_0\,\dot z - \sin(\phi)\,\dot\alpha\,l_1{}^2\,m_1\,\dot\theta + \sin(\alpha)\,l_1\,m_1\,\dot\psi \\
& \dot z + (-g + \dot\beta\,\dot y)\,l_0\,m_0\,\sin(\beta) - l_0\,m_0\,\dot\psi\,\dot x\Big)\sin(\theta) - l_0\,m_0\,\dot\phi\,\dot x \\
& + \cos(\alpha)\,l_1\,m_1\,\dot\phi\,\dot x + \sin(\alpha)\,l_1\,m_1\,\dot\phi\,\dot z\Big)\sin(\psi) + \Big(-\cos(\alpha)\,l_1\,m_1\,\dot\theta \\
& \dot z + \dot\theta\,(m_1\,\dot x\,\sin(\alpha)\,l_1 + \dot y\,\sin(\beta)\,l_0\,m_0 + \dot z\,l_0\,m_0\,\cos(\beta))\Big)\sin(\theta)\Big) \\
& \cos(\phi) + \Big(\Big((I_{0z} - I_{0x})\,\dot\theta\,\dot\phi\,\sin(\alpha)\,\cos(\alpha) + \cos(\beta)\,\sin(\phi)\,l_0\,m_0\,\dot\phi \\
& \dot y + \Big(-l_1\,m_1\,\dot\phi\,\dot y - \sin(\beta)\,l_0\,m_0\,\dot\phi\,\dot z\Big)\sin(\phi)\Big)\sin(\theta) + (\sin(\phi) \\
& \dot\alpha\,l_1\,m_1\,\dot z + (I_{0z} - I_{0x})\,\dot\psi\,\dot\theta\,\sin(\alpha)\Big)\cos(\alpha) + \cos(\beta)\,\sin(\phi)\,l_0\,m_0 \\
& \dot\psi\,\dot y + \Big((-g - \dot x\,\dot\alpha)\,l_1\,m_1\,\sin(\alpha) - l_1\,m_1\,\dot\psi\,\dot y - \sin(\beta)\,l_0\,m_0 \\
& \dot\psi\,\dot z\Big)\sin(\phi) + I_{1x}\,\dot\beta\,\dot\theta\Big)\sin(\psi) + \Big((I_{0z} - I_{0x})\,\dot\psi\,\dot\phi \\
& \cos(\alpha)^2 + (-I_{1x} + I_{1z})\,\dot\psi\,\dot\phi\,\cos(\beta)^2 + \dot\phi\,\dot\psi\,\Big(l_0{}^2\,m_0 + l_1{}^2\,m_1 + I_{0x} + I_{1x}\Big)\Big) \\
& \sin(\theta) + \Big((-0.5\,I_{0x} + 0.5\,I_{0z})\,\dot\phi^2 + (0.5\,I_{0x} - 0.5\,I_{0z})\,\dot\theta^2 + (-0.5\,I_{0x} + 0.5\,I_{0z}) \\
& \dot\psi^2\Big)\cos(\alpha)^2 + \Big((0.5\,I_{1z} - 0.5\,I_{1x})\,\dot\phi^2 + (0.5\,I_{1z} - 0.5\,I_{1x})\,\dot\psi^2\Big)\cos(\beta)^2 \\
& + \Big(0.5\,l_0{}^2\,m_0 + 0.5\,l_1{}^2\,m_1 + 0.5\,I_{0x} + 0.5\,I_{1x}\Big)\dot\phi^2 + \Big(\Big(0.5\,\dot\psi^2 + 0.5\,\dot\alpha^2\Big)l_1{}^2 - g\,z + 0.5\,\dot x^2 \\
& + 0.5\,\dot y^2 + 0.5\,\dot z^2\Big)m_1 + (0.5\,I_{0z} + 0.5\,I_{1x})\,\dot\theta^2 + \Big(0.5\,l_0{}^2\,\dot\psi^2 - g\,z + 0.5\,\dot x^2 + 0.5\,\dot y^2 \\
& + 0.5\,\dot z^2\Big)m_0 + (0.5\,I_{0x} + 0.5\,I_{1x})\,\dot\psi^2 + 0.5\,I_{0x}\,\dot\alpha^2 + 0.5\,I_{1x}.\,\dot\beta^2
\end{aligned}
$$

Chapter 8

Twin Activate Libraries

In **Twin Activate**, libraries are self-contained repositories that essentially provide the application with new blocks. But, beyond just blocks, a library provides also (**OML**) functions that the user can use within his scripts.

8.1   Library structure

A library in **Twin Activate** can be described by the following structure:
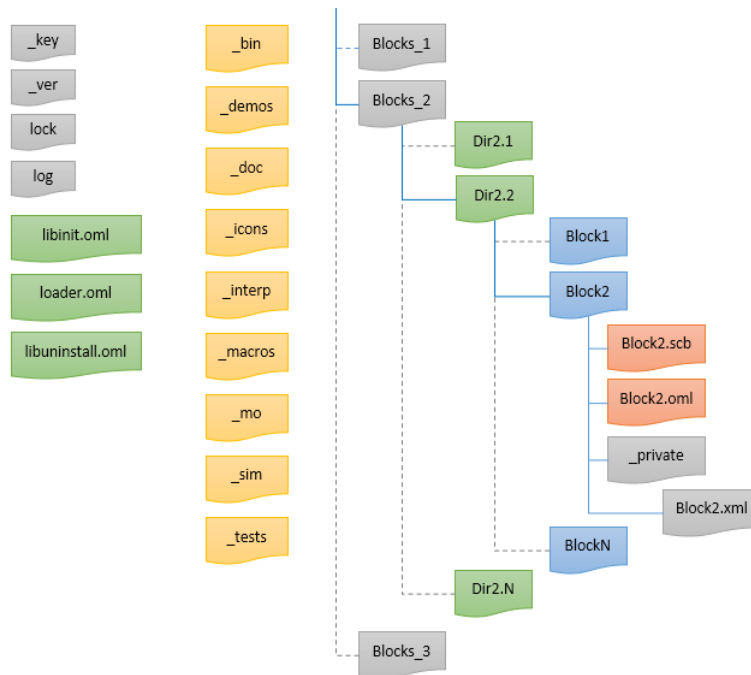
Figure 8.1: Representation of a typical library in **Twin Activate**

- key: The library is identified by its key. The key is generated automatically by **Twin Activate** in the file _key. This file is updated every time the library is edited after a release; a new version of the library is created. The first line of the _key file defines the current version ID, the other lines

defines the IDs of the old versions. This file should not be manually edited, as this could alter the library.

- loader: The blocks of the library are loaded into **Twin Activate** through the loader.oml file. This file is automatically generated, it must not be edited.

- libinit: The library can add to the **Twin Activate** environment palettes, menus and functions. The user can edit the libinit.oml file to perform such actions.

- palette(s): The palettes of the library can be any scm file loaded as palette in the libinit file. The palette browser shows only the blocks. The links and annotations are discarded.

- block : A block in **Twin Activate** is defined by the interface file (**SCB**) and the simulation functions. The block interface files can be organized hierarchically under the condition that the folder that contains the **SCB** file must have the same name as the **SCB**. The Hierarchical path defines the reference path of the block, it is relative to the library top level. The simulation functions of the block must be inside the _sim folder of the library. The relative paths of the simulation functions must be equivalent to the ones defined in the **SCB** file.

- icons: Icons of the blocks should be defined inside the _icons folder of the library.

- documentation: The block documentation is generated inside _doc folder.

The image below can summarize the library structure:

DLLs needed for the _sim DLLs_____	_bin	
Demos models coming with the library *(optional)*_____	_demos	
Documentation of the blocks (XML and HTML)_____	_doc	
Images for the block icons (mostly SVG)_____	_icons	
Scripts (OML, Python) used in the blocks_____	_interp	⊿ _sim
Macros and scripts used in the blocks_____	_macros	blocks_1
Support for Physical Component Library *(optional)*___	_mo	blocks_2
Simulation routines' DLLs_____	_sim	blocks_3
Tests used to validate the library *(optional)*_____	_tests	
Some blocks hierarchy_____	Blocks_1	
Some blocks hierarchy_____	Blocks_2	
Some blocks hierarchy_____	Blocks_3	
Key file (unique, auto-generated)_____	_key	
Script created by library creator_____	libinit.oml	*to load OML functions needed by the library*
Palette created by library creator_____	library_name.scm	*palette (model) file used in library browser*
Image to use on the palette icon_____	library_name.svg	*palette image icon used in library browser*
Script created by library creator _____	libuninstall.oml	*to uninstall the library*
Script created by library creator _____	loader.oml	*to load the library at start-up*

Figure 8.2: Description of a library folder in **Twin Activate**

8.2   Block structure

In a **Twin Activate** library a block is defined by several files.

- **SCB**: The file describes the graphics part of the block. It contains also the parameters definition and the evaluation part of the block. This file is generated by selecting a block and using the menu **File/Save Selected**. The block can also be a masked Super Block, or an atom block. Creating an atom block can be done through the menu **Tools/Block Builder**.

- **OML**: The file is generated from the **SCB**, it should not be edited. It is used during the evaluation phase of the simulation.

- **XML**: The file is used for the documentation, the skeleton of that file is generated automatically from the **SCB** file. This file should be completed by the block creator.

- **HTML**: The file is generated from the **XML** file, it is pushed under _doc folder.

As an example, the image below illustrates the structure of the constant block in the **Twin Activate** library.
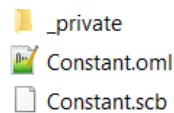
Figure 8.3: Constant block folder content

A block can have one or many simulation functions, they are considered to be C entry point functions. They are defined under the _sim folder of the library. A library creator can decide to either ship the C code or not, but it is mandatory to ship the static and the shared libraries. If a simulation function depends on a third party library, the shared and static library of that third party must be delivered under the _bin folder of the library.
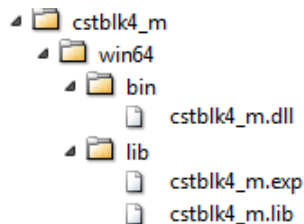
Figure 8.4: Constant block: View of the simulation folder content

8.3 Palette

The palette is the way to expose the blocks of a library to the user. It is very similar to a model. It can be edited by **Twin Activate** as any other model. A non masked Super Block in a palette is used to create hierarchy in the palette browser. A masked Super Block is seen in the palette browser as a block. Since the **B**lock **D**iagram **E**ditor manipulates palettes as models, the extension of the palette file is **scm**.

A library can have several palettes, and in contrast, some libraries may not need to expose any block, in which case the library does not have any palette.

Since the palettes are the interface of the library, the user can choose the blocks to be exposed. Some of the blocks that are considered as "support" blocks, may not be place into a palette. However, users can still access such blocks through the menu **File/Libraries/Insert Block From Library**.

A palette does not need to be associated with any library. The user can create his own palette based on the existing blocks. He can, for example, create his favorite palette to host his most used blocks. In that case, the user will create a model and use the menu **File/Libraries/Save as Palette**. This palette is then loaded and exposed under the Personal page of the Library Browser.

When a library is Uninsulated, files are removed. The palettes loaded by that library should be removed from the palette browser. To remove Personal palettes, the user can use the menu **File/Libraries/Remove Palette**.

8.4 Library creation

Twin Activate offers two ways from creating and editing a library:

- Library Manager: **Twin Activate** provides a **GUI** to help the user create and edit his libraries.

- **OML** functions: **Twin Activate** provides **OML** functions to allow the user to create and edit his libraries.

Note that the Library Manager provides an easy but limited way to edit the library. For advanced users, the use of (**OML**) functions is recommended.

8.4.1 Library Manager
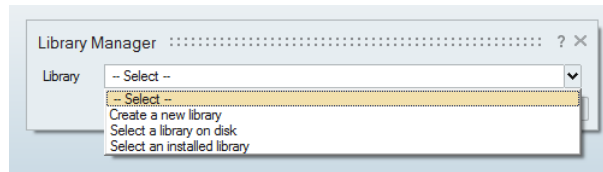
Figure 8.5: Library Manager

The Library Manager allows the user to:

- create a new library: The user must define the name and the path of the library. Note that upon creation, the library is also installed.
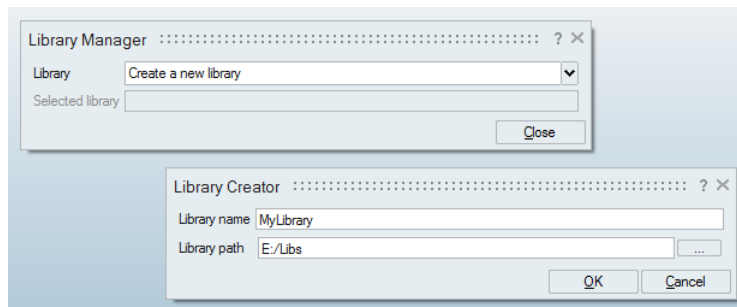
Figure 8.6: Create Library

- edit an installed library: The user can select one of the installed libraries.

- edit a library on disk: The user can select the library (a folder) on his disk. The library will be installed.

Once the library is installed, the Library Manager offers to the user the possibility to add or replace blocks in the library. The user can then select a masked Super Block and click the button **Add/Replace**.
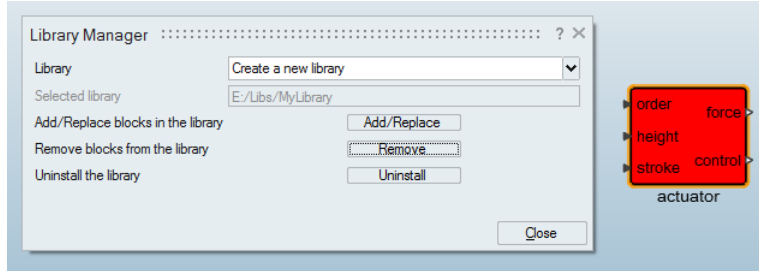
Figure 8.7: Add or Replace action in Library

The block is added to the library, the loader is updated and the palette is regenerated.
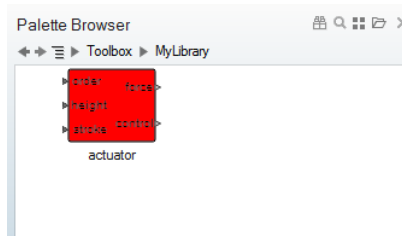
Figure 8.8: Block added to the Library

The user can also remove blocks from the library using the Library Manager.
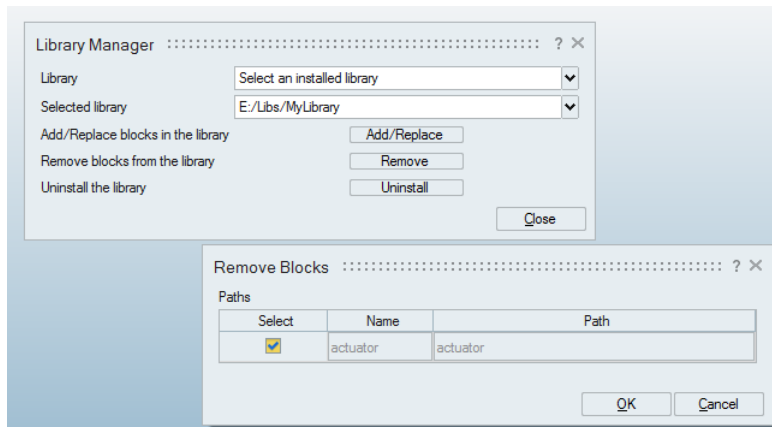
Figure 8.9: Block removal in a library

The Library Manager allows the user to lock his library. When the library is locked it is ready to be released. To do so, the user must uninstall his library through the button **Uninstall**, and then click the button **Lock**. When the library is locked, the user cannot add or remove any block from it.

The Library Manager gives the possibility to the user to edit a locked library through the button **Unlock**. This feature will create a new version of the library, a new key is created in the _key file, the blocks are updated and the palette is regenerated, it references the blocks with the new key.

```
4364825b-cfc2-48b9-b8de-5e236bee2e56
79b2cae8-905e-4851-acfa-77fe58b4ddb3
```

Figure 8.10: Example of library keys

```
<block name="actuator" type="block">
  <template name="4364825b-cfc2-48b9-b8de-5e236bee2e56/actuator"/>
```

Figure 8.11: Library key used (referenced) in a block

The models and the scripts that use the old version of the library will still work. The user must make sure that his library is backward compatible. He cannot remove any block from the library. He cannot change the hierarchical organization of the blocks. He cannot remove any exposed **OML** function. For a given block, he cannot remove any parameter.

8.4.2   OML functions

Twin Activate provides **OML** functions to manipulate explicitly the libraries.

Create library

This function creates a skeleton of the library on the disk. It generates a new key and assigns that key to the library. The arguments of that function are the name of the library and the path where the library must be created.

```
vssCreateEmptyLibrary('mylib','x:/Libraries');
```

Adding blocks

The user can use the menu **Tools/Block Builder** to create his blocks. The block will be created and inserted in the current model. To add the block to the library the user can select the created block and call the following functions:

```
blk=bdeGetSelectedBlock(bdeGetCurrentDiagram());
vssAddBlockToLibrary(blk,'x:/Libraries/mylib/Level1/MyBlock');
```

The user can also use the **B**lock **D**iagram **E**ditor APIs to create the block.

Refresh library

After populating the library with the blocks, the refresh function generates the **OML** functions used by the evaluator, the **XML** files used to generate the documentation, and the libraries used by the simulator. This function generates also the loader of the library. The argument of that function is the path of the library.

```
vssRefreshLibrary('x:/Libraries/mylib');
```

The library is now ready to be used.

Note: Every time the user adds a block to his library, he must refresh the library and execute the loader so that the new block is taken into account.

Generate palette

Twin Activate offers several methods to create a palette.

- The user can generate the palette automatically from the library; In this case the palette contains all the blocks of the library. He can then edit the palette to remove or add blocks. The palettes of a library must contain only blocks from the library or from the System (Activate) library. In this case he can use the function

    ```
    vssGeneratePalette('x:/Libraries/mylib');
    ```

- The user can generate the palette manually. In this case, he can use the **B**lock **D**iagram **E**ditor since a palette is in essence a model. Steps are: open a new model, add new block(s) to the model through the menu **File**/**Libraries**/**Insert Block From Library** and finally save the model in the library.

Add palette

Once the palettes are generated we must edit the libinit.oml file by adding the AddPalette function. That function adds the palettes of the library to the toolbox page.

```
vssAddPalette([libPath, '/mylib.scm']);
```

where libPath is the path of the library.

Note: to make the library movable, use the variable libPath defined in libinit.oml instead of hard coding the path of the library.

Install library

This function installs the library in the **Twin Activate** environment. It does not change the location of the library on the disk.

```
vssInstallLibrary('x:/Libraries/mylib');
```

The library is now installed, the palettes are added to the palette browser and the user can start using the library.

Uninstall library

The user can uninstall the library. The palettes that come with the library will be removed from the Library Browser. Once a library is uninstalled, users cannot run any model that uses blocks from that library.

```
vssUninstallLibrary('x:/Libraries/mylib');
```

Document library

The user can start documenting his blocks. He has to edit the **XML** file corresponding to each block. The fields that are already filled must not be changed. He can give for example more information about the block in the description section. Once the blocks are documented, he can use the following function to update the **HTML** pages.

```
vssGenerateDocumentation('x:/Libraries/mylib');
```

Lock library

The library is now ready to be delivered. The user **must** lock the library before releasing. Once locked user cannot add any block or function to the library. The library is now read-only.

```
vssLockLibrary('x:/Libraries/mylib');
```

Unlock library

As a library creator, user can still edit his library after it has been locked. In this case he must, first, unlock the library.

A new key is generated indicating that this is a new version of the library. The new version of the library must be backward compatible. He cannot remove any block from the library. He cannot change the hierarchical level of the blocks. He cannot remove any exposed **OML** function. For a given block, he cannot remove any parameter.

```
vssUnlockLibrary('x:/Libraries/mylib');
```

Chapter 9

Twin Activate blocks

A **Twin Activate** Model is an interconnection of blocks. Most **Twin Activate** users do not need to develop new blocks because **Twin Activate** provides a large number of blocks through different palettes. Advanced users however may need to develop new specialized blocks. These blocks may be placed in new libraries and distributed to other users.

Blocks in **Twin Activate** may be constructed based on other blocks. A masked Super Block for example behaves almost like any other block in the Block Diagram Editor (BDE); the only difference is that the diagram inside the Super Block can be visualized and edited only using the "Enter Masked Super Block" contextual menu item[1].

A block may also be defined through a diagram inside, similarly to a Super Block, but without any associated graphical information. These blocks are called *programmable super blocks*. The content of the block is defined via **OML** scripts using special APIs instantiating and connecting blocks to define the inside diagram. At the BDE level, programmable super blocks are indistinguishable from basic blocks. In the absence of graphical information concerning the inside diagram, the "Enter Masked Super Block" operation is not available. In fact this operation cannot apply even if graphical information were available since the content of a programmable super block may depend on block parameters. For example a block parameter may define the number of times a structure is repeated in the diagram. The parameter values not necessarily being available in the BDE, the diagram structure is not even known.

A basic block defines its behavior directly through a simulation function. Simulation functions are defined as C functions in most cases but they can also be defined in **OML**. Simulation functions can be developed first for **CCustomBlock** or **OMLCustomBlock** and tested before being used in the definition of a new basic block.

The construction of simulation functions for basic blocks is discussed in the next section. Once the simulation function is developed, the construction of a new block is greatly simplified using the **Twin Activate** tool *Block builder* tool. Block builder is presented in Section 9.2.

9.1 Block simulation function

A block communicates with other blocks via the regular input and output ports and activation input and output ports. Signals associated with regular input and output ports can be of type matrix of real,

---

[1] In case of a library block, the block must be first inlined to enter the Super Block

complex or integer data types. The activation input and output ports, on the other hand, transmit only *events* (which are also called *activations*).

A block can have several regular and activation input and output ports. A block can also have continuous-time, discrete-time states, and zero-crossing surfaces. Of course a block does not need to have all of these elements.

The behavior of a **Twin Activate** block is basically governed by the way it is activated. The block is activated when it receives activation on its activation input ports. If the block does not have input activation, it can be activated by inheritance. In this case, the activation input ports are added by the **Twin Activate** compiler to the block. The block can also be defined as being *Always active*.

A block can be activated in many different ways. In the following a number of scenarios are presented.

Discrete-time activation, or events   When a block is activated by an event on its activation input port, it can update its regular outputs:

$$y(t_e) = f_{\text{output}}(t_e, x(t_e^-), z(t_e^-), u(t_e), n_{\text{evprt}}, m, phase, p)$$

where t_e is the activation or event time, $x(t_e^-)$ and $z(t_e^-)$ represent continuous-time and discrete-time states just before the occurrence of the event. y and u are output and input of the block. n_{evprt} is a positive integer indicating by which activation input port the block has been activated. m, $phase$, p are `mode`, `simulation phase`, and parameters of the block respectively. `mode` and `simulation phase` will be discussed later.

If the block has output activation ports, it can program events on them by defining the delay time (which can be zero) on each activation output port:

$$t_{\text{evo}} = f_{\text{event}}(t_e, x(t_e^-), z(t_e^-), u(t_e), n_{\text{evprt}}, m, phase, p).$$

t_{evo} is a vector with the same size that the number of activation output ports. Once programmed, if $t_{\text{evo}}[i]$ is positive, an event is programmed at time $t_e + t_{\text{evo}}[i]$ at activation output port $i + 1$.

The continuous-time and discrete-time states of the block, if present, can also be updated at event times.

$$[x(t_e), z(t_e)] = f_{\text{discrete}}(t_e, x(t_e^-), z(t_e^-), u(t_e), n_{\text{evprt}}, m, phase, p).$$

Continuous-time activation   If the source of the activation is *always active*, then activation occurs over time interval and not at specific time instants similar to discrete-time events. In Continuous-time activation, the output computation can be represented with

$$y(t) = f_{\text{output}}(t, x(t), z(t), u(t), m, phase, p)$$

During a continuous-time activation, no state update and no event-programming is performed. Discrete-time states z, `mode` variables and `simulation phase` stay unchanged. On the other hand, the continuous-time state follows the differential equation.

$$\dot{x}(t) = f_{\text{continuous}}(t, x(t), z(t), u(t), m, phase, p)$$

Zero-crossing function and `mode` **variables** **Twin Activate** provides several numerical solvers, including fixed step-size and variable step-size solvers such as `Lsoda` and `Radau`. Variable step-solvers adjust the integration step size during the integration to provide a faster simulation without compromising the precision in the solution.

If the differential equation represented by the diagram is not smooth enough, the variable-step solvers reduce the step-size to cope with the non-smoothness or even discontinuity. If f_output or $f_\text{continuous}$ is not smooth (continuously differentiable) at some points, then `mode` variable is used in such way as to make sure the numerical solver never encounters these discontinuity points inside the integration interval. Consider the following example:

$$y = \left\{ \begin{array}{ll} u & \text{if } u \geq 0 \\ -u & \text{otherwise} \end{array} \right.$$

This function implements the absolute value function and is not differentiable at $u = 0$. In this case, a `mode` variable can be defined to specify at the start of the integration period whether u is positive. To make sure the integration stops when sign of u changes, a zero-crossing surface is introduced at zero, here the zero-crossing surface is $u = 0$. During the integration period (which could end because of the zero-crossing), the output y is computed as follows:

$$y = \left\{ \begin{array}{ll} u & \text{if } \mathtt{m} = 1 \\ -u & \text{otherwise} \end{array} \right.$$

Once the zero-crossing is detected, the numerical solver stops and the `mode` is recomputed; then the integration continues. The computation of the `mode` and the zero-crossing surface are performed by the block. The block can be called in different situations or `phases`. In some `phases`, such as during the numerical integration, `mode` variables are fixed and in other `phases`, `modes` are updated. For example when an event activates the block, `mode` variables can be updated. The `phase` variable is not directly accessible to the user. The user can check if `mode` variables are fixed or not by using the macro `isModeFixed(block)`.

When a variable step solver integrates a model, it may happen that a time interval be repeated several times till the solver accepts the step and starts another step. In most situations users need not worry about these intermediate repeated calls, but in some cases, e.g., when the output of the block is displayed, these calls should be ignored. In order to filter these calls, the macro `isinTryPhase(block)` is used. This macro uses the `phase` variable to find out whether the call should be ignored. It is important to indicate that `mode` variables are used only when the numerical solver is variable-step.

Zero-crossing functions can also be used to generate discrete-time events. Once a zero-crossing surface function inside a block crosses zero, the block is activated with a $n_\text{evprt} = -1$, i.e., at this event time t_e, f_discrete is called with $n_\text{evprt} = -1$. This type of activation is used, e.g., in modeling a simple bouncing ball that will be presented later.

9.1.1 Simulation function implementation

The *simulation function* is normally written in `C`, but can also be written in **OML**. This function defines the behavior of the block during the simulation.

The simulation function is called by the simulator with two arguments: a `C` structure containing block information (`vss_block`) and a integer value called `flag`. The calling sequence is as follows:

```
MyNewBlock(vss_block *block, int flag)
```

The calling arguments will be discussed in the following sections.

`vss_block` **structure**

The `vss_block` structure is a hierarchical `C` structure providing access to the block parameters and variables.

```
struct vss_block {
    ....
    BlockInputX input;
    BlockOutput output;
    BlockEventInput evinput;
    BlockEventOutput evoutput;
    SimFunction sim;
    BlockState state;
    BlockDState dstate;
    BlockODState odstate;
    BlockConstraint constraint;
    BlockRpar rpar;
    BlockIpar ipar;
    BlockOpar opar;
    BlockZcross zcross;
    BlockMode mode;
    SCSPOINTER_COP *work;
    SCSINT_COP ajac;
    BlockName name;
    SCSINT_COP dept;
    SCSSTRING_COP blocktype;

    SCSPOINTER_COP simulatorVars;
    .....
  };
```

Note that `vss_block` is a hierarchical `C` structure. For example, `BlockOutput` is another structure defined as:

```
  typedef struct {
    ....
    SCSINT_COP nout;
    SCSINT_COP outdim;
    SCSINT_COP *outsz;
    SCSINT_COP *outtyp;
    SCSPOINTER_COP *outptr;
    ....
  }BlockOutput;
```

In the above structure, the standard `C` data-types, are redefined, for example:

```
  #define SCSREAL_COP double
```

The complete list of these data-types are given in 9.1.1.

The block structure is quite complex; in order to facilitate accessing the embedded data, a large number of macros are provided. All access to this structure must be made through these macros. Most

commonly used macros are presented in Sections and . As an example, in order to access the data-type of the second block output, the user can use the following macro:

```
int *odt=GetOutType(block,2);
```

Flag

`Flag` is an integer (or more precisely an enumeration) value indicating the job for which the block has been called by the simulator. Blocks may be called with different flags to perform different operations. For example the block can be called with `flag= VssFlag_OutputUpdate` $(= 1)$, to update its outputs or can be called with `flag= VssFlag_Derivatives` $(= 0)$, to provide the state derivatives.

The list if all flags and their meanings are provided below.

- `VssFlag_Initialize`: The block is called for the first time at the beginning of the simulation with this flag where initialization issues can be done. Operations such as opening data files, initializing a TCP/IP socket, or dynamic memory allocation can be done in this flag call. Although continuous-time and discrete-time states have already been initialized, they can be reinitialized (if needed). In this flag, input and output of the block are not available, hence cannot be initialized or read. Each block is called with this flag only once at the beginning of the simulation.

- `VssFlag_Terminate`: Once the simulation is finished, or stopped due to an error or on the user request, the simulator calls the blocks once at final time with this flag. This flag is useful, e.g., for closing a file or a socket that has been opened in flag `VssFlag_Initialize`.

- `VssFlag_OutputUpdate`: When the block is called with this flag, the simulator is requesting the output(s) of the block. The block should compute the outputs as a function of time, inputs, `nevprt`[2], and internal states. These values should be obtained from the block structure (the first argument of the simulation function). The computed output should be written in the block structure using appropriate macros. Note that if the block contains `modes`[3], then the output may be computed as a function of whether `modes` are fixed or relaxed; in some cases the call should be ignored. In this case `isModeFixed(block)` and `isinTryPhase(block)` macros may be used.

- `VssFlag_StateUpdate`: When the simulator calls the blocks with this flag, it means that an event has activated the block (`nevprt>0`) and the block can update is discrete-time and continuous-time states. The activation event may also be due to a zero-crossing event happened inside the block (an internal event), in which case (`nevprt=-1`). When this happens, the `jroot`[4] vector indicates the surface that has crossed and the direction of crossing. If ith entry of `jroot` is +1 (respectively -1), the the ith surface has crossed zero with a positive (respectively negative) slope. If it is zero, i-th surface has not crossed zero.

- `VssFlag_Derivatives`: When the block is called with this flag, the simulator requests the block to provide the continuous-time state time derivatives $(\dot{x})$[5]. The simulation function should compute $\dot{x}$ in the address provided by the block for this purpose. If the block is implicit, i.e., the dynamics of the block is represented by a DAE, the block should compute the residual[6] in the corresponding address.

---

[2]accessible by macro `GetNevIn(block)`
[3]accessible by macro `GetModePtrs(block)`
[4]accessible by macro `GetJrootPtrs(block)`
[5]accessible by macro `GetDerState(block)`
[6]accessible by macro `GetResState(block)`

- `VssFlag_ZeroCrossings`: When the block is called with this flag, the simulator requests the block to provide its zero-crossing surfaces. Variable-step numerical solvers of **Twin Activate** assume model smoothness. This means that the behavior of blocks, which contain or affect continuous-time states of the model must be smooth, or at least piecewise smooth. In case of non-smoothness, the points of discontinuity must be specified in such a way that the solver can, if needed, issue a cold restart when going though such points. An example of such a block is the `ABS` block. The absolute value function is smooth except at zero, where it is not differentiable. In this case, we say that absolute value block has two `modes`: one where the output equals the input, and one where the output is the negative of the input. A block may have many different modes. The point of non-smoothness or discontinuity is represented by a zero-crossing, and the `mode` is updated after such a crossing occurs. The `VssFlag_ZeroCrossings` flag is used both for evaluating the zero-crossing function and returning the zero-crossing vector[7] and setting the `mode` variables. Mode setting is done only when `isModeFixed(block)` is False. Note that the block may have more zero-crossing than modes. For example, the `Zero-cross` block which generates an event when its input crosses zero, has one zero-crossing surface, but no mode.

- `VssFlag_EventScheduling`: When the block is called with this flag, it means that the block can program a new event at one or more of its output activation ports. After this call, the simulator updates its event scheduler. An event can also be programmed after an internal zero-crossing. In this case, the block can use the `jroot` vector.

- `VssFlag_Reinitialize`: When the simulator calls the block with this flag, the block is allowed to reset its continuous-time state as a function of its input values.

- `VssFlag_ReinitializeImplicit`: In implicit models, after event leading to restarting the solver, the implicit blocks are called with this flag allowing blocks updating states and state derivatives.

- `VssFlag_Projection`: If a block dynamics is represented by an ODE with algebraic constraints, the simulator calls the block with this flag to either compute the projection update or the residual of constraints. The type of constraint is either `CONSTRAINT` or `PROJECTION`. If the model contains blocks with constraints, the number of constraints should be given by this macro `GetNConstraint(block)`. Note that only the `CPODE` solver can be used to integrate these type of models.

- `VssFlag_Jacobians`: When the block is called with this flag, the simulator requests the block to provide the analytical Jacobian matrix of the model. The block is called with this flag only if the block informs the simulator by setting `SetAjac(block,1)` in flag=`VssFlag_Initialize`.

- `VssFlag_GotoPause`: Before stopping the simulation and going to pause, the block is called for the last time to perform necessary jobs if needed, for example flushing temporary data into file.

- `VssFlag_ReturnFromPause`: After returning from the pause, the blocks are called with this flag before resuming the simulation.

Flag calling sequence in blocks

The simulator invokes the blocks with specified `flag` to perform a job or a model update. In order to instantiate a block in the simulator before any subsequent call, the simulator calls all blocks

---

[7]accessible by macro `GetGPtrs(block)`

with `flag= VssFlag_Initialize`. Just before finishing the simulation, all blocks are called with `flag= VssFlag_Terminate` to inform the block that this is the last call from the simulator. A simple flowchart has been given in Fig.9.1.
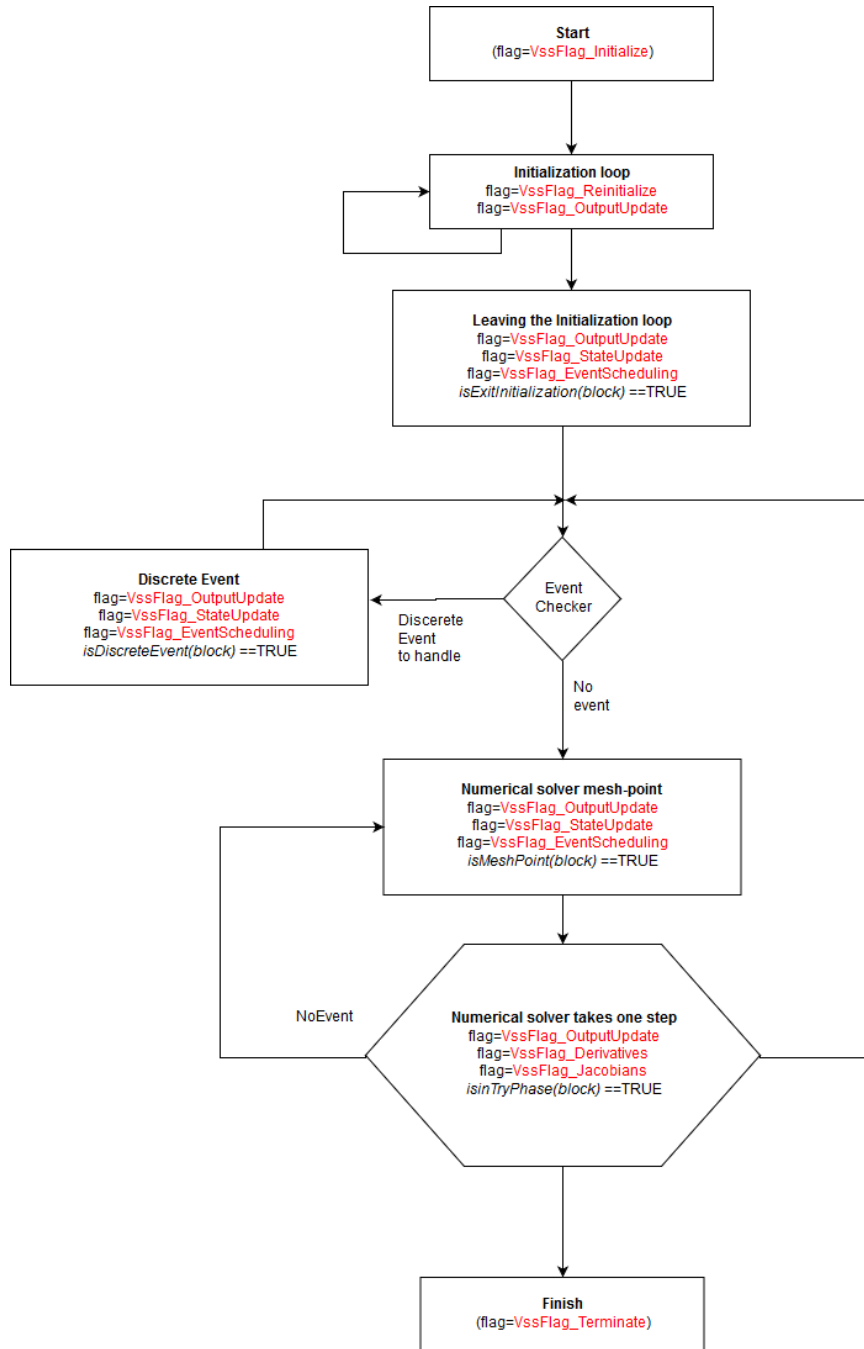
Figure 9.1: Simple flowchart indicating different stages of a simulation and different flag by which a block can be called.

Data-type supported in blocks

Signals in **Twin Activate** can be matrices of various data-types. The data-type of input and output of a block may either be defined and fixed by the block or be inherited from data-type of other connecting blocks.

In general a block supporting multiple data types should provide a different simulation function for each data type. But in some cases it is possible to avoid this by using a single simulation function. For that, the data types and sizes of the parameters, states, inputs and outputs are made available inside the simulation function via specific macros. For example `GetInType(block,n)` returns the data-type of nth input port. The returned data-type can be one of these enumeration values.

- `SCSREAL_N`: double data-type ID
- `SCSCOMPLEX_N`: complex data-type ID
- `SCSINT32_N`: int32 data-type ID
- `SCSINT16_N`: int16 data-type ID
- `SCSINT8_N`: int8 data-type ID
- `SCSUINT32_N`: unsigned int32 data-type ID
- `SCSUINT16_N`: unsigned int16 data-type ID
- `SCSUINT8_N`: unsigned int8 data-type ID
- `SCSBOOL_N`: Boolean data-type ID
- `SCSSTRING_N`: string data-type ID

The corresponding types that can be used in the simulation function are:

- `SCSREAL_COP`: double data-type
- `SCSCOMPLEX_COP`: complex data-type
- `SCSINT32_COP`: int32 data-type
- `SCSINT16_COP`: int16 data-type
- `SCSINT8_COP`: int8 data-type
- `SCSUINT32_COP`: unsigned int32 data-type
- `SCSUINT16_COP`: unsigned int16 data-type
- `SCSUINT8_COP`: unsigned int8 data-type
- `SCSBOOL_COP`: Boolean data-type
- `SCSSTRING_COP`: string data-type

Users are encouraged to use these data-type instead of using directly `C` data-types in the definition of simulation functions.

Feedthrough information in blocks

The feedthrough is a block property that specifies the input-output dependency of the block. This property must be specified when a new block is constructed (including for blocks realized by custom blocks). Specifically the feedthrough property is a vector of size equal to the number of block inputs.

The ith entry of the vector indicates whether or not the block outputs depend "directly" on the ith input value. Consider a Gain block with gain G, input u and output y (one input, one output) realizing:

$$y = Gu$$

In this case the output depends directly on the input (unless G is identically zero), so the Gain block in general has feedthrough property (more specifically its input has it). Now consider an integrator block with input u, output y and state x:

$$\begin{cases} \dot{x} & = u \\ y & = x \end{cases}$$

In this case the output y does not directly depend on the input u. The integrator block (its input in particular) does not have feedthrough property. The feedthrough property should be provided correctly otherwise the **Twin Activate** compiler cannot compute proper scheduling. A scheduling error may occur if a true feedthrough property is erroneously set to false. On the other hand, specifying a feedthrough property to be true when it is not can lead to a non-existing algebraic loop error detected by the compiler. The algebraic loop error occurs when the diagram contains a path looping through ports having all feedthrough properties. In that case the blocks cannot be scheduled by the compiler. Even if there are ways to "break" algebraic loops (in particular using the **Loopbreaker** block), it is better not to introduce non-existing ones by properly defining the feedthrough properties of new blocks. For end users, learning about this property is useful for dealing with algebraic loop errors.

9.1.2 Examples

In this section in order to illustrate the way a simulation function is written, several examples are given.

Simplified Gain block

Consider a Gain block that multiplies the input matrix by a matrix gain value and puts the results at the block output. Here is the simulation function of this block.

```
#include "vss_block4.h"

VSS_EXPORT void SimpleGainBlock (vss_block *block,int flag)
{
    if (flag==VssFlag_OutputUpdate){
        SCSREAL_COP *u=GetRealInPortPtrs(block,1);
        int ru=GetInPortRows(block,1);
        int cu=GetInPortCols(block,1);

        SCSREAL_COP *y=GetRealOutPortPtrs(block,1);
        int ry=GetOutPortRows(block,1);
        int cy=GetOutPortCols(block,1);

        SCSREAL_COP *Gain=GetRealOparPtrs (block,1);
        int rowGain=GetOparSize(block,1,1);
        int colGain=GetOparSize(block,1,2);
        int i, j, k;
```

```
        for (i=0;i<ry;++i){
            for (j=0;j<cy;++j){
              k=i+j*ry;
              y[k]=Gain[k]*u[k];
            }
        }
    }
}
```

Note that the macro `GetRealInPortPtrs(block,1)` returns a pointer to the vector of the first input port. The gain matrix which is a block parameter is accessed by `GetRealOparPtrs(block, 1)`. The gain matrix value is of type Real (double). The row and column size of the gain matrix are accessed as follows:

```
rowGain=GetOparSize(block,1,1);
colGain=GetOparSize(block,1,2);
```

The output of the block should be computed whenever the output is requested by the simulator, i.e., when the block is called with `flag=VssFlag_OutputUpdate`. At this flag, the input and parameter matrices are read and the output is computed. This block may also be invoked by other flags, but it does nothing.

Note that the dimensions of the matrices are not checked for consistency. The checks should be made at the evaluation or compilation phases.

Simplified ODE/DAE block

In order to demonstrate the way a simple ODE and DAE systems can be implemented inside a **Twin Activate** block, consider the following initial value problem problem.

$$\begin{cases} \dot{x}_1 = -x_1 + x_2 x_1 \\ \dot{x}_2 = 2x_1 - 3x_2 \end{cases}$$

with initial value

$$[x_1, x_2] = [1, 2].$$

This ODE can be implemented as follows.

```
#include "vss_block4.h"

VSS_EXPORT void Simple_ODE_Block (vss_block *block, int flag)
{
    SCSREAL_COP *y=GetRealOutPortPtrs(block,1);
    SCSREAL_COP *xd=GetDerState(block);
    SCSREAL_COP *x=GetState(block);
    int nx=GetNstate(block);
    int i;

    switch(flag)
    {
     case VssFlag_Initialize:
```

```
        x[0]=1.0;
        x[1]=2.0;
        break;
    case VssFlag_OutputUpdate:
        for (i=0;i<nx;++i){
            y[i]=x[i];
        }
        break;
    case VssFlag_Derivatives:
        xd[0]=-x[0]+x[1]*x[0];
        xd[1]=-3*x[1]+2*x[0];
        break;
    default:
    }

}
```

In this code, first the pointers to state (x) and state derivative (xd) as well as the size of the state (nx) are obtained. Three flags are used, VssFlag_OutputUpdate for output the state value and VssFlag_Derivatives for returning the time derivative of the state.

In order to demonstrate the way a DAE is implemented in a **Twin Activate** block, let's reformulate the above ODE as a DAE. We get the following DAE.

$$\begin{cases} 0 = -\dot{x}_1 - x_1 + x_2 x_1 \\ 0 = -\dot{x}_2 + 2x_1 - 3x_2 \end{cases}$$

The C code for this DAE can be expressed as follows:

```
#include "vss_block4.h"

VSS_EXPORT void Simple_DAE_Block (vss_block *block, int flag)
{
    SCSREAL_COP *y=GetRealOutPortPtrs(block,1);
    SCSREAL_COP *xd=GetDerState(block);
    SCSREAL_COP *x=GetState(block);
    SCSREAL_COP *res=GetResState(block);
    int nx=GetNstate(block);
    int i;

    switch(flag)
    {
     case VssFlag_Initialize:
        x[0] =1.0;
        x[1] =2.0;
        xd[0]=0.0;
        xd[1]=0.0;
        break;

     case VssFlag_OutputUpdate:
```

```
        for (i=0;i<nx;++i){
            y[i]=x[i];
        }
        break;

    case VssFlag_Derivatives:
        res[0]=-xd[0]-x[0]+x[1]*x[0];
        res[1]=-xd[1]-3*x[1]+2*x[0];
        break;

    default:
    }
}
```

Simplified Counter (inc/dec) block

In this example, we develop a counter block. Whenever an event activates the counter, the counter increments or decrements its value. The initial value of the counter and the direction of the counter are given as block parameters.

```
#include "vss_block4.h"
VSS_EXPORT void Simple_counter_Block (vss_block *block, int flag)
{
    SCSINT8_COP  *y=Getint8OutPortPtrs(block,1);
    SCSINT8_COP  initial=*Getint8OparPtrs(block,1);
    SCSINT8_COP  dir    =*Getint8OparPtrs(block,2);
    SCSINT8_COP  mem    = Getint8OzPtrs(block,1);
    int nevprt = GetNevIn(block);

    if (flag==VssFlag_Initialize){
            mem[0]=initial;
    }else if (flag==VssFlag_OutputUpdate) {
          y[0]=mem[0];
    }else if (flag== VssFlag_StateUpdate) {
        if (nevprt ==1){
          if (dir==1)
              mem[0]++;
          else
              mem[0]--;
        }
    }
}
```

The initial value of the counter is returned by the macro `Getint8OparPtrs(block,1)`. The block needs a storage to keep the counter value. We chose `OZ` storage, which here is a scalar 8-bit integer. In general `OZ` is initialized before the simulation phase, but for demonstration purposes we reinitialized it in flag=`VssFlag_Initialize`. On every block activation with flag=`VssFlag_StateUpdate`, the `nevprt` is checked to verify if the activation is a discrete event. Then based on the block parameter (`dir`), the counter is incremented or decremented.

Simplified Counter block with overflow

In the above counter suppose that the block needs to generate an event when the counter over-flows/underflows. In this case, an event needs to be programmed in flag=`VssFlag_EventScheduling`.

```c
#include "vss_block4.h"
VSS_EXPORT void Simple_counter_Block (vss_block *block, int flag)
{
    SCSINT8_COP  *y=Getint8OutPortPtrs(block,1);
    SCSINT8_COP  initial=*Getint8OparPtrs(block,1);
    SCSINT8_COP  dir    =*Getint8OparPtrs(block,2);
    SCSINT8_COP  mem    = Getint8OzPtrs(block,1);
    int nevprt = GetNevIn(block);
    double *evout= GetNevOutPtrs(block);

    if (flag==VssFlag_Initialize){
            mem[0]=initial;
    }else if (flag==VssFlag_OutputUpdate) {
          y[0]=mem[0];
    }else if (flag== VssFlag_StateUpdate) {
        if (nevprt ==1){
          if (dir==1)
              mem[0]++;
          else
              mem[0]--;
        }
    } else if (flag==VssFlag_EventScheduling){
         if (nevprt ==1){
           if (dir==1){
             if (mem[0]==127) {
                 evout[0]=0.0;
             }
           }else{
             if (mem[0]==-128){
                 evout[0]=0.0;
             }
           }
         }

    }
}
```

In this counter, when the block is called with flag=`VssFlag_EventScheduling`, the content of the counter is checked and based on the counter direction and its value, an immediate discrete event (with zero delay) is generated at its first activation output port.

Simplified Bouncing ball block

In order to demonstrate the way zero-crossings and `modes` are handled, consider the model of a simple bouncing ball.

$$\begin{cases} \dot{h} & = v \\ \dot{v} & = -9.81 \end{cases}$$
$$[h, v] = [10, 0]$$

When the ball hits the ground, its vertical speed changes according to

$$v = -0.9v^-$$

The code for modeling this system is:

```
#include "vss_block4.h"
VSS_EXPORT void Simple_BouncingBall_Block (vss_block *block, int flag)
    SCSREAL_COP *h=GetRealOutPortPtrs(block,1);
    SCSREAL_COP *v=GetRealOutPortPtrs(block,2);
    SCSREAL_COP *xd=GetDerState(block);
    SCSREAL_COP *x=GetState(block);
    SCSREAL_COP *g=GetGPtrs(block);
    int nevprt = GetNevIn(block);

    switch(flag)
    {
     case VssFlag_OutputUpdate:
        h[0]=x[0];
        v[0]=x[1];
        break;

     case VssFlag_Derivatives:
        xd[0]= x[1];
        xd[1]= -9.81 ;
        break;

     case VssFlag_StateUpdate:
        if (nevprt==-1){
          x[0]=  0.0;
          x[1]= -0.9*x[1];
        }
        break;

     case VssFlag_ZeroCrossings
        g[0]=x[0];
        break;

     default:
    }
}
```

The model contains two state variables, one for the height of the ball (h) and another for the velocity of the ball (v). The block has two output ports, the first outputs h and the second, v. A zero-crossing surface has been used to check whether the ball has hit the ground. Naturally the height of the ball (h) can be used as a zero-crossing surface in the block. When the ball hits the ground, an internal event is generated (with `nevprt==-1`). At this moment, the state variable (h and v) are reset with new values.

Absolute value block

In order to demonstrate the way `mode` variables are handled in a block, consider a simplifed version of the `Abs` block in **Twin Activate**. This block computes the absolute value of its input.

$$y = \left\{ \begin{array}{ll} u & \text{if } u \geq 0 \\ -u & \text{otherwise} \end{array} \right.$$

When a variable step-size numerical solver is chosen in **Twin Activate**, if the output of the `Abs` block affects continuous-time state of the model, mode variable should be used inside the block to handle correctly the non-smoothness or discontinuity. The absolute value block needs then a zero-crossing surface to detect when the input crosses zero and a `mode` to handle the discontinuity. The zero-crossing surface is defined in flag=`VssFlag_ZeroCrossings`. When the block is called with this flag, the block updates the zero-crossing surface and modes.

In order to update the `mode` variables, the `areModesFixed(block)` macro is checked, if false, `modes` can be updated. If the simulator calls the block with flag=`VssFlag_OutputUpdate` for updating the outputs, the block should make use of the `mode` variable. If the simulator calls the block when `areModesFixed(block)` is true, the outputs depends on the `mode`. The code for the simulation function of the block is:

```
#include <vss_block4.h>
VSS_EXPORT void  simple_abs(vss_block *block,int flag)
{
    SCSREAL_COP *u=GetRealInPortPtrs(block,1);
    SCSREAL_COP *y=GetRealOutPortPtrs(block,1);
    SCSREAL_COP *g=GetGPtrs(block);
    SCSINT32_COP *mode=GetModePtrs(block);
    int side;

    switch(flag)
    {
    case VssFlag_OutputUpdate:
        if (areModesFixed(block)) {
            side=mode[0];
        }else {
            if (u[0]<0){   side=2;
            } else{        side=1;
            }
        }
        if (side==1){   y[0]=+u[0];
        } else{         y[0]=-u[0];
        }
```

```
      break;
    case VssFlag_ZeroCrossings:
         g[0]=u[0];
         if (!areModesFixed(block)) {
            if(g[0]<0){
                mode[0]=2;
            }else{
                mode[0]=1;
            }
         }
    }
}
```

A block to write signals to a file

In this example we demonstrate the way the block input can be written into an ascii file. The input of the block is scalar of data type double. The code is:

```
#include "vss_block4.h"

typedef struct {
    unsigned int index;
    FILE *pFile;
}write2File_t;

#define WS      ((write2File_t *) GetWorkPtrs(block))

VSS_EXPORT void write_to_file(vss_block *block,int flag)
{
    switch(flag)
    {
        /*--------------------------*/
    case VssFlag_Initialize:
        {
            char *fname=Getint8OparPtrs(block,1);
            GetWorkPtrs(block)= (write2File_t*)vss_malloc(block,
                sizeof(write2File_t) );
            if (!WS){SetBlockError(block,1);return;}
            WS->index=0;
            WS->pFile = fopen (fname,"w");
            if(!WS->pFile) {Coserror(block,"Unable to open the file %s\n"
                , fname); return;}
            fprintf (WS->pFile, "input data.....\n");
        }
        break;
        /*--------------------------*/
    case VssFlag_Terminate:
        {
```

```
            if (!WS) return;
            if (WS->pFile){
                fprintf (WS->pFile, "Number of lines: %d",WS->index);
                fclose(WS->pFile);
            }
        }
        break;
        /*-------------------------*/
    case VssFlag_OutputUpdate:
        {
            SCSREAL_COP *u = GetRealInPortPtrs(block,1);
            if (!isinTryPhase(block)){
                fprintf (WS->pFile, "%g\n", u[0]);
                WS->index++;
            }
        }
        break;
    }
    return;
}
```

At the beginning of the simulation, when the block `write_to_file` is called with flag = `VssFlag_-Initialize` the file whose name (`fname`) is given as a block parameter is opened for writing. In order to dump the block input data into the file at block calls with flag = `VssFlag_OutputUpdate`, the pointer to file should be kept throughout the simulation. This pointer can either be stored as a static variable in the simulation function or in the block's WorkSpace[8]. The static method is not recommended, since if two instances of this block are used in a model, both share the same static variable that perturbs the operation of the blocks. For this reason, we create a structure to keep the file pointer as well as any other variable that needs to be kept during the simulation. The structure is allocated with `vss_malloc` function of **Twin Activate**. This function is similar to `malloc`, but with the difference that the pointer of the allocated memory is managed and freed by **Twin Activate** simulator. So the user does not need to worry about freeing the memory at the end. **Twin Activate** provides also the function `vss_file_open(block,fname,"w");` instead of `fopen(fname,"w")`. `vss_file_open` does not need `fclose`, it is done automatically by the simulator. Another important point in the above code is using `isinTryPhase (block)` macro. This macro filters the try calls where the input of the block is not a valid value and can be ignored.

Simplified ODE with Constraint preservation

In some models the states of the dynamic systems need to satisfy invariant or some algebraic relationships. This kind of invariants is usually resulting from mass or energy conservation or other physical laws. If the ODE is being integrated by standard numerical solvers, slight errors in the numerical solution are accumulated and cause the solutions to fail to satisfy solution invariant exactly. In order to add these kind of invariant to the ODE of the block, **Twin Activate** provides the use of flag =`VssFlag_-Projection`. and the `CPODE` solver. During the simulation the simulator calls the simulation functions of the blocks with constraints with this flag and requests the block to compute the invariant or the projections as a function of newly computed states. This results in a solution that sticks to the constraints.

---

[8]accessible by macro `GetWorkPtrs(block)`

The following example is the standard pendulum model with two invariants. The dynamic model of the pendulum is

$$\begin{cases} \dot{x} & = v_x \\ \dot{y} & = v_y \\ \dot{v}_x & = -x(v_x^2 + v_y^2 - gy)/L^2 \\ \dot{v}_y & = -y(v_x^2 + v_y^2 - gy)/L^2 - g \end{cases}$$

The first invariant, e.g., ensures that the length of the pendulum remains equal to L throughout the simulation.

$$\begin{cases} x^2 + y^2 & = L^2 \\ xv_x + yv_y & = 0 \end{cases}$$

The code for the simulation function of the pendulum model with constraint preservation follows.

```
#include "vss_block4.h"
#define g    9.81
VSS_EXPORT void CBlockFunction(vss_block *block,int flag)
{
    double *xd=GetDerState(block);
    double *X=GetState(block);
    double *y=GetRealOutPortPtrs(block,1);
    int nx=GetNstate(block);
    int  i, ip;

    switch(flag)
    {
        /*------------------------*/
    case VssFlag_Initialize:
        {
            SetConstraintKind(block,CONSTRAINT);
            SetNConstraint(block,2);
            X[0]=1;
            X[1]=X[2]=X[3]=0;
            return;
        }

    case VssFlag_Projection:
        {
            double x, y, xd, yd;
            double *corr=GetCorrPtr(block);
            x  = X[0];
            y  = X[1];
            xd = X[2];
            yd = X[3];

            corr[0]= x*x + y*y - 1.0;
            corr[1]= x*xd + y*yd;
            return;
        }
```

```
    case VssFlag_Derivatives:
        {
            double x, y, xd, yd, tmp;
            x  = X[0];
            y  = X[1];
            xd = X[2];
            yd = X[3];
            tmp = xd*xd + yd*yd - g*y;

            xd[0] = xd;
            xd[1] = yd;
            xd[2] = -x*tmp;
            xd[3] = -y*tmp - g;
            return;
        }

    case VssFlag_OutputUpdate:
        {
            for (i=0;i<nx;i++)
                y[i]  = X[i];
            return;
        }
    default:
        return;
    }
}
```

ODE and DAE blocks providing Analytical Jacobian

In order to solve an ODE/DAE, the numerical variable-step solvers need the Jacobian matrix of the ODE/DAE. In **Twin Activate**, the Jacobian matrix can either be computed numerically by the finite difference method or can be provided by blocks. In this section, we show the way the Jacobian of a ODE/DAE can be provided by the block. Consider the following ODE model and its Jacobian matrix is

$$\dot{x} = f(x) = \begin{cases} -x_1 + x_1 x_2 \\ 2x_1 - 3x_2 \end{cases}$$

$$J = \frac{\partial f}{\partial x} = \begin{bmatrix} -1 + x_2 & x_1 \\ 2 & -3 \end{bmatrix}$$

The simulation function for this block providing the analytical Jacobian matrix is given below. In flag=`VssFlag_-Initialize`, the jacobian flag is set to 1 to inform the simulator that the block provides the analytical Jacobian. The `J` vector has been filled with the Jacobian entries in flag `VssFlag_Jacobians`. Note that the `J` matrix is filled column-wise.

```
#include "vss_block4.h"
```

```
VSS_EXPORT void Simple_ODE_Block (vss_block *block, int flag)
{
    SCSREAL_COP *y=GetRealOutPortPtrs(block,1);
    SCSREAL_COP *xd=GetDerState(block);
    SCSREAL_COP *x=GetState(block);
    SCSREAL_COP *J= GetJacobianPtrs(block);
    int nx=GetNstate(block);
    int i;

    switch(flag)
    {
     case VssFlag_Initialize:
        SetAjac(block,1);
        return;

     case VssFlag_OutputUpdate:
        for (i=0;i<nx;++i){
            y[i]=x[i];
        }
        break;

     case VssFlag_Derivatives:
        xd[0]=-x[0]+x[1]*x[0];
        xd[1]=-3*x[1]+2*x[0];
        break;

     case VssFlag_Jacobians:
        J[0]=-1+x[1];
        J[1]=2;
        J[2]=x[0];
        J[3]=-3;
        break;

     default:
    }
}
```

The Jacobian matrix for a DAE is a little different. For the DAE

$$0 = F(\dot{x}, x)$$

the Jacobian Matrix is defined as follows

$$J = \alpha \frac{\partial F}{\partial x} + \beta \frac{\partial F}{\partial \dot{x}}$$

where α[9] and β[10] values are given by the simulator. As a result the Jacobian matrix for the above DAE

---

[9]accessible by macro `GetAlphaPt(block)`

[10]accessible by macro `GetBetaPt(block)`

will be

$$J = \begin{bmatrix} -\alpha + \alpha x_2 - \beta & \alpha x_1 \\ 2\alpha & -3\alpha - \beta \end{bmatrix}$$

```c
#include "vss_block4.h"

VSS_EXPORT void Simple_ODE_Block (vss_block *block, int flag)
{
    SCSREAL_COP *y=GetRealOutPortPtrs(block,1);
    SCSREAL_COP *xd=GetDerState(block);
    SCSREAL_COP *x=GetState(block);
    SCSREAL_COP *J= GetJacobianPtrs(block);
    SCSREAL_COP *alpha=GetAlphaPt(block);
    SCSREAL_COP *beta =GetBetaPt(block);
    int nx=GetNstate(block);
    int i;

    switch(flag)
    {
     case VssFlag_Initialize:
        SetAjac(block,1);
        return;
     case VssFlag_OutputUpdate:
        for (i=0;i<nx;++i){
            y[i]=x[i];
        }
        break;
     case VssFlag_Derivatives:
        xd[0]=-x[0]+x[1]*x[0];
        xd[1]=-3*x[1]+2*x[0];
        break;
     case VssFlag_Jacobians:
        J[0]=-alpha[0]+alpha[0]*x[1]-beta[0];
        J[1]=-2*alpha[0] ;
        J[2]=alpha[1]*x[0];
        J[3]=-3*alpha[1]-beta[1];
        break;
     default:
    }
}
```

A simple block for programming an initial event

Consider a simple block with one output event port that generates an event at a time instant spec-
ified by a block parameter. In order to generate the event, the block should read the parameter
value using `evtime= GetRealOparPtrs(block,1)`. Events are only programmed at `flag==
VssFlag_EventScheduling`. In order to program this event only once, at the beginning of the
simulation, the `isExitInitialization(block)` macro can be used. This macro indicates if the

initialization has finished and the simulation is being started. The relative time with respect to the current simulation time and the desired event time is defined and set to the `evout` vector.

```
#include "vss_block4.h"

VSS_EXPORT void EventGenerateBlock(vss_block *block,int flag)
{
    double *evout =GetNevOutPtrs(block);
    double *evtime=GetRealOparPtrs(block,1);
    if(flag==VssFlag_EventScheduling){
        if (isExitInitialization(block)){
            evout[0]=evtime[0]-GetVssInitialTime(block);
        }
    }
}
```

9.1.3   Macros for accessing the block structure

The following macros are used to access `vss_block` structure inside the block simulation function. In all macros the `block` argument is of type `vss_block` and `n, m` are strictly positive integer values.

- `GetNin(block)`: Get the number of regular input ports of the block.

- `GetInDim(block)`: Get the dimension of the regular input ports.

- `GetInPortPtrs(block,n)`: Get regular input port pointer of port number n. n must be strictly positive.

- `GetNout(block)`: Get number of regular output port of the block.

- `GetOutDim(block)`: Get the dimension of the regular output ports.

- `GetOutPortPtrs(block,n)`: Get regular output port pointer of port number n. n must be strictly positive.

- `GetInPortSize(block,n,m)`: Get size of the regular input port number n for the dimension m. n and m must be strictly positive.

- `GetInPortSize(block,n,1)`: Get first dimension (row size) of input port number n. n must be strictly positive.

- `GetInPortSize(block,n,2)`: Get second dimension (column size) of input port number n. n must be strictly positive.

- `GetInPortRows(block,n)`: Get number of rows (first dimension) of regular input port number n. n must be strictly positive.

- `GetInPortCols(block,n)`: Get number of columns (second dimension) of regular input port number n. n must be strictly positive.

- `GetInType(block,n)`: Get data type of regular input port number n. n must be strictly positive.

- `GetOutPortSize(block,n,m)`: Get size of the regular output port size number n for the dimension m. m,n must be strictly positive.

- `GetOutPortSize (block,n,1)`: Get first dimension (row size) of output port number n . n must be strictly positive.

- `GetOutPortSize (block,n,2)`: Get second dimension (column size) of output port number n . n must be strictly positive.

- `GetOutPortRows(block,n)`: Get number of rows (first dimension) of regular output port number n. n must be strictly positive.

- `GetOutPortCols(block,n)`: Get number of columns (second dimension) of regular output port number n . n must be strictly positive.

- `GetOutType(block,n)`: Get data-type of regular output port number n. n must be strictly positive.

- `GetRealInPortPtrs(block,n)`: Get pointer of real part of regular input port number n. n must be strictly positive.

- `GetImagInPortPtrs(block,n)`: Get pointer of imaginary part of regular input port number n. n must be strictly positive.

- `Getint8InPortPtrs(block,n)`: Get pointer of int8 typed regular input port number n. n must be strictly positive.

- `Getint16InPortPtrs(block,n)`: Get pointer of int16 typed regular input port number n. n must be strictly positive.

- `Getint32InPortPtrs(block,n)`: Get pointer of int32 typed regular input port number n. n must be strictly positive.

- `Getuint8InPortPtrs(block,n)`: Get pointer of unsigned int8 typed regular input port number n. n must be strictly positive.

- `Getuint16InPortPtrs(block,n)`: Get pointer of unsigned int16 typed regular input port number n. n must be strictly positive.

- `Getuint32InPortPtrs(block,n)`: Get pointer of unsigned int32 typed regular input port number n. n must be strictly positive.

- `GetRealOutPortPtrs(block,n)`: Get pointer of real part of regular output port number n. n must be strictly positive.

- `GetImagOutPortPtrs(block,n)`: Get pointer of imaginary part of regular output port number n. n must be strictly positive.

- `Getint8OutPortPtrs(block,n)`: Get pointer of int8 typed regular output port number n. n must be strictly positive.

- `Getint16OutPortPtrs(block,n)`: Get pointer of int16 typed regular output port number n. n must be strictly positive.

- `Getint32OutPortPtrs(block,n)`: Get pointer of int32 typed regular output port number n. n must be strictly positive.

- `Getuint8OutPortPtrs(block,n)`: Get pointer of unsigned int8 typed regular output port number n. n must be strictly positive.

- `Getuint16OutPortPtrs(block,n)`: Get pointer of unsigned int16 typed regular output port number n. n must be strictly positive.

- `Getuint32OutPortPtrs(block,n)`: Get pointer of unsigned int32 typed regular output port number n. n must be strictly positive.

- `GetNipar(block)`: Get number of integer parameters of the block.

- `GetNrpar(block)`: Get number of real parameters of the block.

- `GetNopar(block)`: Get number of object parameters of the block. This kind of parameters can be of any data type or tables.

- `GetIparPtrs(block)`: Get pointer of the integer parameters vector .

- `GetRparPtrs(block)`: Get pointer of the real parameters vector.

- `GetPtrWorkPtrs(block)`: Get the pointer of pointer of the Work array.

- `GetNstate(block)`: Get number of continuous state. n must be strictly positive.

- `GetState(block)`: Get pointer of the continuous state vector.

- `GetDerState(block)`: Get the pointer to the vector of the derivative of continuous state.

- `GetResState(block)`: Get pointer of the residual continuous state vector.

- `GetJacobianPtrs(block)`: Get pointer on the Jacobian matrix.

- `GetXpropPtrs(block)`: Get pointer of continuous state properties vector. For implicit blocks, this vector indicates if a state is differential or algebraic

- `GetNdstate(block)`: Get number of real data-type discrete state.

- `GetDstate(block)`: Get pointer of the discrete state vector.

- `GetNevIn(block)`: Get the input event number. This number is coded as binary. For example if first than third events are synchronously activated, this macro returns 5.

- `GetNevin(block)`: Get number of event input ports.

- `GetNevOut(block)`: Get number of event output ports.

- `GetNevOutPtrs(block)`: Get pointer of event output vector. The block fills this vector with the time of the next coming event. The time is relative, e.g., for programming an event instantly the vector element is 0.0.

- `GetEvOutVal(block,n)`: Get the content of the event vector, negative element means no event is programmed. n must be strictly positive.

- `GetOparType(block,n)`: Get data-type of object parameters number n. n must be strictly positive.

- `GetOparSizePtrs(block)`: Get size of object parameters

- `GetOparSize(block,n,m)`: Get size of object parameters number n for the dimension m. m and n must be strictly positive.

- `GetOparSize (block,n,1)`: Get first dimension (row size) of object parameters number n . n must be strictly positive.

- `GetOparSize (block,n,2)`: Get second dimension (column size) of object parameters number n . n must be strictly positive.

- `GetOparPtrs(block,n)`: Get pointer of object parameters number n.

- `GetRealOparPtrs(block,n)`: Get pointer of real object parameters number n.

- `GetImagOparPtrs(block,n)`: Get pointer of imaginary part of object parameters number n.

- `Getint8OparPtrs(block,n)`: Get pointer of int8 typed object parameters number n.

- `Getint16OparPtrs(block,n)`: Get pointer of int16 typed object parameters number n.

- `Getint32OparPtrs(block,n)`: Get pointer of int32 typed object parameters number n.

- `Getuint8OparPtrs(block,n)`: Get pointer of unsigned int8 typed object parameters number n.

- `Getuint16OparPtrs(block,n)`: Get pointer of unsigned int16 typed object parameters number n.

- `Getuint32OparPtrs(block,n)`: Get pointer of unsigned int32 typed object parameters number n.

- `GetStringOparPtrs(block,n)`: Get pointer of unsigned int32 typed object parameters number n.

- `GetNoz(block)`: Get number of object discrete state.

- `GetOzType(block,n)`: Get type of object discrete state number n.

- `GetOzTypePtrs(block)`: Get type of object state.

- `GetOzSizePtrs(block)`: Get size of object state.

- `GetOz(block)`: Get object state

- `GetOzSize(block,n,m)`: Get size of object state number n for the dimension m.

- `GetOzSize (block,n,1)`: Get first dimension (row size) of object state number n.

- `GetOzSize (block,n,2)`: Get second dimension (column size) of object state number n.

- `GetOzPtrs(block,n)`: Get pointer of object state number n.

- `GetRealOzPtrs(block,n)`: Get pointer of real object state number n.

- `GetImagOzPtrs(block,n)`: Get pointer of imaginary part of object state number n.

- `Getint8OzPtrs(block,n)`: Get pointer of int8 typed object state number n.

- `Getint16OzPtrs(block,n)`: Get pointer of int16 typed object state number n.

- `Getint32OzPtrs(block,n)`: Get pointer of int32 typed object state number n.

- `Getuint8OzPtrs(block,n)`: Get pointer of unsigned int8 typed object state number n.

- `Getuint16OzPtrs(block,n)`: Get pointer of unsigned int16 typed object state number n.

- `Getuint32OzPtrs(block,n)`: Get pointer of unsigned int32 typed object state number n.

- `GetSizeOfOz(block,n)`: Get the data size of each element of the block discrete OZ state. As an example, if the data type of OZ is Double, this macro returns sizeof(Double).

- `GetSizeOfOpar(block,n)`: Get the data size of each element of the block Opar parameter. As an example, if the data type is Double, this macro returns sizeof(Double).

- `GetSizeOfOut(block,n)`: Get the sizeof of the regular output port number n. Get the data size of each element of the block output. As an example, if the data type is Double, this macro

returns sizeof(Double).

- `GetSizeOfIn(block,n)`: Get the data size of each element of the block input. As an example, if the data type is Double, this macro returns sizeof(Double).

- `GetNg(block)`: Get number of zero crossing surface.

- `GetGPtrs(block)`: Get pointer of the zero crossing vector. This vector should be filled by the block and contain the zero-crossing surfaces.

- `GetJrootPtrs(block)`: Get pointer of the direction of the zero crossing vector

- `GetNmode(block)`: Get number of modes

- `GetWorkPtrs(block)`: Get the pointer of the Work array.

- `GetModePtrs(block)`: Get pointer of the mode vector

- `GetLabelPtrs(block)`: Get pointer of the block label

- `GetBoolInPortPtrs(block,n)`: Get pointer of boolean typed regular input port number n.

- `GetBoolOutPortPtrs(block,n)`: Get pointer of boolean typed regular output port number n.

- `GetImplicitFlag(block)`: check if the block is implicit (DAE)

- `GetBlockType(block)`: Get Block Type.

- `GetBlockNamePtrs(block)`: Get Block Name structure pointer (II)

- `GetBlockName(block)`: Get the block *instance* name

- `GetBlockNSubNames(block)`: list of block inside a virtual block

- `GetBlockSubName(block,n)`: Get the block number n from the list of blocks inside a virtual block

- `GetDept(block)`: check if a block is time dependent

- `GetInher(block)`: Get the inheritance method, 0)no inheritance 1)inheritance from a single event input 2)inheritance from multiple event inputs

- `GetAjac(block)`: check if the block provides the Analytical Jacobian

- `GetCorrPtr(block)`: Get pointer of projection corrector vector

- `GetConstraintKind(block)`: This macro returns the constraint type defined by the block. There are two constraint types can be defined by the block:

  - `CONSTRAINT`: Additional equations defining algebraic equation between states are added to the block dynamical equations. In this case number of additional equations should be given by macro `SetNConstraint(block,n)`.

  - `PROJECTION`: When this option is chosen, states values are readjusted to meet some constraints inside the model.

- `GetNConstraint(block)`: Get number of Constraints, if the constraint type is `CONSTRAINT`

- `SetAjac(block)`: Indicate that the block provides the Analytical Jacobian

9.1.4 Macros for accessing the simulator structure

The followign macros used to access some global variables or information such as simulator current time or error tolerances. These variables are available on the simulator structure.

- `GetFinalTime(block)`: Get the final time of the simulation.
- `GetHmax(block)`: Get the (maximum) step-size provided by the user (Note that this value can be *auto*). If the solver is fixed-step, this value will be the step-size of the solver.
- `GetHmaxUsed(block)`: Get the (maximum) step-size actually used by the simulator.
- `GetAtol(block)`: Get the absolute error tolerance used by the simulator.
- `GetRtol(block)`: Get the relative error tolerance used by the simulator.
- `GetVssInitialTime(block)`: Get the start time of the simulation used by the simulator.
- `GetTtol(block)`: Get the error tolerance in time provided by the user (Note that this value can be *auto*).
- `GetTtolUsed(block)`: Get the error tolerance in time actually used by the simulator.
- `GetAtolptr(block)`: Get the absolute error tolerance vector used by the simulator.
- `GetBlockNum(block)`: Get the ordering number of the block defined by the simulator.
- `GetSQRU(block)`: Get the square-root of the machine epsilon used by the simulator.
- `GetNeq(block)`: Get the number of continuous-time state of the model used by the simulator.
- `GetNzero(block)`: Get the number of zero-crossing surfaces of the model used by the simulator.
- `GetInitialX(block)`: Get initial value of continuous-time states of the model used by the simulator.
- `GetInitialXd(block)`: Get initial value of the derivative of continuous-time states of the model used by the simulator.
- `IsHotReStart(block)`: Indicates if the block is called while the solver is just being reinitialized after a discrete event. 0 means that the solver is just being restarted.
- `isColdRestart(block)`: Same as `IsHotReStart`, but 1 means that the solver is just being restarted.
- `DoColdRestart(block)`: Force the solver to reinitialize.
- `GetVssTime(block)`: Get the current time in the simulator.
- `GetSimulationPhase(block)`: Get the phase of the simulation used by the simulator.
- `GetMeshPoint(block)`: Get the last mesh-point time instant taken by the solver.
- `GetLastStepSize(block)`: Get the last step-size successfully taken by the solver.
- `GetNextTryStepSize(block)`: Get the next step-size being tried by the solver.
- `GetSimulatorInitiated(block)`: indicates if the simulator has finished the initialization of the model.
- `isStopRequested(block)`: Indicates if the user has pressed the stop button to terminate the simulation.

- `StopSimulation(block,val)`: Force the simulator to finish the simulation.

- `isZeroCrossingEnabled(block)`: Indicates if the zero-crossing option is enabled in the simulator.

- `isModeEnabled(block)`: Indicates if handling mode is enabled by the simulator.

- `GetXpropPtr(block)`: Get the vector of the xproperty assigned to the block by the simulator. This vector is used to define if a continuous-state is differential or algebraic.

- `GetAlphaPt(block)`: Get the actual value of alpha (α) when computing the Jacobian. Note that the Jacobian of the DAe $0 = F(\dot{x}, x)$ is computed as follows: $J = \alpha \frac{\partial F(\dot{x},x)}{\partial x} + \beta \frac{\partial F(\dot{x},x)}{\partial \dot{x}}$.

- `GetBetaPt(block)`: Get the actual value of beta (β) when computing the Jacobian.

- `isinTryPhase(block)`: Indicates if the block is being called by the numerical solver. In this case, the value given to the block is not a definitive value.

- `areModesFixed(block)`: Indicates that if the block is being called with mode values fixed or relaxed (can be changed/updated by the block).

- `isZeroCrossing(block)`: Indicates if the block is being called at a zero-crossing event.

- `isDiscreteEvent(block)`: Indicates if the block is being called at a discrete event.

- `isMeshPoint(block)`: Indicates if the block is being called at a mesh-point.

- `isDiscreteLeftLimit(block)`: Indicates if the block is being called at a left-limit of a discrete event.

- `isZeroCrossingLeftLimit(block)`: Indicates if the block is being called at a left-limit of a zero-crossing event.

- `isExitInitialization(block)`: Indicates if the block is being called at the end of the initialization.

- `isSimulatorInInitializationPhase(block)`: Indicates if the simulator is still in the initialization stage.

- `isSimulatorInTerminationPhase(block)`: Indicates if the simulator is in the termination stage.

- `Coserror(block,msg,...)`: Emits an error message. This message usually follows an error in the block and is followed by a return from the block.

- `Coswarning(block,msg,...)`: Emits a warning message.

- `Cosmessage(block,msg,...)`: Emits a purely informative message.

- `GetSimulatedModelName(block)`: returns the name of the model being simulated.

- `GetSimulatedModelFilePath(block)`: returns the path of the model being simulated.

- `GetSimulatedModelTempDir(block)`: returns the path of the temporary folder for the current model.

- `isInDebugMode(block)`: Indicates if the simulation is done in the debug mode.

- `Discard(block)`: This The block can indicate the numerical solver that the state derivative cannot be computed and the solver should take smaller step size. This feature is used in `flag= VssFlag_Derivatives`, for ther situation is treated as error.

- `isinRepeatEventBlock(block)`: Indicates if this call to the block is the repetition of a previous event.

9.2 Block builder

The construction of simulation functions has been presented in the previous section. A `C` simulation function can be associated with a **CCustomBlock** and be used in **Twin Activate** models. The **CCustomBlock** however has a generic user interface in the BDE which cannot be customized. The customization can be done by hiding this block inside a masked Super Block, however the level of customization is limited. A new basic block provides a much higher level of customization, as it will be seen in the next section.

Basic blocks and programmable super blocks can be constructed using the Block builder tool. These two types of blocks are similar at the BDE level and so is their construction. The block must provide its graphical and strcutural properties: parameters, numbers of ports of different types, block size, color, icon, etc., in an XML file. It must also provide in the case of the basic block: the list of simulation functions used by the block and three **OML** functions instantiating the block. In the case of the programmable super block, only one **OML** function is required.

By using the Block builder, user needs not worry about creating XML files or even defining **OML** functions. He/she needs simply provide the required information and the block builder creates the XML file. For the **OML** functions, only part of the code must be provided, the rest is auto-generated by the Block builder.

The Block builder interface contains 7 tabs:

1. Label: used to define the block default name, its location and whether or not it is visible by default.

2. Atom: used to define the internal properties of the block. Its definition depends on whether the block is a basic block or a prgorammable super block. Both cases will be discussed in details later.

3. Properties: used for documenting the block.

4. Parameters: used to define the number, type, name and default values of the block parameters. Callback functions can also be associated with parameters.

5. Graphics: used to set the block shape, size and color.

6. Ports: used to define the number of ports, their types and locations.

7. Icon: used to associate an image to the block icon. A second image may be used when the block is mirrored.

The block builder can be invoked from scratch or by selecting, in the current diagram, a basic block or a masked Super Block. In the latter case, the properties of the selected block are reflected in the Block builder interface so that if no modifications are done, the block builder creates a duplicate of the selected block (in the case the selected block is a masked Super Block, the new block will be an equivalent programmable super block). The ability to initialize the Block builder interface using an existing block is very helpful for defining new blocks: user can start with an existing block similar to the new block and edit only the differences.

Atom is where the internal structure of the block is defined. The content of all the tabs, except Atom, are straightforwad to understand and edit, and are identical for both basic blocks and programmable

super blocks.

9.2.1 Atom: Basic block

For a basic block, Atom provides the list of simuation functions. In general a block has more than one simulation function, in particular if it supports different data types. For each simulation function, the shared library and the corresponding entry point is specified. A basic block definition requires up to 3 **OML** functions. The partial code for these functions are provided in Atom.

The first function, `SetParameters`, is the main function; it creates the block structure based on block parameters and port properties. The other two functions are used to modify this structure if needed. The **OML** code that must be provided to define the `SetParameters` function assumes that the followings are availble:

- the name of the block, defined in the variable `_label`,
- the block parameter values under the name of the corresponding parameters,
- the diagram where the block is to be placed, defined in the variable `_diagram`.

For a basic block, the block added to diagram is of type "block", so the **OML** code invariably contains the following lines

```
_block = vssAddNewObject ('Block', _diagram);
vssSetBlockName (_block, _label);
```

The structure `_block` is then supplemented using provided APIs. For example simulation parameters `opar` are defined based on the value of block parameters. The code used here vary considerably depending on the properties of the new block. Instead of providing an exhaustive list of APIs and their usage, we encourage users to examine the codes associated with existing blocks, in particular blocks having similar properties to that of the new block. By doing so, the creation of the new code is reduced to changing a few lines of an existing code.

The `SetParameters` function is automatically created based on the user provided **OML** code mainly by adding systematic tests so that in case of error, an informative error message can be generated. This function is called in the model evaluation phase, which constructs the model structure used for compilation and simulation.

The two other functions are used to set the values not set by the `SetParameters` function in the block structure. The non-set values correspond to undefined size and data types of block input and ouput ports. The `SetParameters` function in general specifies all the properties of the block structure but it can leave out some porperties, to be found by the compiler. In particular the size of an input may be left unknown. This is done by setting the size vector to negative values, $[-1, -2]$. This means the input has totally arbitrary size. If another input or an output size is also set to $[-1, -2]$, then the latter size is also arbitray but equal to the former. If an input size is set to $[-1, -1]$, then the matrix is a square matrix of arbitrary size. Negative value means unknown, but the same negative values mean same sizes. This way constraints on block port sizes can be imposed for many common blocks. For example the matrix transpose block has input size $[-1, -2]$ and output size $[-2, -1]$. This captures all the constraints imposed by the block on its input and output.

The compiler figures out the actual port sizes by taking into account these constraints and the way the blocks are interconnected. Similarly block port data types may be set to arbitrary and the compiler deduces the actual types. Once the compiler completes the computation of block input/output data sizes and types, the block **OML** function `Reset` is called with argument the block structure including

the actual port data sizes and types. This function can then modify other block properties if needed. In most cases, the block `Reset` function selects the appropriate simulation function based on the port data types. The `Reset` function is defined in Atom based on user-supplied code.

The last **OML** function associated with a basic block is `SetIO`. This function is defined when the constraints on port data types and sizes cannot be imposed by using negative numbers. For example a block may be such that the size of its input is arbitrary and the size of its output is one bigger than the size of its input. These "exotic" constraints cannot be imposed using negative size values. In such cases the block can provide a `SetIO` function working on the block structure (constructed previously by `SetParameters`. This function, based on current status of port sizes and types, may replace negative port sizes and types by positive values. This function may be called many times during compilation. For the example where the output is one bigger than the input, the function can fully define the port sizes if either the input or the output size is known, when the function `SetIO` is called.

Most user defined blocks don't require `SetIO` and `Reset` functions and the associated code in Atom should be left empty.

9.2.2 Atom: Programmable Super Block

For programmable super blocks, there is no simulation function to define and there is only the `SetParameters` function that needs to be defined in Atom. In this case the content being defined is a "subsystem" and the **OML** code is used to construct the diagram inside the block. The code contains:

```
_block = vssAddNewObject('SubSystem',_diagram);
_diagr= vssCreateObject('Diagram');
```

and then goes on populating `_diagr` with blocks and links constructing a diagram.

The definition of a block inside the diagram is done by defining the block parameters by an **OML***struct*, loading the block from the corresponding library and instantiating it. Consider for example the block **DiscreteDelay**, which has parameters `init_cond`, `typ`, and `externalActivation`. This block can be instantiated in the diagram `_diagr` for example as follows

```
 _params=struct();
_params.init_cond=z0;
_params.typ='double';
_params.externalActivation=0;
vssLoadBlock('system','Dynamical','DiscreteDelay');
_vss._palettes.('_system').('Dynamical').('DiscreteDelay').setparams(_diagr,'DiscreteDelay',_params);
```

Here the initial condition is set to `z0`, which must be a parameter of the programmable super block.

Links are created using the **OML** function `vssAdd_Link`. The following instruction for example connects the first ouptut of the **Input** block to the first output of the **DiscreteDelay** block by a regular link:

```
vssAdd_Link(_diagram,'Input','DiscreteDelay',1,1,0,1,1);
```

When the Block builder is invoked with a masked Super Block, the `SetParameters` function of the block is generated automatically containing the code that constructs the diagram inside the Super Block. In this case the diagram is fixed.

A fixed-diagram programmable super blocks is not particularly useful because it can also be implemented by a standard masked Super Block. It can however be a good starting point for defining the `SetParameters` function of a general programmable super block where the diagram inside and the number of ports can depend on block parameters. This process is illustrated through the following example.

Example

Here the problem is to create a shift register block where each register value is exposed through a separate output. The number of registers (and outputs) is thus variable, equal to `nout`, given as a block parameter. In case of three registers/outputs (`nout=3`), the diagram is simple:
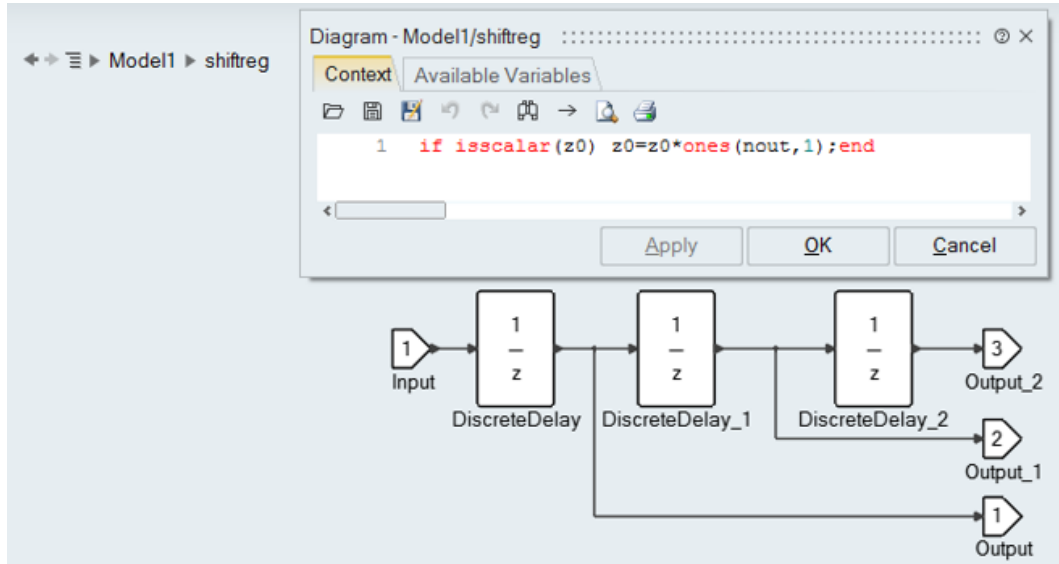
This Super Block will be used as a starting point for the constrcution of the programmable super block.

First, the Super Block is prepared to be masked by defining the undefined variables `z0` and `nout` in the Initialization script of the model. Auto-masking can then be applied:
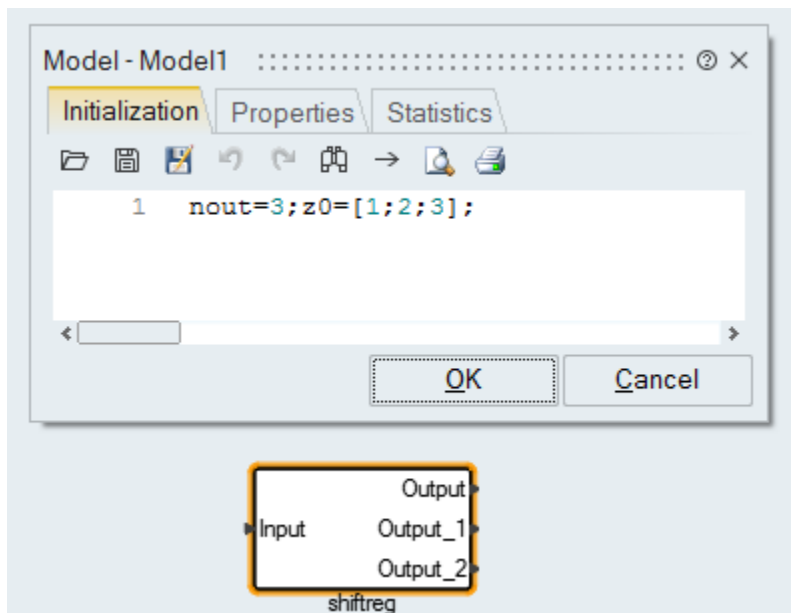
which yields:
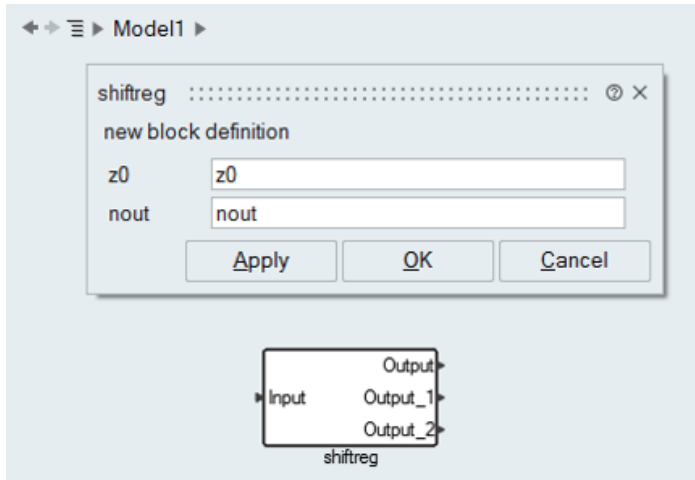
Now that the Super Block is masked, block builder can be applied to create a new block. The Block builder now contains a good starting point for defining the new block.

Some of the properties must be modified for the new block. In particular the number of ports in the Ports tab. Originally the ports are fixed:
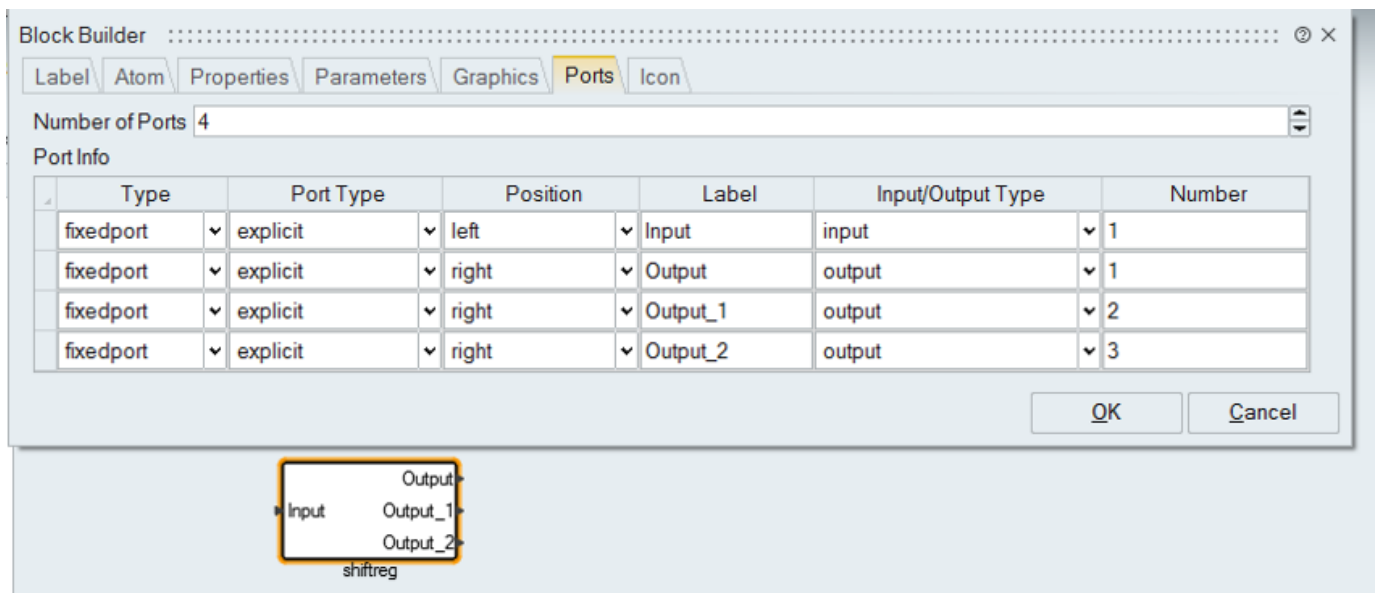
The Ports tab must be edited to allow the number of outputs to be variable, in particular `nout`:
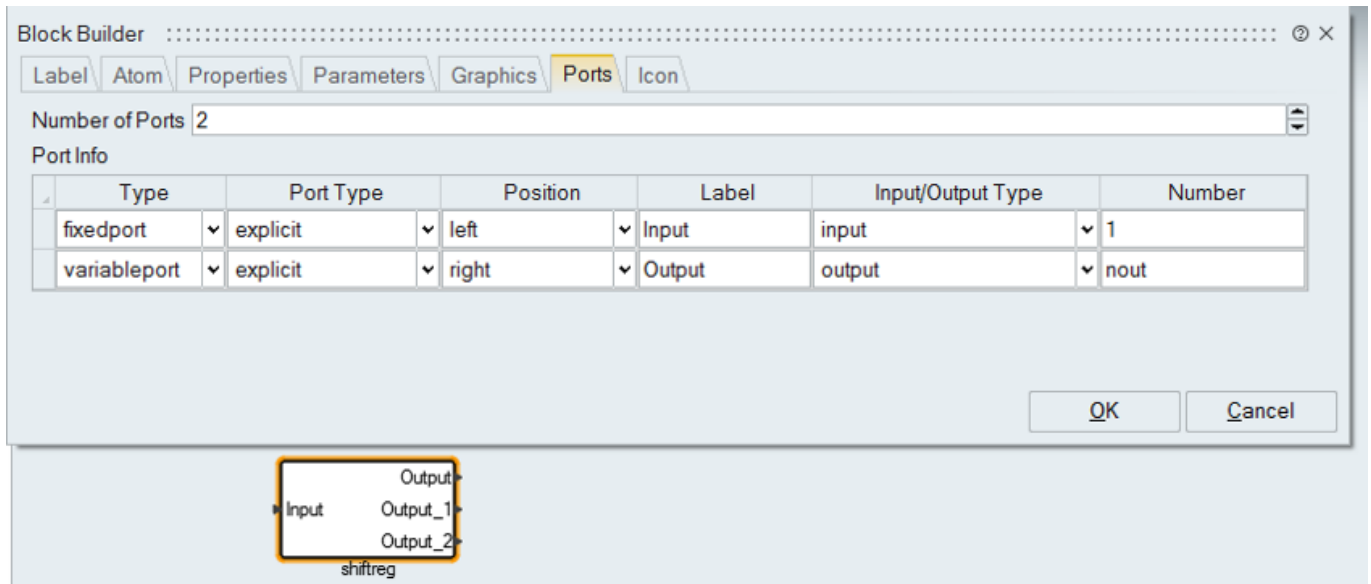
The Parameters tab must be edited to change the property of the parameter `nout` to make it of type "Number". Structural properties such as the number of ports can only be set by parameters of type "Number".
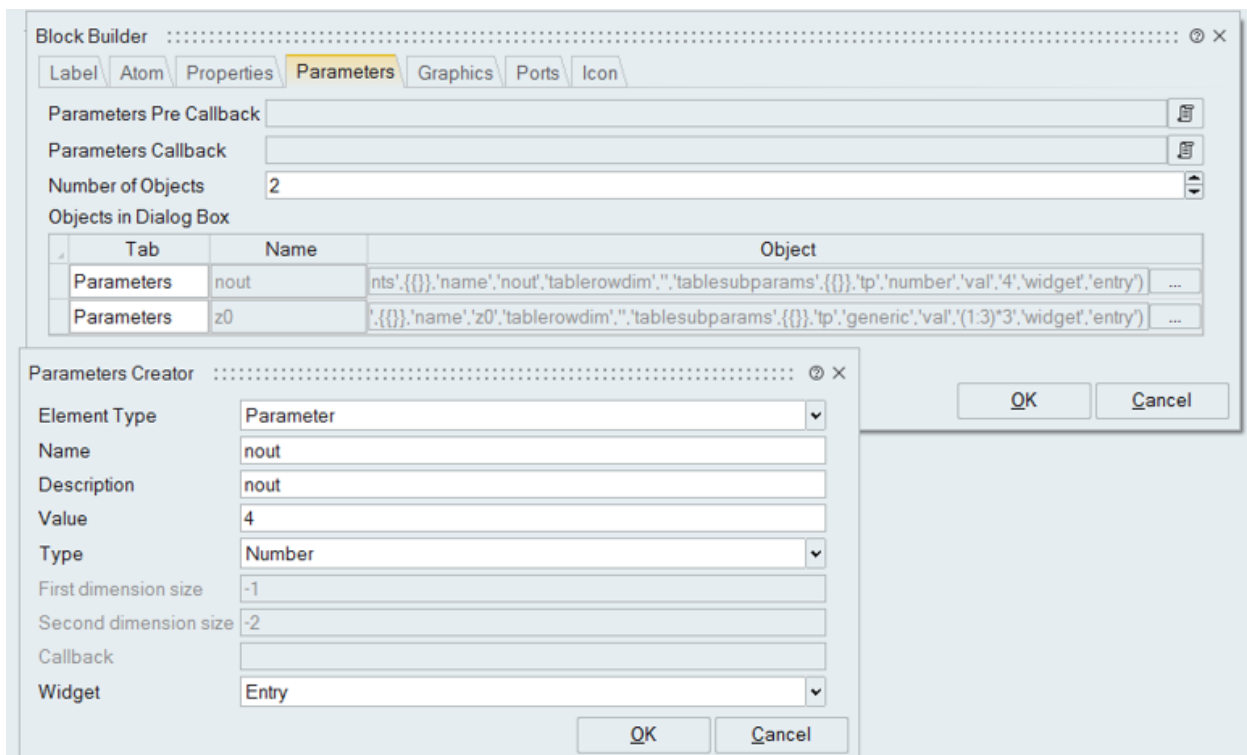
The main modification to be made is of course in the `SetParameters` code (inside the Atom tab) which defines the diagram inside the Super Block. The current code constructs the diagram by instantiating blocks and connecting them for the case `nout=3`. Note that no graphical properties are used at this point: the diagram does not have a corresponding graphics. The code gnerated by the Block builder for the fixed diagram is as follows:

```
function _setdiagram_1 (_diagram, _label, __fenv__)
_curdiagram=_diagram;
_curlabel=_label;
```

```
importenv(__fenv__);
_diagram=_curdiagram;
_label=_curlabel;
clear _curdiagram;
clear _curlabel;
global _vss;
global __OBJ_ERR_;
_parent = __OBJ_ERR_.('block');
__OBJ_ERR_.('type')='context';
if isscalar(z0)  z0=z0*ones(nout,1);end
__OBJ_ERR_.('block')=vssConstructBlockFullName(_parent,'Output_2');
__OBJ_ERR_.('type')='parameters';
_params=struct();
_params.portNumber=3;
_params.insize=[-1;-2];
_params.intyp='inherit';
vssLoadBlock('system','Ports','Output');
_vss._palettes.('_system').('Ports').('Output').setparams(_diagram,'Output_2',_params);
__OBJ_ERR_.('block')=vssConstructBlockFullName(_parent,'DiscreteDelay_2');
__OBJ_ERR_.('type')='parameters';
_params=struct();
_params.init_cond=z0(3);
_params.typ='double';
_params.externalActivation=0;
vssLoadBlock('system','Dynamical','DiscreteDelay');
_vss._palettes.('_system').('Dynamical').('DiscreteDelay').setparams(_diagram,'DiscreteDelay_2',_params);
__OBJ_ERR_.('block')=vssConstructBlockFullName(_parent,'Output_1');
__OBJ_ERR_.('type')='parameters';
_params=struct();
_params.portNumber=2;
_params.insize=[-1;-2];
_params.intyp='inherit';
vssLoadBlock('system','Ports','Output');
_vss._palettes.('_system').('Ports').('Output').setparams(_diagram,'Output_1',_params);
__OBJ_ERR_.('block')=vssConstructBlockFullName(_parent,'Output');
__OBJ_ERR_.('type')='parameters';
_params=struct();
_params.portNumber=1;
_params.insize=[-1;-2];
_params.intyp='inherit';
vssLoadBlock('system','Ports','Output');
_vss._palettes.('_system').('Ports').('Output').setparams(_diagram,'Output',_params);
__OBJ_ERR_.('block')=vssConstructBlockFullName(_parent,'Input');
__OBJ_ERR_.('type')='parameters';
_params=struct();
_params.portNumber=1;
_params.outsize=[-1;-2];
_params.outtyp='inherit';
_params.dept=0;
vssLoadBlock('system','Ports','Input');
_vss._palettes.('_system').('Ports').('Input').setparams(_diagram,'Input',_params);
__OBJ_ERR_.('block')=vssConstructBlockFullName(_parent,'Block');
__OBJ_ERR_.('type')='parameters';
_params=struct();
vssLoadBlock('system','Links','Split');
_vss._palettes.('_system').('Links').('Split').setparams(_diagram,'Block',_params);
__OBJ_ERR_.('block')=vssConstructBlockFullName(_parent,'DiscreteDelay_1');
__OBJ_ERR_.('type')='parameters';
_params=struct();
_params.init_cond=z0(2);
_params.typ='double';
_params.externalActivation=0;
vssLoadBlock('system','Dynamical','DiscreteDelay');
_vss._palettes.('_system').('Dynamical').('DiscreteDelay').setparams(_diagram,'DiscreteDelay_1',_params);
__OBJ_ERR_.('block')=vssConstructBlockFullName(_parent,'Split');
__OBJ_ERR_.('type')='parameters';
_params=struct();
vssLoadBlock('system','Links','Split');
```

```
_vss._palettes.('_system').('Links').('Split').setparams(_diagram,'Split',_params);
__OBJ_ERR_.('block')=vssConstructBlockFullName(_parent,'DiscreteDelay');
__OBJ_ERR_.('type')='parameters';
_params=struct();
_params.init_cond=z0(1);
_params.typ='double';
_params.externalActivation=0;
vssLoadBlock('system','Dynamical','DiscreteDelay');
_vss._palettes.('_system').('Dynamical').('DiscreteDelay').setparams(_diagram,'DiscreteDelay',_params);
vssAdd_Link(_diagram,'Output_2','DiscreteDelay_2',1,1,1,0,1);
vssAdd_Link(_diagram,'DiscreteDelay_1','Block',1,1,0,1,1);
vssAdd_Link(_diagram,'Block','Output_1',1,1,0,1,1);
vssAdd_Link(_diagram,'DiscreteDelay_2','Block',1,2,1,0,1);
vssAdd_Link(_diagram,'Input','DiscreteDelay',1,1,0,1,1);
vssAdd_Link(_diagram,'DiscreteDelay','Split',1,1,0,1,1);
vssAdd_Link(_diagram,'Split','DiscreteDelay_1',1,1,0,1,1);
vssAdd_Link(_diagram,'Split','Output',2,1,0,1,1);
end
_block = vssAddNewObject('SubSystem',_diagram);
_diagr= vssCreateObject('Diagram');
_setdiagram_1(_diagr,_label,getcurrentenv());
vssSet_SubSystem(_block,1,3,0,0,0,_label,_diagr);
```

The variable $\_\_OBJ\_ERR\_$ is used to anchor the error locations for generating precise error messages. In the case of programmable super blocks, this is in general not very useful and it is better to reference all error message to the parent block (the programmable super block). User in general does not know about the content of this block.

The above code, which corresponds to the fixed diagram with three delays and outputs, is extended so that it can work for arbitrary `nout` number of delays and outputs. Note that the individual error tags are removed so that all errors are referred to the parent. The script used in the context of the Super Block is also extended to produce better error messages:

```
function shiftreg_setdiagram_1 (_diagram, _label, __fenv__)
_curdiagram=_diagram;
_curlabel=_label;
importenv(__fenv__);
_diagram=_curdiagram;
_label=_curlabel;
clear _curdiagram;
clear _curlabel;
global _vss;
global __OBJ_ERR_;
_parent = __OBJ_ERR_.('block');
__OBJ_ERR_.('type')='context';
if isscalar(z0) z0=z0*ones(nout,1); elseif length(z0)~=nout error('size of z0 must match nb of outputs.');end
__OBJ_ERR_.('block')=vssConstructBlockFullName(_parent,'DiscreteDelay');
_params=struct();
_params.init_cond=z0(1);
_params.typ='double';
_params.externalActivation=0;
vssLoadBlock('system','Dynamical','DiscreteDelay');
_vss._palettes.('_system').('Dynamical').('DiscreteDelay').setparams(_diagram,'DiscreteDelay',_params);
_params=struct();
vssLoadBlock('system','Links','Split');
_vss._palettes.('_system').('Links').('Split').setparams(_diagram,'Split_1',_params);
_params=struct();
_params.portNumber=1;
_params.outsize=[-1;-2];
_params.outtyp='inherit';
_params.dept=0;
vssLoadBlock('system','Ports','Input');
_vss._palettes.('_system').('Ports').('Input').setparams(_diagram,'Input',_params);
_params=struct();
_params.portNumber=1;
_params.insize=[-1;-2];
_params.intyp='inherit';
```

```
vssLoadBlock('system','Ports','Output');
_vss._palettes.('_system').('Ports').('Output').setparams(_diagram,'Output',_params);
vssAdd_Link(_diagram,'Input','DiscreteDelay',1,1,0,1,1);
vssAdd_Link(_diagram,'DiscreteDelay','Split_1',1,1,0,1,1);
vssAdd_Link(_diagram,'Split_1','Output',2,1,0,1,1);
for i=2:nout-1
  _params=struct();
  _params.init_cond=z0(i);
  _params.typ='double';
  _params.externalActivation=0;
  vssLoadBlock('system','Dynamical','DiscreteDelay');
  _vss._palettes.('_system').('Dynamical').('DiscreteDelay').setparams(_diagram,['DiscreteDelay_',...
    num2str(i-1)],_params);
  _params=struct();
  _params.portNumber=i;
  _params.insize=[-1;-2];
  _params.intyp='inherit';
  vssLoadBlock('system','Ports','Output');
  _vss._palettes.('_system').('Ports').('Output').setparams(_diagram,['Output_',num2str(i-1)],_params);
  _params=struct();
  vssLoadBlock('system','Links','Split');
  _vss._palettes.('_system').('Links').('Split').setparams(_diagram,['Split_',num2str(i)],_params);
  vssAdd_Link(_diagram,['Split_',num2str(i-1)],['DiscreteDelay_',num2str(i-1)],1,1,0,1,1);
  vssAdd_Link(_diagram,['DiscreteDelay_',num2str(i-1)],['Split_',num2str(i)],1,1,0,1,1);
  vssAdd_Link(_diagram,['Split_',num2str(i)],['Output_',num2str(i-1)],2,1,0,1,1);
end
_params=struct();
_params.init_cond=z0(nout);
_params.typ='double';
_params.externalActivation=0;
vssLoadBlock('system','Dynamical','DiscreteDelay');
_vss._palettes.('_system').('Dynamical').('DiscreteDelay').setparams(_diagram,['DiscreteDelay_',...
   num2str(nout-1)],_params);
_params=struct();
_params.portNumber=nout;
_params.insize=[-1;-2];
_params.intyp='inherit';
vssLoadBlock('system','Ports','Output');
_vss._palettes.('_system').('Ports').('Output').setparams(_diagram,['Output_',num2str(nout-1)],_params);
vssAdd_Link(_diagram,['DiscreteDelay_',num2str(nout-1)],['Output_',num2str(nout-1)],1,1,0,1,1);
vssAdd_Link(_diagram,['Split_',num2str(nout-1)],['DiscreteDelay_',num2str(nout-1)],1,1,0,1,1);
end
_block = vssAddNewObject('SubSystem',_diagram);
_diagr= vssCreateObject('Diagram');
shiftreg_setdiagram_1(_diagr,_label,getcurrentenv());
vssSet_SubSystem(_block,1,nout,0,0,0,_label,_diagr);
```

The new repeated blocks are named by extensions _2, _3, up to nout. This is not needed for links which are not named.

After creating the first layer of blocks, the repeated structure is created within the following for loop:

```
for i=2:nout-1
  ....
end
```

The final layer is created after the for loop. Another change needs to be made in the code to change the number of outputs of the new block:

```
vssSet_SubSystem(_block,1,3,0,0,0,_label,_diagr);
```

is replaced by

```
vssSet_SubSystem(_block,1,nout,0,0,0,_label,_diagr);
```

indicating that the block has nout outputs and not 3.

The newly created block can be moved to a new model and its parameters can be set to desired values:
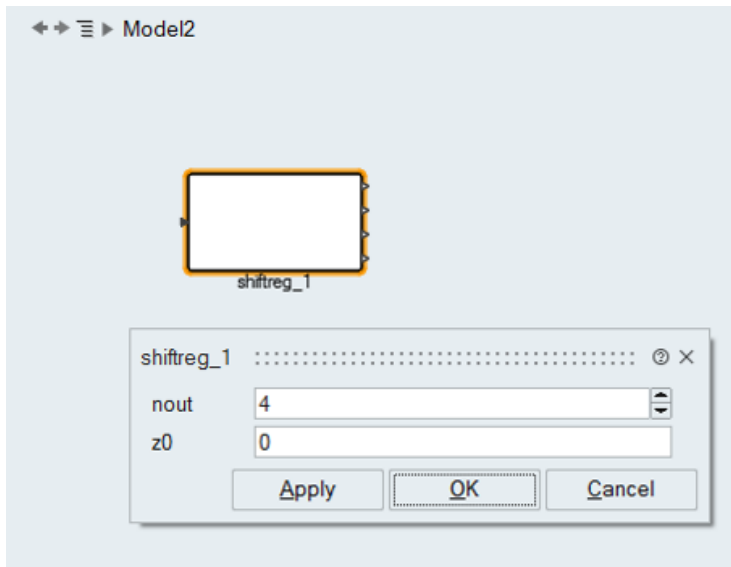
and tested:
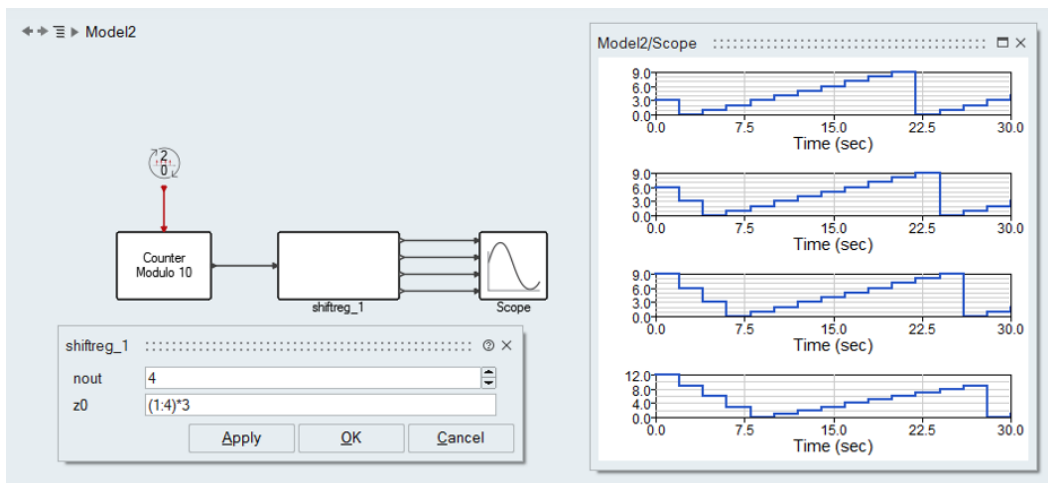
Chapter 10

Twin Activate hybrid simulator and its interface with numerical solvers

Twin Activate is a simulator for hybrid systems. A hybrid system is composed of discrete-time and continuous-time subsystems in which the values of continuous-time variables are governed by differential equations and discrete-time variables are updated at time instants called *event times*. In **Twin Activate** several numerical solvers have been interfaced with the simulator to integrate the differential equations. These numerical solvers have many control parameters that should be used to optimize the simulation in different situations. Indeed, in the hybrid context, controlling the solver, which should be done automatically and should remain transparent to the user, is a complex matter due to the interactions between the continuous-time dynamics and the rest of the system. In this document, the simulator of **Twin Activate**, the numerical solvers and the interface between the simulator and the numerical solvers will be discussed.

10.1   Introduction

A hybrid system is a dynamical system that involves the interactions of continuous states and discrete states. As an example, consider a bouncing ball which is a hybrid system since the state variables (position and speed) vary continuously according to Newton laws when falling, but they have a discrete change (speed is reversed) when entering in collision with the ground. Jump in state and discontinuities are basic phenomena that cannot be represented and analyzed by methods elaborated either in continuous-time or in discrete-time system theory. In a hybrid system composed of continuous-time and discrete-time subsystems, each subsystem can interact with the other, e.g., if a discrete-time variable used in a differential equation is changed all the continuous-time variables that depend on this value may change. This would imply a discontinuity in these variables.

The discrete-time part of a hybrid system is activated by events and events introduce discontinuities in continuous-time variables or in their time derivatives. Events may come from several sources such as intervention by human operators or from an input control or may be programmed to be generated autonomously.

There are two event types that are more important. They are events that occur at a given time, i.e., a predictable event (or time events), and those which appear when a variable reaches a certain threshold value, i.e., an unpredictable (or a zero-crossing event).

The underlying hybrid formalism in **Twin Activate** allows modeling very general hybrid systems: systems including continuous, discrete-time and event based behaviors. **Twin Activate** includes a graphical editor for constructing models by interconnecting blocks, representing predefined or user-defined functions, a compiler, a simulator, and code generation facilities. The graphical editor allows creating and connecting blocks selected from block palettes. A **Twin Activate** block diagram is composed of blocks and connection links. There are two connection link types; regular and activation links. The regular links represent continuous-time or discrete-time signals or variables in the model, while the activation links transmit activation timings information. A block corresponds to an operation and by interconnecting blocks through links, a model or an algorithm is constructed. These blocks represent elementary systems that can be used as model building blocks. They can have several inputs and outputs, continuous-time states, discrete-time states, zero-crossing functions, etc. Note that it is not possible to treat an activation link as a regular link or to interconnect an activation link with a regular link. In order to get an idea what a **Twin Activate** model looks like, a model of a simple control system implemented in **Twin Activate** is shown in the following figure.
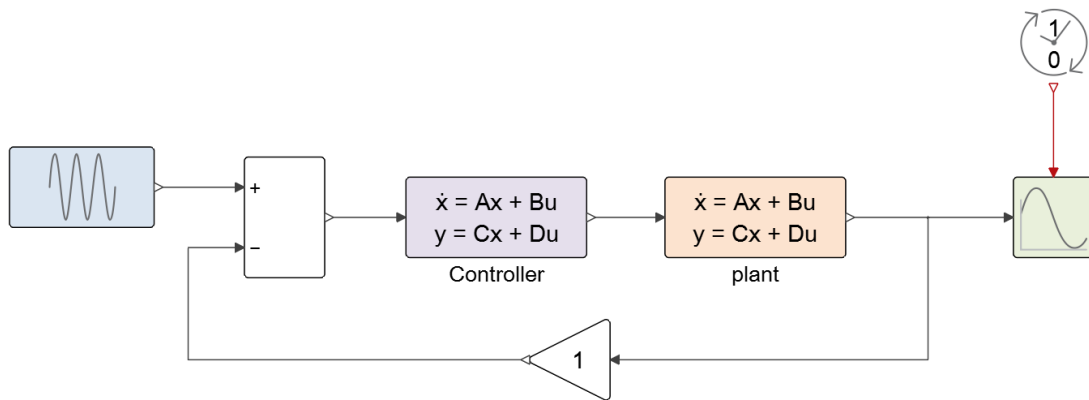
Figure 10.1: Simple control model (simple_control1.scm)

In this figure (10.1) the clock block generates a periodic activation signal (events) that activates the Scope block. At event times the scope reads its input signal and displays it.

Here is an example of a pure discrete-time system where the state is updated by a periodic clock with period T.

$$x(k + 1) = \left\{ \begin{array}{ll} x(k) + u(k) & \text{if } u(k) > 0 \\ x(k) & \text{otherwise} \end{array} \right.$$

$$x(1) = 0$$

The implementation of this model and its simulation result in **Twin Activate** are given in following figures (10.2, 10.3). In this model $u(k)$ is generated by a random signal generator block with output values between $[-1, +1]$.
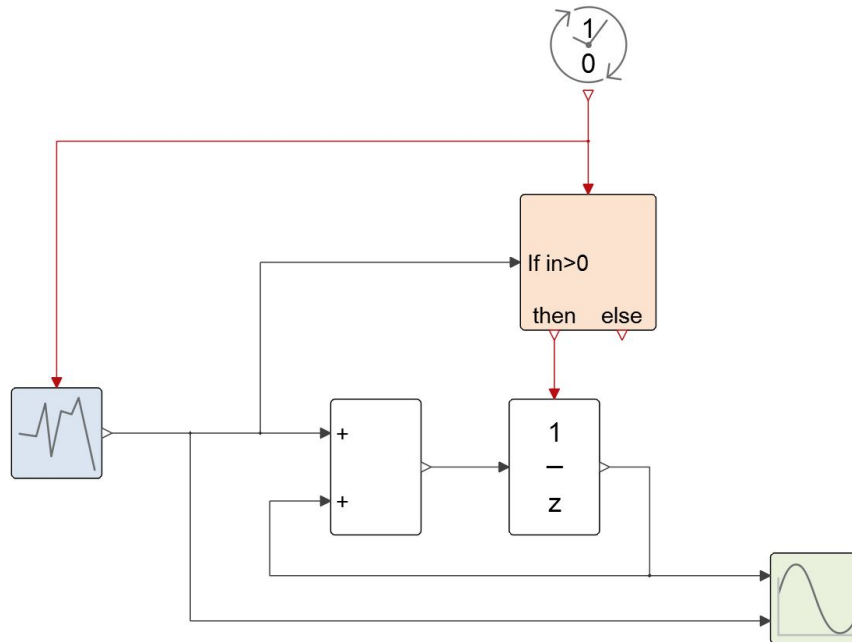
Figure 10.2: Discrete model (discretem.scm)
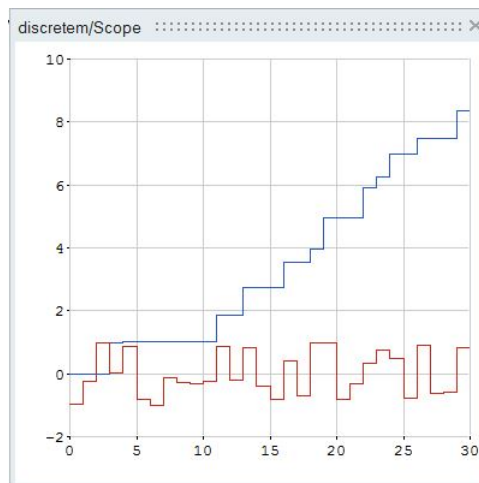
Figure 10.3: Results of Discrete model (discretem.scm)

Now consider the model of the bouncing ball. The model implemented in **Twin Activate** and its simulation result are given in the following figures (10.4, 10.5). In this model when the altitude of the ball (x) becomes negative, the `zero-crossing` block generates an event that resets the `Linear Continuous-time` block with new state values.

$$\begin{cases} \dot{x} & = v \\ \dot{v} & = -9.8 \end{cases}$$
$$\text{when } (x < 0) \text{ then } v = -0.9\, v$$
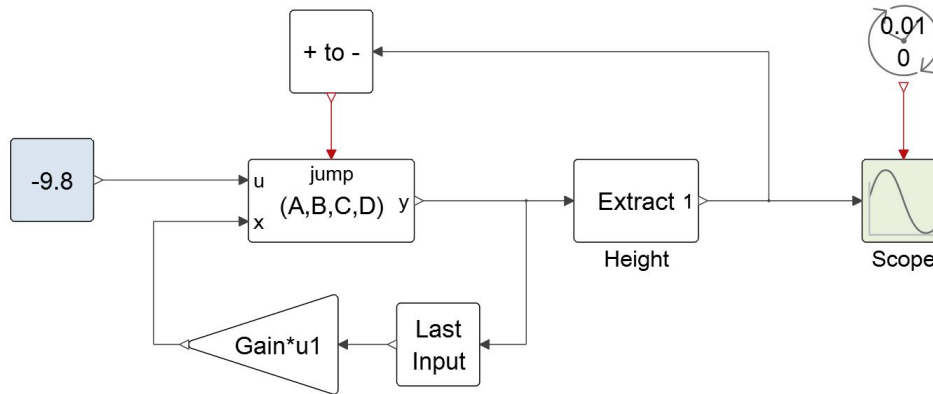
Figure 10.4: Simple bouncing ball model (simple_bball.scm)
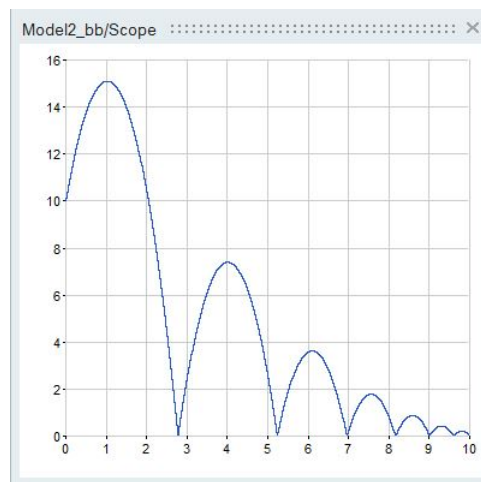
Figure 10.5: Results of Simple bouncing ball model

10.2   Simulation of an hybrid model in Twin Activate

The simulator of **Twin Activate** is designed to generate the solution of the **Twin Activate** model, which is very often a hybrid system. A hybrid system simulator should be able to perform, discrete-time scheduling, continuous-time integration, and interactions between discrete-time and continuous-time part (zero-crossing events and event handling).

Simulation of a hybrid system is composed of two phases: the discrete-phase and the continuous-phase. In the discrete phase, event scheduling, event activating and state updating (discrete-time and continuous-time) is done, and the simulation time does not advance. In the continuous-phase, the numerical solver integrates the differential equation and the simulation time advances. Integration of the continuous-time system is done from one event time to the next event time. The exact time instant of an event may not be known in advance, hence during the continuous-time phase an event detection process is being performed to stop the continuous-phase and enter into the discrete-phase.

The continuous-time parts of a hybrid system can be modeled either by Ordinary Differential Equations

(ODE), i.e.,

$$\dot{x} = f(x, u, t)$$
$$y = h(x, u, t)$$

where x represents the state variable, $\dot{x}$ denotes the derivative of x with respect to the time variable t, u is the input vector variable, and y defines the output vector, or by Differential Algebraic Equations (DAE)

$$\dot{x}_1 = f(x_1, x_2, u, t)$$
$$0 = g(x_1, x_2, u, t)$$
$$y = h(\dot{x}_1, x_1, x_2, u, t)$$

The first equation is the differential part and the second is the algebraic part of the DAE. This equation is a semi-explicit DAE where differential and algebraic parts can be decomposed. In semi-explicit index-1 DAE, the variable whose derivative is present in the equation is referred to as differential equation and others are called algebraic variables. This equation can be cast in the form

$$0 = F(\dot{x}, x, u, t)$$
$$y = h(\dot{x}, x, u, t)$$

where $x = (x_1, x_2)$. This system is called a fully implicit DAE. Note that if partial derivative of F with respect to $\dot{x}$ is non-singular then it is possible to solve for $\dot{x}$ in order to obtain an ODE. However, if it is singular, this is no longer possible and the solution must satisfy certain algebraic constraints.

DAEs are characterized by their index. The index of a DAE is the smallest number of differentiations required to obtain an ODE by algebraic manipulations. In general, the higher the index, the greater the numerical difficulty one encounters when trying to solve the system numerically. In a semi-explicit index-1 DAE, variables whose derivatives appear in DAE are called differential variables and the others are called algebraic.

Modeling a hybrid system very often leads to a differential equation with discontinuities or a multi-mode system such as an automaton (see Documentation of the `Automaton` block). In simulation of a hybrid system, state variables or their time derivative may be discontinuous at some time instants. In such a situation special care has to be taken to solve the model by a numerical solver. Some numerical solvers such as `Radau` or `Lsoda` use variable order and variable step-size methods to integrate. In simulation of a discontinuous model, such solvers may repeat several times the same step with smaller step-size to obtain a coherent derivative and meet the accuracy requirements at the discontinuity point. For some problems, the solver may give up with an error message when the step gets too small, and in other cases the solver may continue with an erroneous result. In general, such kind of solvers require that the system be sufficiently smooth over an integration period (at least be C_0). This means that simulator must make sure to stop and reinitialize the solver at each potential point of non-smoothness (discontinuity, discontinuity in the derivative, etc.). The discontinuity handling problem is solved in **Twin Activate** by introducing the `mode` variable in **Twin Activate**.

A multi-mode system is a way of describing non-smooth systems in terms of a finite number of smooth systems. The idea is to divide the state space of the system into different regions and associate a `mode` to each region. It is assumed that the system is described in terms of a single smooth model within each region. A simple example of a multi-mode system, where the model in each mode is described as DAE, is:

$$0 = \begin{cases} f_1(\dot{x}, x) & \text{if } g(x) > 0 \\ f_2(\dot{x}, x) & \text{otherwise} \end{cases}$$

This system has two modes: the first one is active when $g(x) > 0$ and the second is when the condition is not true. The switching between two modes of the system (f_1 and f_2) may cause discontinuity in the signals. As a result the system may not be integrated by numerical solvers. Numerical solvers do not like discontinuities in the differential equation and they assume that the state variables and their first derivatives are continuous. If the signal is not continuous, special precautions must be taken at the discontinuity points. To overcome this problem, the discontinuity should be detected and the numerical solver be reinitialized after the discontinuity point. In order to detect and localize the discontinuity time, the simulator uses zero-crossing functions that cross zero at the discontinuity point. For example, for the above model, the zero-crossing function to be used by the simulator is $g(x)$. However, to localize this point, the numerical solver should simulator say f_1 and step over the discontinuity point and use the other model (f_2). Using the model beyond the discontinuity point (f_2) normally results in repeated step failure and a reduction in integration step-size to meet accuracy requirements. For some problems especially models with hard discontinuity the solver gives up with an error message when the step gets too small, and in some other cases the solver may continue with erroneous results. In order to solve this problem we have associated a `mode` variable with each zero-crossing function. We use the `mode` variable to choose a single ODE/DAE when the numerical solver is called. The assigned ODE/DAE does not change till the next discontinuity detection or event. For example, for the above model the general solution has been illustrated in the following figure where x_a indicates the discontinuity point or the instant where $g(x_a)$ becomes zero.
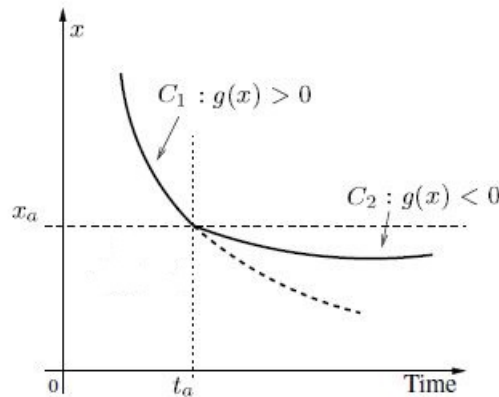
Before reaching x_a, the solver is in (`mode=1`), i.e. , it uses $0 = f_1(\dot{x}, x)$, without considering the $f_2(\dot{x}, x)$ equation even when the solver needs to compute $f_1(\dot{x}, x)$ for $g(x) > 0$ (the thick dashed line). The solver employs the $f_1(\dot{x}, x)$ equation to integrate until it detects a zero-crossing. Then the solver localizes the crossing point and stops just at the discontinuity point (i.e., at $x = x_a$). After detecting the zero-crossing and stopping the integration, the `mode` variable should be updated in order to use another model, i.e., $f_2(\dot{x}, x)$.

In general, when the numerical solver is called, the system of equations should not be changed or show discontinuity. During integration, the zero-crossing functions are monitored by the simulator. In case of any sign change in the zero-crossing functions, the `mode` variables should be updated to provide another ODE/DAE to the solver. In **Twin Activate**, a `mode` variable is assigned systematically to each zero-crossing function, characterized by `If-then-Else` conditions, e.g., to simulate the above system, **Twin Activate** considers the model in the form:

$$0 = \begin{cases} f_1(\dot{x}, x) & \text{if mode} = 1 \\ f_2(\dot{x}, x) & \text{if mode} = 2 \end{cases}$$

where `mode` variables are updated at event time instants, with the following rules.

$$\begin{array}{ll} \text{when } (g(x) > 0) \text{ is true} \quad \text{then} & \text{mode} = 1 \\ \text{otherwise} & \text{mode} = 2 \end{array}$$

It should be mentioned that any If-then-Else condition, not resulting in a discontinuity, can be used without `mode` variables. For instance, in this system the `mode` variable is not necessary.

$$0 = \begin{cases} x^2 + x + 1 & \text{if } x \geq 1 \\ 2x^2 - x + 2 & \text{otherwise} \end{cases}$$

Using `mode` variables eliminates solver difficulties over discontinuities but there is still the problem of initialization the `modes` for DAE.

Most of the problems in simulation of hybrid systems are common to both ODE and DAE solvers. However, there is an additional difficulty with the DAE case: the problem of (re)initialization and finding consistent initial conditions. Simulation of an ODE can be started from any initial state, but simulation of a DAE should be started from a consistent initial state. A DAE should be initialized with consistent initial values to be solved by a numerical solver, i.e., initial values of states and their derivatives needs to be consistent with the DAE and some other hidden equations resulted from differentiation of the DAE.

Computation of consistent initial conditions is not in general an easy problem and it becomes particularly difficult for multi-mode DAE (defined with several `modes`). The search for the initial condition in this case is not just the classical problem of finding zeros of smooth functions but it is interlaced with searching for the correct `mode` value. Indeed, each `mode` value implies a smooth function and the solution of this function may imply another `mode` value.

The classification in terms of differential and algebraic is also a key element in the study of multi-model DAEs, and in particular, the problem of finding consistent initial conditions since the classification specifies which variables are solved for. Consider, for example, the following multi-model DAE

$$\begin{cases} \dot{x} = \sin(t) \ \sin(x) \\ \text{if } (x < 1) \text{ then } y = 2 \text{ else } y = 3 \end{cases}$$

where in **Twin Activate**, it is rewritten as:

$$\text{DAE :} \qquad \begin{cases} \dot{x} = \sin(t) \ \sin(x) \\ \text{if } (\text{Mode}_1 = 1) \qquad \text{then} \quad y = 2 \quad \text{else} \quad y = 3 \end{cases}$$

$$\text{Mode changing :} \quad \begin{cases} \text{when } (x < 1) \quad \text{then} \quad \text{Mode}_1 = 1 \\ \text{when } (x \geq 1) \quad \text{then} \quad \text{Mode}_1 = 2 \end{cases}$$

In this example, x is a differential variable and its initial value is given. On the other hand, y is an algebraic variable. This multi-model DAE is well-posed and it is easy to find consistent initial conditions for it. In this case, the condition that needs to be tested involve the variable x which is given. Now consider the following multi-model DAE

$$\begin{cases} \dot{x} = x + y + z \\ \text{if } (y < 1) \quad \text{then} \quad z = 0, \text{ else } \quad z = 3 \\ \text{if } (z < 1) \quad \text{then} \quad y = 0, \text{ else } \quad y = 3 \end{cases}$$

where in **Twin Activate**, we use:

$$\text{ODE}: \quad \begin{cases} \dot{x} = x + y + z \\ \text{if } (\text{ Mode}_1 = 1 \text{) then } \ z = 0, \quad \text{else } \ z = 3 \\ \text{if } (\text{ Mode}_2 = 1 \text{) then } \ y = 0, \quad \text{else } \ y = 3 \end{cases}$$

$$\text{Zero-crossing}: \quad \begin{cases} \text{when } (y < 1) \quad \text{then } \text{Mode}_1 = 1 \\ \text{when } (y \geq 1) \quad \text{then } \text{Mode}_1 = 2 \\ \text{when } (z < 1) \quad \text{then } \text{Mode}_2 = 1 \\ \text{when } (z \geq 1) \quad \text{then } \text{Mode}_2 = 2 \end{cases}$$

This problem is more complicated because the conditions are based on the values of y and z (both algebraic variables), and the solution cannot be obtained without considering multiple scenarios. In fact, this system does not even have a unique solution for the algebraic variables in terms of the differential variables. If x is given a value, then (y, z) can be either $(0, 0)$ or $(3, 3)$.

In **Twin Activate** it is assumed that differential variables are known and the other variables should be computed. As a result, based on the dependence of discontinuity functions on these variables and derivatives, mode initialization can be complicated. Consider the the initialization of the following DAE

$$0 = F(\dot{x}_d, x_d, x_a)$$

where x_d and x_a are the differential and algebraic states. In this case, if the zero-crossing functions depend on x_a and $\dot{x}_d$, the initialization becomes complicated, because the initial values of these variables are not known and their value may change during the initialization, that causes several mode changing during the initialization.

A dynamical system which contains N Modes, say characterized by N If-then-else conditions, can start in any of the 2^N possible Modes. For the systems with large N and especially when the conditions are functions of algebraic variables, it is practically impossible to find the starting mode via testing all possible Modes. As a result, we have elaborated a search method to find the consistent Modes and initial conditions. The following algorithm has been tested and being used in **Twin Activate**:

1. Assigning the differential states (x_d) and take the guess values of x_a and $\dot{x}_d$

2. Based on x_d values and current x_a and $\dot{x}_d$ guess values, fix the modes.

3. If number of iterations exceeds nmod, return by error.

4. Store the current modes.

5. Invoke the algebraic solver to find the consistent initial conditions, *i.e.*, $x_a, \dot{x}_d$

6. Re-evaluate the zero-crossing functions and fix the modes

7. Compare new modes with stored ones, If there is any change, go to step 3

8. The modes and states are consistent. Initialization is finished and the simulation can be started.

10.3   DAE and ODE with constraints

DAE systems arise in many applications such as constrained mechanical systems. One attribute of DAE systems is the differentiation index of the system, which can be defined as the number of differentiations of each equation necessary to convert the system into an ODE system.

ODE can be considered as an index-0 DAE[1]. As an example, consider the following system

$$\dot{x} = f(x, y) \tag{10.1a}$$
$$0 = g(x, y), \tag{10.1b}$$

Differentiation of (10.1b) once gives:

$$\dot{x}\frac{\partial g}{\partial x} + \dot{y}\frac{\partial g}{\partial y} = 0 \tag{10.2}$$

If $\frac{\partial g}{\partial y}$ is not singular, then (10.1) with (10.2) can be used to compute the values of $\dot{x}$ and $\dot{y}$. Hence we now have an ODE system. Equation (10.1) is an index 1 DAE, because one differentiation yields an ODE. This process (of differentiation to obtain an ODE system from a DAE system) is called index-reduction [27].

Index-1 systems can be treated in this way, but this introduces a constraint (*i.e.,* $g(x, y) = 0$) that will not be taken into account when using a standard numerical ODE solver.

An alternative approach for index 1 problems is to treat the original system as-is, using the equation (10.1a) to solve for $\dot{x}$, and treating y as a purely algebraic variable, to be solved using the equation (10.1b). Solution of this system requires modifications to standard ODE solvers to accommodate the algebraic variables. Ideally these variables should have some error control measures applied that is similar in effect to the error control on $\dot{y}$ of the index reduced system. Advantages of the direct approach are twofold. No unnecessary state y is introduced. Fewer constraints is always better, both making the error through constraint handling smaller, and in this case removing the need for constraint handling altogether. But this method needs the numerical integrator to be modified to accommodate error control on algebraic variables. There are also standard DAE solvers such as DASSL, IDA[2], or RADAU-IIA[3] that can take the equation (10.1) as input and solve it over time. In these DAE solvers, consistent initial values of x and y are provided by the user. Some solvers can help the user to initialize the DAE by solving the initialization equation. In this case, the user should indicate which variable are differential and which are algebraic.

Compiling complex Modelica models, in particular mechanical models, very often results in high index DAEs. The DAE is usually transformed into ODEs to be simulated by ordinary ODE solvers. Simulating the ODE instead of the original DAE means some constraints in the original DAE have been ignored. Keeping the constraints and making sure they are satisfied is important to avoid drift in the solution.

Consider the overdetermined system:

$$\dot{x} = f(x) \quad x(t_0) = x_0 \tag{10.3a}$$
$$0 = \phi(x), \tag{10.3b}$$

The constraint (10.3b) is supposed to be consistent with the ODE (10.3a) in the sense that the solution of this ODE satisfies (10.3b). So theoretically, the constraint (10.3b) is redundant. However for numerical simulation, it provides valuable information that can be used by the solver to reduce numerical errors. This can be done by keeping $\phi(x)$ close to zero.

---

[1]Further information on DAEs, and numerical methods for DAEs, can be found in [16] and [23].
[2]https://computation.llnl.gov/projects/sundials/ida
[3]https://www.unige.ch/~hairer/software.html

Such constraints may be present in different situations. For example in a conservative physical system, where the total energy is conserved, the conservation of energy may be expressed as such a constraint. But the scenario that is of particular interest in **Twin Activate** is when the ODE (10.3a) is obtained by differentiating algebraic equations such as (10.3b). This is done when the original system is a DAE. The algebraic equations are differentiated until an ODE is obtained so that an ODE solver can be used for simulation. In such cases ignoring the original algebraic constraints results often in unacceptable drift in the numerical solution of the system.

For higher index systems, constraints cannot be avoided, even if only performing index-reduction to make the system index 1 DAE or ODE, so a mechanism for handling constraints is required. As an example, consider the unit length planar pendulum in Cartesian coordinates, which can be expressed by the following equations:

$$\begin{cases} \ddot{x} & = & Fx \\ \ddot{y} & = & Fy - g \\ 0 & = & x^2 + y^2 - 1 \end{cases} \tag{10.4}$$

One differentiation of the constraint in x, y gives:

$$0 = 2x\dot{x} + 2y\dot{y}$$

And a second differentiation gives:

$$0 = 2x\ddot{x} + 2y\ddot{y} + 2\dot{x}^2 + 2\dot{y}^2$$

which after replacing $\ddot{x}$ and $\ddot{y}$ from (10.4) and simplification we get:

$$0 = 2Fx^2 + 2Fy^2 - 2yg + 2\dot{x}^2 + 2\dot{y}^2$$

which gives

$$F = yg - (\dot{x}^2 + \dot{y}^2). \tag{10.5}$$

In order to fully index reduce this model, one further differentiation of (10.5) would be needed to obtain an equation that can be used to get $\dot{F}$ as a function of other states, so this problem is an index 3 system.

Leaving F in algebraic form, *i.e.*, keeping (10.5) in the system, instead of its derivative) gives us the following system of equations

$$\begin{cases} F & = & yg - (\dot{x}^2 + \dot{y}^2) \\ \ddot{x} & = & Fx \\ \ddot{y} & = & Fy - g \end{cases} \tag{10.6}$$

where x, y, $\dot{x}$, $\dot{y}$ are differential states and F is the algebraic variable. The two hidden constraints on the states are:

$$\begin{cases} 0 & = & x^2 + y^2 - 1 \\ 0 & = & 2x\dot{x} + 2y\dot{y} \end{cases} \tag{10.7}$$

In order to solve such a system with constraints, various approaches are possible:

- Simply treat the ODE and index 1 portion of the system ignoring the hidden constraints. Problem: Over time the solution will drift away from the constraints giving an inaccurate or even non-physical solution for the model.

- Use Baumgarte constraint stabilization [19], by adding correcting terms to the ODEs. Problem: This can only reduce (not eliminate) the drift for the problem. Furthermore, the parameter values for Baumgarte are not known in advance.

- Pantelides index reduction and dummy derivatives algorithms [27] , [24], [28], usually reduce the DAE index to zero or one.

- Another solution is simulating the ODE part of the system using an ODE integrator, but project back the solution onto the constraint manifold after each integrator's time step. After completion of each integrator step, the required projection is computed, and if its norm is large enough, it is applied to the solution so that the constraints are satisfied. In this method, monitoring the magnitude of the projection and integrate it into the error control mechanism is required. This method has been implemented and in **Twin Activate** and can be used in any block to check the constraints. If a block needs a constraint to be respected, the block should either provide the additional algebraic constraint equations or the necessary projection correction equations.

10.3.1 Coordinate projection

The key idea to reduce or avoid drift is to project the solution points found by the numerical solver of the index 1 DAE or ODE system back on the manifold defined by the original system. Consider the ODE with constraints (10.3). The coordinate projection method essentially consists of two steps for each integration step.

1. Suppose that x_{n-1} is a point consistent with the original system (10.3). Using x_{n-1} as the initial value, the ODE numerical solver takes a step applying some numerical integration method on the equation (10.3a), and gets the point $\tilde{x}_n$ at t_n.

2. The solution point $\tilde{x}_n$, computed by the ODE solver, is then projected orthogonally back onto the manifold (10.3b) given by constraints, *i.e.,* the projected solution is computed as the solution of (10.8)

$$\begin{cases} \|x_n - \tilde{x}_n\|_2 &= \; \underset{x_n}{\text{minimize}} \\ \phi(x_n) &= \; 0 \end{cases} \qquad (10.8)$$

   which is a nonlinear constrained least squares problem. The projection gives the orthogonal projection to the manifold to get the next point x_n. The projected value x_n is then used to advance the solution for the next step [21].

In [29], [22], [18], [17], and [23] the coordinate projection was discussed for one-step methods such as Runge-Kutta methods. In case of BDF-methods or, more generally, multi-step methods, the projection is more complex, since the correction computed by the projection method should enter into the error equation [20]. In **Twin Activate**, several solvers support the projection correction, for example the Cpode solver (see 11 and 12) which is a BDF-based solver and Radau-IIA which is a single-step solver.

The projection process simply computes the changes required for each state variable so that the current values of the system lie on the constraint manifold. Ideally this should be computed as the minimum (or near minimum) change to accomplish this, as the constraint problem is typically under-determined, so many solutions are possible.

For error-controlled integrators, the change required to move the solution back to the constraint manifold can be integrated into the error control mechanism, so if too large a change is needed, the step can be rejected, and step with a smaller step size can be attempted.

There are two ways to define constraints in **Twin Activate** blocks which will be explained in 10.3.2 and 10.3.3.

10.3.2 Algebraic cpnstraint functions

If a block in **Twin Activate** provides constraint equations as algebraic functions, then the number of these constraints should be reported to the **Twin Activate** simulator using the API at the initialization.

```
SetConstraintKind(block,CONSTRAINT);
SetNConstraint(block,nc);
```

During the simulation, the numerical solver invokes the block with the `flag=VssFlag_Projection` to retrieve the constraint values.

```
double *C=GetCorrPtr(block);
```

The block should return the `C` vector filled with the constraint residuals, where the length of `C` is `nc`. When the solution is on the manifold of the constraint, the norm of the `C` vector is zero or nearly zero.

The use of constraint functions provides the simulator with complete control over the projection process, so one can implement his own scaling method or apply a different solution technique than least squared. in **Twin Activate**, the `CPODE` solver (12) is able to handle this kind of algebraic constraints.

Pendulum model as an ODE with algebraic constraints

The pendulum model explained in section 10.3 can be implemented in a block as follows.

```
double *X=GetState(block);
double x, y, xd, yd;

if (flag==VssFlag_Initialize){
    SetConstraintKind(block,CONSTRAINT);
    SetNConstraint(block,2);
    X[0]=1;
    X[1]=0;
    X[2]=0;
    X[3]=0;
}
if (flag==VssFlag_Derivatives){
    double *XD=GetDerState(block);
    double  F, g=9.81, F;
    x  = X[0];
    y  = X[1];
    xd = X[2];
    yd = X[3];

    F =  g*y--(xd*xd + yd*yd);

    XD[0] = xd;
    XD[1] = yd;
    XD[2] = x*F;
    XD[3] = y*F - g;
}

if (flag==VssFlag_Projection){
    double *C=GetCorrPtr(block);
    x  = X[0];
    y  = X[1];
    xd = X[2];
    yd = X[3];

    C[0]= x*x + y*y - 1.0;
    C[1]= x*xd + y*yd;
}

if (flag==VssFlag_OutputUpdate){
    double *output=GetRealOutPortPtrs(block,1);
    output[0]  = X[0];
    output[1]  = X[1];
}
```

In `flag==VssFlag_Initialize`, the four states of the model, i.e., `X[i]` are initialized. In this flag, the constraint type is declared as `CONSTRAINT` which means the block provides the raw algebraic

constraint equations. In this case, the numerical solver should find the projection vector and do the constraint correction all internally. Since this constraint type has been selected, the block reports the simulator the number of algebraic constraints using the API `SetNConstraint(block,2)`. In the pendulum model, there are two raw algebraic constraints , i.e., the equation 10.7.

In `flag==VssFlag_Derivatives`, the state derivatives of the model, i.e., the equation 10.6 are evaluated and `xd[i]` are computed. In `flag==VssFlag_Projection`, the algebraic constraint equations of the model, i.e., the equation 10.7 are evaluated and `C[i]` are computed. Note that there are two constraints defined by `C[0] and C[1]` values.

In `flag==VssFlag_OutputUpdate`, the block outputs are evaluated, i.e., the x and y in equation 10.6.

10.3.3 Projection correction functions

If the block is able to directly provide the constraint corrections to the numerical solvers, then the Projection type constraint correction should be used. This constraint correction method is very usefull, because most numerical solvers are unable to exploit and utilize the pure algebraic constraints, as it is done in `CPODE(12)`. On the other hand, using the projection correction is straightforward in many single-step solvers such as Runge-Kutta family solvers[4]. If the Projection type constraint is chosen, then constraint correction values should be computed by the block and returned to the numerical solver. The provided corrections should do a full projection of the current solution back onto the constraint manifold. The projection is applied to the current continuous-time states until the states satisfy the constraints to within an error tolerance or until the maximum number of iterations is exceeded.

The correction vector provides the current local projection step for the constraint residual minimization problem 10.8. For example, in a very simple case, the correction vector can be computed as follows. First, the constraint vector C and its Jacobian $D = \frac{\partial C}{\partial x}$ are computed from the model at the current operating point x.

Then, the projection correction vector P is computed as the pseudo inverse of the matrix D, *i.e.,*

$$P = \left(D^T D\right)^{-1} D^T C$$

The size of correction vector P is the same size of continuous-time state vector x. Note that this is just for illustration purposes and here we have not considered variable scaling, or cautious handling for rank deficiency. If the block is unable to compute the step, for example, when in the above equation $D^T D$ is singular, the block returns by error.

The number of constraints in a model may change during the simulation when the model configuration changes due to a discrete event. This kind of systems are called variable constraint systems. Since in a variable constraint system, the number of (active) constraints can change, we can use this kind of projection function to query the constraint corrections that are currently active. If this function returns zero, the model does not require coordinate projection. This may happen during the simulation of a variable constraint system.

If a **Twin Activate** block can handle constraints by providing the constraint corrections, the block should indicate the constraint correction type to the **Twin Activate** simulator using the API during the initialization.

---

[4]in **Twin Activate**, RADAU, CPODE(11), RK4, RK5, TRAPEZOID and EULER solvers support this kind of constraint correction.

```
SetConstraintKind(block,PROJECTION);
```

Stabilizing the pendulum model with constraint correction using state projection

The pendulum model explained in section 10.3 can be stabilized using constraint correction in a block
as follows.

```
double *XD=GetDerState(block);
    double *X=GetState(block);
    double x, y, xd, yd;
    double  g = 9.81;

    if (flag==VssFlag_Initialize){
        SetConstraintKind(block,PROJECTION);
        X[0]=1;
        X[1]=0;
        X[2]=0;
        X[3]=0;
    }

    if (flag==VssFlag_Derivatives){
        double  F;
        x  = X[0];
        y  = X[1];
        xd = X[2];
        yd = X[3];

        F =  g*y-(xd*xd + yd*yd);

        XD[0] = xd;
        XD[1] = yd;
        XD[2] = x*F;
        XD[3] = y*F - g;
    }

    if (flag==VssFlag_Projection){
        double *P=GetCorrPtr(block);
        double x_new, y_new, xd_new, yd_new, R;

        x  = X[0];
        y  = X[1];
        xd = X[2];
        yd = X[3];

        R = sqrt(x*x+y*y);

        x_new = x/R;
        y_new = y/R;
        xd_new =   xd*y_new*y_new - yd*x_new*y_new;
        yd_new = - xd*x_new*y_new + yd*x_new*x_new;

        P[0] = x_new  - x;
        P[1] = y_new  - y;
        P[2] = xd_new - xd;
        P[3] = yd_new - yd;
    }

    if (flag==VssFlag_OutputUpdate){
        double *output=GetRealOutPortPtrs(block,1);
        output[0]=X[0];
        output[1]=X[1];
    }
```

In `flag==VssFlag_Initialize`, the four states of the model, i.e., `x[i]` are initialized. In this
flag, the constraint type is declared as `PROJECTION` which means the block provides the constraint
corrections in form of projection equations. In this case, the block should find the projection vector to
push the current states back on the constraints.

In `flag==VssFlag_Derivatives`, the state derivatives of the model, i.e., the equation 10.6 are
evaluated and `xd[i]` are computed.

In `flag==VssFlag_Projection`, the error correction vector is computed as a function of the cur-
rent state values x and `P[i]` which the state correction vector is provided to the numerical solver. The
numerical solver, adds these correction values to the continous-time state vector elements, i.e., `X[i]`.

In `flag==VssFlag_OutputUpdate`, the block outputs are evaluated, i.e., the x and y in equation
10.6.
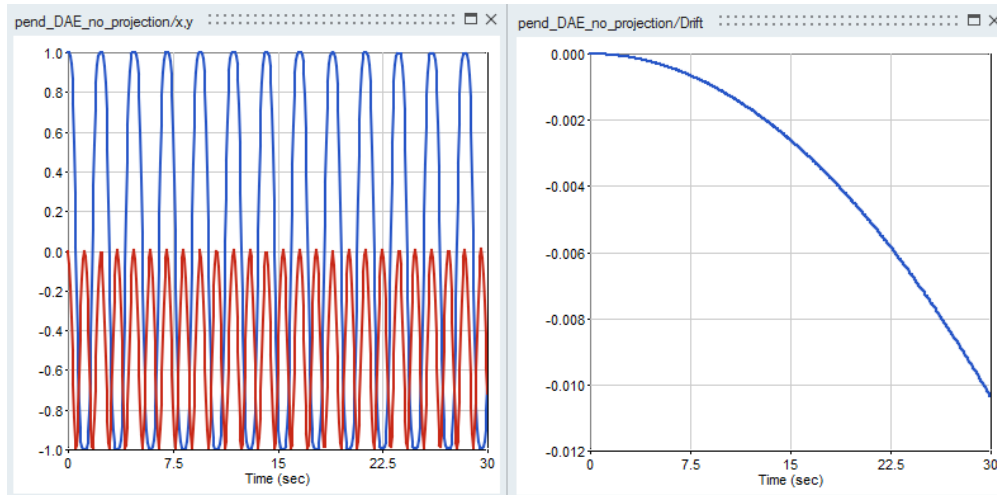
Figure 10.6: Simulation result of the Pendulum without constraints.
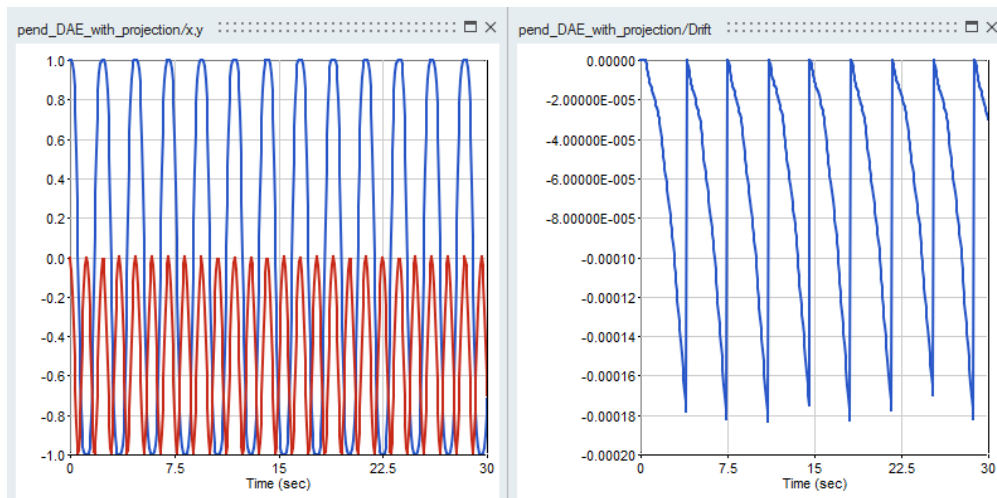
Figure 10.7: Simulation result of the Pendulum with constraints.

In the first test, the pendulum model is simulated ignoring the constraints, i.e., as a pure ODE without constraints. The RadauII-A solver with error tolerance=1e-4 is used. In Figure 10.6, the left plot displays the `x` and `y` variables and the right plot is the `drift` variable. `drift` is growing as time advances.

Then, the same pendulum model is simulated with additional constraints and projection APIs. As shown in Figure 10.7, the drift in the solution is kept below the requested error tolerance. Whenever the drift exceeds the error tolerance, the projection is applied and the drift becomes zero.

10.4 Numerical solver classifications and characteristics

Once a model is built in the **Twin Activate** block diagram editor, the solution of the model in a time domain must be computed numerically. There are several solvers available in **Twin Activate**. Each solver implements a specific numerical algorithm for solving the model. As a result, the choice of the solver becomes important: which solver to use and with what parameters. This section tries to explain

general characteristics of solvers that help the user to choose the right solver for right model.

10.4.1 Stiff vs. non-stiff ODE

An important characteristics of an ODE solver is to know if it is appropriate for solving a stiff ODE. Based on this information the appropriate solver can be chosen for a particular problem. In order to simulate an ODE, the numerical solver divides the simulation time into small intervals of length step-size and finds the solution at each interval. Smaller step-size generally results in higher accuracy of the solution, but it also means higher computation time. Even if the model to be simulated is stable, the stability of the solution provided by the numerical method depends directly on the step-size. In many cases smaller step-sizes should be used to ensure the stability of the numerical method. Whatever the numerical method, for large step sizes the solution tends to be unstable and stability may be achieved by using sufficiently small step sizes. Generally, the critical step-size depends on the eigenvalues of the dynamics of the model. In problems that have both large and small eigenvalues, the solution may have a short transient solution as the high frequencies settle, followed by a longer period where it is the result of the low frequencies that dominate. In this latter phase, the high frequencies are still present in the system even if they are not visible in the solution. Nevertheless, the integration method continues to use a small step-size to ensure the numerical stability. A system with such behavior is called stiff. Solution of stiff ODE's show sharp changes in the solution with respect to time scale of the overall solution.

10.4.2 Explicit vs. Implicit Solvers

Numerical solvers for differential equations are divided into two categories: Explicit and implicit methods. explicit methods calculate the new state of a system as a function of current state and current state derivatives. This is in contrast with implicit methods where the new state is found by solving a (very often) nonlinear equation involving both the new state and the new derivatives as a function of the current state. In other words, if $x(t)$ is the current system state and $x(t + h)$ is the unknown new state at the new time $(t + h)$, where h is the time step-size, then, for an explicit method we have

$$x(t + h) = F(x(t))$$

while for an implicit method a nonlinear equation like the following should be solved.

$$G(x(t), x(t + h)) = 0$$

Implicit solvers should solve a nonlinear equation at each time step which makes these solvers slower compared to explicit solvers. If these solvers are slower so why should they be used? They are actually used because many problems in real life are stiff. In simulation of a stiff model, explicit methods need to take small step sizes for a slowly varying solution. For such models, implicit methods, even though more costly, can integrate much faster by taking much larger step sizes. For stiff problem an explicit method requires very small time steps to keep the error small. The need for very small step-size makes the simulation very slow and costly. For such problems, to achieve given accuracy, implicit methods can take big steps and much less computational time event taking into account that they need to solve a nonlinear equation at each step. In summary the choice of the solver depends directly on the model, whether the model is stiff or not and what is the requirement for the simulation time.

10.4.3 Fixed-step vs. variable-step solvers

Another aspect of numerical solvers is whether they are fixed-step or variable-step. The numerical solver can use the same step-size all over the simulation or adjust it during the simulation. Based on

this property, the solver is fixed-step or variable step. In variable-step solvers the step-size is adjusted as a function of the estimated error in the solution. There are two kinds of error: absolute and relative error. Absolute error is the amount of physical error in a measurement. While relative error gives an indication of how good a measurement is relative to the size of the signal being measured. The acceptable error on a state value is defined as follows:

$$|x| * \text{Rtol}$$

where x is a state in the model. Since the x may be zero or near zero, the acceptable error is generally computed as follows:

$$|x| * \text{Rtol} + \text{Atol}$$

where Rtol and Atol are relative and absolute error tolerances respectively. The estimated error can be used to adjust the step-size to maintain the error under the acceptable error tolerances.

Variable step-size ODE solvers take small time steps when the variables change rapidly and take larger time steps when the variables vary slowly. This allows maintaining the estimated error under the error threshold. This is an important feature that increases the efficiency of the solvers. A variable-time step method is usually faster than its constant time step counterpart, because the computational effort is concentrated only on those time intervals that need it most, taking large steps over intervals that do not need small time steps. This feature usually compensates for the extra work required for error estimation and step-size adjustment. Variable step solvers are more reliable to handle fast changes in the solution by reducing the time step, where constant time step methods may give wrong results. On the other hand in some situations where the computational effort should be limited in each step, such as real-time applications fixed step-size solvers are privileged. The user will often want the solution at specified points in time, for example at equally spaced intervals, to produce tables or plots. For these cases, we do not need using fixed-step solvers. Fortunately, most variable solvers can provide the solution values at requested times by interpolating the computed solution points. This approach is more efficient than using fixed-step solvers.

10.4.4 Analytical vs. numerical Jacobian matrix

In solving the ODE,
$$\dot{x} = f(x)$$

implicit solvers need the matrix
$$J = \frac{\partial f}{\partial x}$$

and for solving the DAE
$$0 = F(\dot{x}, x)$$

implicit solvers need the matrix

$$J = \alpha \frac{\partial F}{\partial x} + \beta \frac{\partial F}{\partial \dot{x}}$$

where J is called the Jacobian matrix. The Jacobian matrix can either be provided as an analytical matrix by the user or can be computed numerically by the solver via the finite difference method. The numerical method works fine in most cases, but if the model is stiff or complex, using a numerically computed Jacobian matrix may cause the solver to fail. In **Twin Activate**, if a block provides its local

Jacobian matrix analytically, the simulator can take advantage of it and build the overall Jacobian matrix and deliver it to the solver.

A **Twin Activate** block can provide analytical Jacobian of its internal model, i.e., the matrix $J(x)$. If a block does not provide this matrix analytically, the simulator computes it numerically via finite difference method. The mixture of analytical and numerical Jacobian matrices of blocks are used to compute the global Jacobian of the whole model. Currently only few blocks provide the analytical Jacobian information, such as FMU and linear ABCD block.

10.4.5 Single-step vs. multi-step solvers

Single-step solvers approximate the solution of the model at time $t + h$ as a function of the solution of the model between times t and $t + h$ (current step). On the other hand, multi-step solvers approximate the solution at the end of the time step as a function of the model solution at previous steps. In order to achieve high order accuracy, single-step solvers may need to evaluate the differential equation more often per time step than multi-step solvers. As a result, single step solvers may have higher computational cost per step than multi-step solvers. Due to this cost, multi-step solvers may be more efficient that single-step solvers. The downside of multi-step solvers is the requirement for the initialization phase. These solvers should start with lower order methods or with other single-step methods for very first steps. This may affect the accuracy. Furthermore, multi-stage methods are less efficient if the solution contains frequent discontinuities with respect to single-stage solvers. Indeed, at discontinuities, all solution history of multi-step solvers is obsolete and should be thrown away and the solver needs another costly initialization. A single-step solver may be more efficient for applications with high number of discontinuities. In **Twin Activate** `Euler`, `Radau` and `Runge-Kutta` are examples of single-step solver, and `Lsoda`, `Dopri`, `Ida` are examples of multi-step solver.

10.4.6 ODE vs. DAE solver

A **Twin Activate** model is either represented as ODE or DAE. Simulation of a DAE model needs special solver. **Twin Activate** provides `IDA`, `Daskr`, and `Radau` for simulation of DAE models. Although these solvers can be used for simulation of ODEs, they are slower than ODE solvers such as `Lsoda` for the simulation of an ODE. DAE solvers are however good for simulation of stiff ODEs.

Some kind of DAEs can be represented as ODEs with algebraic constraints, i.e.,

$$\begin{cases} \dot{x} & = f(x) \\ 0 & = g(z) \end{cases}$$

where x is the state vector and z is a subset of vector x. This model can be simulated using an ordinary ODE solver without taking into account the constraint $g(z)$. The solution of course does not satisfy the constraint and that may leads to results in a solution which does not respect physical laws such as energy or mass conservation laws. In order to simulate this kind of models, **Twin Activate** provides the `CPODE` solver (12). In this case the user should provide the constraints or a projection that brings back the solution on the constraints.

When an ODE is simulated, any initial value can be used as start values of states. DAE's, on the other hand, do not have this freedom. In a DAE, initial value of states and their derivatives should be consistent, i.e., they should satisfy a set of initial algebraic equations. In order to simulate a DAE these algebraic equations should be solved at the beginning of the simulation as well as at any discontinuity instants. In order to solve these initial equations an algebraic solver should be used. `IDA` and `DASKR`

has their own internal algebraic solvers, but `Radau` needs an external one. **Twin Activate** provides `IDACALC`, `DASKRCALC` and `FSOLVE` algebraic solvers that can be used with any of DAEs solvers.

10.5 Numerical Solver interface with the simulator

Any numerical solver has several parameters and a correct and fast simulation requires the fine tuning of these parameters and a good interface with the simulator. In this section we explain some important common parameters of solvers and the way they are handled by the simulator of **Twin Activate**.

10.5.1 Forward looking in time (stopping time)

Another important feature of most solvers is the possibility of providing a dense output result. Consider a situation where the user needs to integrate a model from T_{initial} to T_{final} and many intermediate output points are required. It is not efficient to run the solver from one output point to the next and then return to the main program to print the results at the output points, even if the solver is called with a hot-start option. To handle this situation efficiently, the solver may integrate past the output points and interpolate back to obtain the results at requested output points. This increases the efficiency, but sometimes it is not possible to integrate beyond some point t_{stop} because the equation changes there or simply because the solution or its derivative is not defined past t_{stop}. To overcome this obstacle, most solvers have another important property to set a time limit on the solver. Time limit can be imposed to forbid the solver from advancing the time beyond a given time called the stopping time. In **Twin Activate** during the simulation, the nearest event that requires a cold-restart is considered as the stop time and is given to the solver.

10.5.2 Zero-crossing and event-detection

An important feature of a hybrid simulator is the ability to detect and handle zero-crossings or state-events. Zero-crossings are introduced in models to correctly handle discontinuities or generate events. The ability to detect the time when a discontinuity occurs, or more precisely, the time when a function crosses some given value (by default considered zero) is of capital importance in the hybrid environment. When a zero-crossing occurs, the simulator pinpoints the exact crossing time and stops the simulation at the crossing point. Zero-crossings are not just used to generate events in order to communicate with the discrete-time part of the model. In most cases zero-crossings are used to properly control the solver to simulate non-smooth continuous-time dynamics. Consider a continuous-time model in which the absolute value function is used. For example, the system

$$\begin{cases} \dot{x} & = |u| \\ u & = \sin(t) \end{cases}$$

can produce a non-smooth signal even if the input $u = sin(t)$ is smooth. It is important to halt the solver and do a cold restart at the point where the non-smoothness occurs.

The simulator triggers a zero-crossing event if the sign of a zero-crossing function changes or the function becomes zero at the end of a solver's step. This works fine to some extent. In fact in some particular cases, where for example there is a model change, or having loose error tolerances, the simulation may show chattering and detects numerous unphysical zero-crossings. In **Twin Activate** simulator the user can set a threshold value to the zero-crossing function. If this threshold is defined, after detection of a zero-crossing event, that zero-crossing is disabled (masked) until its absolute value

becomes larger than the threshold value and leaves the threshold region (unmasked). This mechanism avoid unwanted chattering. The default value of the threshold is zero.

10.5.3 Cold-start versus hot-start of the solver

There are two strategies for starting the integration with the multi-step solvers like `IDA` or `Lsoda` solvers: cold-start and hot-start. A cold-start is the first call to the solver for a given problem. In this case, many cumbersome memory setting tasks are done and the integration is started with a very small step-size and a first order method. A hot-start, on the other hand, is a call to the solver that is not the first call for the given problem. In this case, assuming no change in the problem or discontinuity, the solver does not perform the memory setting tasks and can use all past information about the solution. This makes a hot-start more efficient than a cold start. Cold restart is costlier for multi-step solvers with respect to single step ones. Since in multi-step solvers memory about solution in previous step is lost and the solver needs a new reinitialization and restart from lower order methods. In **Twin Activate** the simulator has been designed to avoid unnecessary costly cold-restarts.

In order to demonstrate the effect of a bad design and excessive cold restarts, consider the model illustrated in Fig.10.8. In this model, instead of using a clock to generate an event sequence, a Sine block is connected to a zero-crossing block. At each zero-crossing detection, the numerical solver is cold restart-ed. The model contains continuous-time states, and detection of zero-crossings forces the multi-step solver (LSODA in this example) to do a cold-restart. The simulation time becomes much shorter, if a Clock block is used to activate the Counter block instead of the Zero-crossing block.
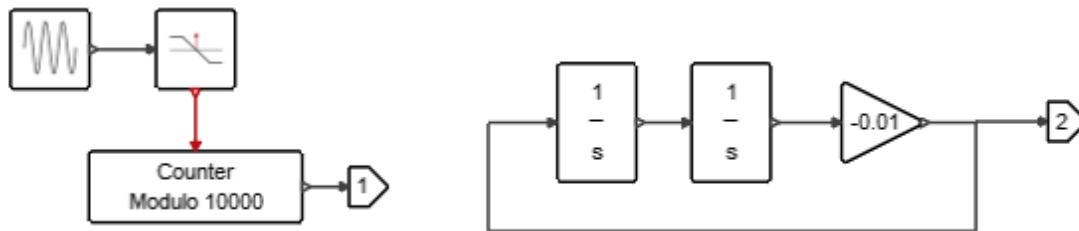

Figure 10.8: Effect of excessive cold-restart (coldrestart.scm)

10.5.4 Absolute and relative error tolerances

The absolute and relative error tolerances specified for the integration of an ODE affects the speed and accuracy of the computation for variable step solvers. Small error tolerances may result in higher accuracy, but the simulation time will become large. On the other hand looser error tolerance values, speeds up the simulation but the error introduced in the solution will be higher. As a consequence, it is important to find a compromise between speed and accuracy. It is not recommended to choose very tight (small) or very loose (large) Atol and Rtol values. If Very small error tolerances are chosen the integration may become too slow or even fail to achieve the desired accuracy. In general a relative error tolerance around 1% or 0.1% is acceptable. The value of absolute error tolerance is only important if the value of signal being observed is close to zero.

10.5.5 Maximum, minimum, and initial step-size

Variable step solvers adjust the step-size during the integration, if the model becomes stiff or the error becomes large, the solver reduces the step-size to reduce the error. If the error estimation remains

constant or reduces, the solver tries a larger step-size to advance faster in time. Thus, the step-size value can be very small or very large. **Twin Activate** provides the possibility of setting high and low limits on the step-size chosen by the solver. As an example to demonstrate how setting a high limit on the step-size may be useful, consider a simple non-stiff model containing a zero-crossing. The model is not stiff, thus the solver tries to advance the time by taking large step sizes. If it happens that the zero-crossing function changes the sign several times during a large step, the solver will miss the sign change in the zero-crossing function and it may introduce error in the result. In order to avoid such kind of errors, the user can limit the maximum step-size. If the maximum step-size set to *auto* the solver chooses according to the following formula.

$$H_{max} = \frac{T_{\text{final}} - T_{\text{initial}}}{100}$$

Where T_{final} is the final time and T_{initial} is the initial simulation time.

The size of the very first steps after a cold-restart in variable step solvers does not usually have much importance, since the solver can adjust it and find an appropriate value. In some situations, the solver is unable to find a good value for the step-size and fails. In order to help the solver, the user can set the initial value of the step-size.

10.5.6   Fixed-step solver interface with the simulator

Fixed-step solvers are usually used for non-stiff models where convergence is ensured by choosing small enough step-sizes. The way fixed step solvers have been interfaced with the simulator is a bit different from that of variable step solvers. First important issue is handling events when a fixed-step solver is chosen. With fixed-step solvers, the simulator considers the step-size as the basic clock of the model. All other events are synchronized with solver steps. No event is allowed to be triggered during a step. If an event is programmed to happen during a solver step, it is postponed to the end of the step. It is also important to indicate that when a fixed-step solver is chosen, no zero-cross iteration is performed. The sign of zero-crossing functions are checked only at the beginning of the step. An if a sign-change happens, an event is triggered instantly. `mode` variables are also completely ignored.

10.6   Numerical solvers available in Twin Activate

Twin Activate provides several solvers for simulation of ODE and DAE models. Here is the list of available solvers. If a model is represented by an ODE, an ODE solver can be used. On the other hand if the model is represented by a DAE, a DAE solver should be used. A model using Modelica blocks, implicit blocks or `Automaton` block usually results in DAE. Note that for simulation of ODE models, DAE solvers can also be used but they are less efficient that ODE solvers.

ODE solvers

1. **Lsoda** is a variable step and variable order solver [1, 2, 3, 5]. It is a multi-step solver based on BDF method. This solver is the best choice when the user does not know if the model is stiff or not. The solver switches automatically between stiff and non-stiff methods during the integration. it automatically selects between nonstiff (Adams) and stiff (BDF) methods. It uses the stiff method initially, and dynamically monitors data in order to decide which method to use. The maximum step-size for this solver is unlimited. The maximum integration order for stiff and non-stiff methods

is set to 5 and 12 respectively. `Lsoda` is a part of `ODEPACK`[5]. `ODEPACK`[4] is a collection of Fortran solvers for the initial value problem for ordinary differential equation systems. It consists of nine solvers, namely a basic solver called LSODE and eight variants of it `LSODES,` `LSODA,` `LSODAR,` `LSODPK,` `LSODKR,` `LSODI,` `LSOIBT,` and `LSODIS`. The collection is suitable for both stiff and nonstiff systems. It includes solvers for systems given in explicit form, $\frac{dy}{dt} = f(t, y)$, and also solvers for systems given in linearly implicit form,

$$A(t, y)\frac{dy}{dt} = g(t, y)$$

In **Twin Activate**, `Lsoda` has been used and modified to support multi-threaded usage (usefull in FMU export).

2. **CVODE** which is a part of the SUNDIALS package [6] [6] is a solver for stiff and nonstiff ordinary differential equation (ODE) systems (initial value problem) given in explicit form $y' = f(t, y)$. The methods used in `CVODE` [8] are variable-order, variable-step multistep methods. For nonstiff problems, `CVODE` includes the Adams-Moulton formulas, with the order varying between 1 and 12. For stiff problems, `CVODE` includes the Backward Differentiation Formulas (BDFs) in so-called fixed-leading coefficient form, with order varying between 1 and 5. For either choice of formula, the resulting nonlinear system is solved (approximately) at each integration step. For this, `CVODE` offers the choice of either functional iteration, suitable only for nonstiff systems, and various versions of Newton iteration. In the cases of a direct linear solver (dense or banded), the Newton iteration is a Modified Newton iteration, in that the Jacobian is fixed (and usually out of date). The direct solvers can only be used with the provided serial vector structure. These are written in C, but assume that the user calling program and all user-supplied routines are in Fortran. In **Twin Activate**, the `CVODE` solver is provided in four variants:

   - **CVODE-BDF-NEWTON** is a variable step and variable order solver. The multi-step method is based on BDF with maximum order of 5. The Newton iterations are used to solve the nonlinear equation resulting from discretization. This solver is used when we know the model stays always in the stiff region.

   - **CVODE-BDF-Functional** is a variable step and variable order solver. The multi-step method is based on BDF with maximum order of 5. A functional iteration (some sort of fixed-point iteration) is used to solve the nonlinear equation resulting from discretization.

   - **CVODE-ADAMS-NEWTON** is a variable step and variable order solver. The multi-step method is based on Adams method with maximum order of 12. The Newton iterations are used to solve the nonlinear equation resulting from discretization.

   - **CVODE-ADAMS-Functional** is a variable step and variable order solver. The multi-step method is based on Adams method with maximum order of 12. A functional iteration is used to solve the nonlinear equation resulting from discretization. This solver which is usually used when the model is known to be non-stiff, allows to get a high accuracy and fast integration.

3. **RADAU-IIA** [9, 10, 11] is variable step and variable order solver. It is a single-step method based on implicit Runge-Kutta methods (Radau IIa) of order 5, 9, 13. These methods are L-stable. These methods are suitable for $Mf' = f(t, y)$ with a possibly singular matrix M. this property makes this solver suitable for stiff ODEs and index-1 and index-2 DAEs. A Newton iteration is

---

[5]Available at https://www.netlib.org
[6]Available at https://computation.llnl.gov/casc/sundials/main.html

used to solve the nonlinear equation resulting from discretization. The RADAU solver[7]. The higher order methods perform better than low order methods as soon as the convergence of the Newton iteration is sufficiently fast. This solver is used when the model is stiff and since it is a single-step solver it is more efficient in presence of frequent discontinuities. The switching between ODE and DAE modes are done automatically by the **Twin Activate** solver. In other words, if the model is explicit (ODE), the ODE mode of RADAU is used. On the other hand, if the model is implicit (DAE), the DAE mode of the RADAU solver is used.

4. **Backward Euler** is an order one method with fixed step size. The backward Euler method (or implicit Euler method) is one of the most basic numerical methods for the solution of ordinary differential equations. This solver is widely used in industry as the simplest fixed-step solver for stiff models. It is similar to the forward Euler method, but differs in that it is an implicit method. The backward Euler method has order one and is A-stable.

5. **Implicit Trapezoidal** is second order, fixed step and single-step solver. It is an implicit method from family of Runge–Kutta method.

6. **DOPRI (Dormand-prince)** is variable step, 5th order and single-step solver. It is based on a family of Runge-Kutta order 5 method. The Dopri solver [8] provides a fast integration for non-stiff ODE. the Dormand–Prince method, is an explicit method for solving ordinary differential equations [12, 9, 10]. This solver uses six function evaluations to calculate fourth- and fifth-order accurate solutions. The difference between these solutions is then taken to be the error of the (fourth-order) solution. This error estimate is very convenient for adaptive stepsize integration algorithms.

7. **Runge-Kutta** is a fixed-step, 5th order explicit Runge-Kutta method solver. It is based on a family of Runge-Kutta order 5 method. This solver is used for non-stiff models.

8. **Classical Runge-Kutta** is a fixed-step, 4th order Runge-Kutta method solver.

9. **Forward Euler** is a first order, fixed step, single-step solver. This solver is widely used in industry as the simplest fixed-step solver. the Euler method is the simplest numerical method for ODEs with a given initial value. The Euler method is a first-order method, which means that the local error (error per step) is proportional to the square of the step size, and the global error (error at a given time) is proportional to the step size. The Euler method often serves as the basis to construct more complex methods, e.g. Predictor-corrector method. This method is widely used in industry when the computation time is critical such as in embedded systems.

10. **Explicit Trapezoidal** is a second order, fixed step, single-step solver. It is an explicit method from family of Runge-Kutta method.

11. **CPode(ODE with constraint)** is a variable step and variable order solver. It is also a multi-step method is based on BDF with maximum order of 5. The Newton iterations are used to solve the nonlinear equation resulting from discretization. This solver is especially used when the model is an ODE with several additional constraints.

Cpode is a numerical integrator for solving ODE problems using coordinate projection. It is based on the CVODES integrator which is part of the DOE Sundials[9] suite. CPODES is a multi-step integrator providing variable order Adams (up to 12th order) and BDF (up to 5th order) methods for non-stiff problems and BDF (up to 5th order) for stiff problems. It uses CVODES to advance

---

[7]Available at http://www.unige.ch/~hairer/
[8]Available at http://www.unige.ch/~hairer/
[9]https://computation.llnl.gov/projects/sundials

the ODE (10.1a), and then performs coordinate projection back to the constraint manifold (10.1b) to exactly solve the DAE (10.1). The projection is also incorporated back into the error test where it permits larger steps.

In order that the constraints be respected during the simulation, the blocks should check the constraints on each step taken by the solver and propose a correction to the states. The constraint corrections are then applied to the continuous-time states of the model. This type of constraint correction is called projection in **Twin Activate** (see 10.3.3). The **CPode(ODE with constraint)** solver supports projection type constraint correction (see 10.3.3).

12. **CPode(ODE with algebraic constraint)** is the same solver as **CPode(ODE with constraint)**, but it supports the second type of constraint correction, i.e., algebraic constraints (see 10.3.2). In this constraint correction type, the model provides directly the constraint equations and it is up to the solver (**CPode(ODE with algebraic constraint)**) to respect the algebraic constraints along with the original ODE. The **CPode(ODE with algebraic constraint)** solver is the only solver in **Twin Activate** that handles algebraic constraints (see 10.3.2).

13. **Cash-Karp** is a variable step, 5th order and single-step solver. This solver is a member of Runge-Kutta family methods of order 5. The Cash-Karp solver[14, 15] provides a fast integration for non-stiff ODE or semi-stiff ODE. the Cash-Karp method is an explicit method for solving ordinary differential equations. This solver uses six function evaluations to calculate fourth- and fifth-order accurate solutions. The difference between these solutions is then taken to be the error of the (fourth-order) solution. This error estimate is very convenient for adaptive stepsize integration algorithms. This solver supports constraint correction of type projection (see 10.3.3).

14. **SDIRK** [9, 10, 11] is variable step and variable order solver. It is a single-step method based on implicit Runge-Kutta methods (SDIRK) of order 5. These methods are suitable for $Mf' = f(t, y)$ with a possibly singular matrix M. this property makes this solver suitable for stiff ODEs and index-1 and index-2 DAEs. A Newton iteration is used to solve the nonlinear equation resulting from discretization. The SDIRK solver[10]. This solver is used when the model is stiff and since it is a single-step solver it is more efficient in presence of frequent discontinuities. The switching between ODE and DAE modes are done automatically by the **Twin Activate** solver. In other words, if the model is explicit (ODE), the ODE mode of SDIRK is used. On the other hand, if the model is implicit (DAE), the DAE mode of the SDIRK solver is used.

DAE solvers

1. **IDA** which is a part of the SUNDIALS package [6] [11] is a numerical solver for the solution of differential-algebraic equation (DAE) systems in the form $F(t, y, y') = 0$. It is written in C, but derived from the package DASPK which is written in Fortran. The integration method in IDA [7] is variable-order, variable-coefficient BDF, in fixed-leading-coefficient form. The method order varying between 1 and 5. The solution of the resulting nonlinear system is accomplished with some form of Newton iteration. In the cases of a direct linear solver (dense or banded), the nonlinear iteration is a Modified Newton iteration, in that the Jacobian is fixed. When the model is DAE this solver can be used. The minimum step-size is zero for this solver.

2. **DASKR** is a variable step, variable order method solver for systems of differential-algebraic equations (DAEs). DASKR[12] [13] is a multi-step solver based on BDF method. In contrast to the DASSL

---

[10]Available at http://www.unige.ch/~hairer/

[11]Available at https://computation.llnl.gov/casc/sundials/main.html

[12]Available at http://www.netlib.org/ode/

solver, DASKR includes a procedure for calculating consistent initial conditions for a large class of problems (which includes semi-explicit index-1 systems). The package also includes an option to omit the algebraic components from the local error control. If the model is DAE this solver can be used. The minimum step-size is zero for this solver.

The following table classifies the solver to clarify which solver is appropriate for stiff or non-stiff ODE or for DAE models.

Solver type	Stiffness	Solver Name

Fixed step-size	non stiff ODE	`Forward Euler`
		`Explicit Trapezoidal`
		`Classical Runge-Kutta`
		`Runge-Kutta`

	stiff ODE	`Backward Euler` `Implicit Trapezoidal`

variable step-size	(semi)-stiff ODE	`CVODE-BDF-Functional`
		`CVODE-ADAMS-Functional`
		`DOPRI (Dormand-prince)`
		`CASH-KARP`

	stiff ODE	`Lsoda`
		`CVODE-BDF-NEWTON`
		`CVODE-ADAMS-NEWTON`
		`RADAU-IIA`
		`SDIRK`
		`CPODE`

	DAE	`IDA`
		`RADAU-IIA`
		`SDIRK`
		`DASKR`

Bibliography

[1] Alan C. Hindmarsh, Lsode and Lsodi, "Two new initial value ordinary differential equation solvers", acm-signum newsletter, vol. 15, no. 4 (1980), pp. 10-11.

[2] Linda R. Petzold, "Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations", Siam J. Sci. Stat. Comput. 4 (1983), pp. 136-148.

[3] Kathie L. Hiebert and Lawrence f. Shampine, "Implicitly defined output points for solutions of ODEs", sandia report sand80-0180, February, 1980.

[4] A. C. Hindmarsh, "ODEPACK, A Systematized Collection of ODE Solvers," in Scientific Computing, R. S. Stepleman et al. (eds.), North-Holland, Amsterdam, 1983 (vol. 1 of IMACS Transactions on Scientific Computation), pp. 55-64.

[5] K. Radhakrishnan and A. C. Hindmarsh, "Description and Use of LSODE, the Livermore Solver for Ordinary Differential Equations," LLNL report UCRL-ID-113855, December 1993.

[6] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, "SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers," ACM Transactions on Mathematical Software, 31(3), pp. 363-396, 2005. Also available as LLNL technical report UCRL-JP-200037.

[7] A. C. Hindmarsh, "The PVODE and IDA Algorithms," LLNL technical report UCRL-ID-141558, December 2000.

[8] S. D. Cohen and A. C. Hindmarsh, "CVODE, A Stiff/Nonstiff ODE Solver in C," Computers in Physics, 10(2), 1996, pp. 138-143. Also available as LLNL technical report UCRL-JC-121014 Rev. 1, August 1995.

[9] Ernst Hairer, G. Wanner, S.P. Norsett , "Solving Ordinary Differential Equations I: Nonstiff Problems", Springer Series in Computational Mathematics, 1996.

[10] Ernst Hairer, Gerhard Wanner, "Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems", Springer Series in Computational Mathematics, vol 2, 1993

[11] Ernst Hairer, Gerhard Wanner, "Stiff differential equations solverd by Radau methods", J. comput. Appl. Math., 11:93-111, 1999.

[12] Dormand, J. R.; Prince, P. J. , "A family of embedded Runge-Kutta formulae", Journal of Computational and Applied Mathematics 6: 19–26

[13] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold, "Consistent Initial Condition Calculation for Differential-Algebraic Systems", SIAM J. Sci. Comp. 19 (1998), pp. 1495-1512.

[14] J. R. Cash and A. H. Karp, "A Variable Order Runge-Kutta Method for Initial Value Problems with

rapidly Varying Right-Hand Sides", ACM Transaction on Mathematical Software, Vol. 16, No 3, page 201-222, (1990).

[15] William H. Press and Saul A. Teukolsky, "Adaptive Stepsize Runge-Kutta Integration", Numerical recipes, Computers in Physics 6, 188 (1992).

[16] U. Ascher and L. Petzold. "Computer methods for ordinary differential equations and differential-algebraic equations", SIAM, 1988.

[17] U. M. Ascher and L. R. Petzold, "Projected implicit runge-kutta methods for differential-algebraic equations", SIAM, Numerical Analysis, 28(4), 1097-1120., 1992.

[18] U. M. Ascher, H. Chin, and S. Reich."Stabilization of daes and invariant manifolds", Numer. Math. 67: 131, 1994.

[19] J. Baumgarte, "Stabilization of constraints and integrals of motion in dynamical systems", Computer Methods in Applied Mechanics and Engineering Volume 1, Issue 1, Pages 1-16, 1972.

[20] E. Eich, "Convergence results for a coordinate projection method applied to mechanical systems with algebraic constraints", SIAM J. on Numerical Analysis 30(5):1467-1482, 1993.

[21] E. Eich-Soellner and C. Fuhrer, "Numerical Methods in Multibody Dynamics", European Consortium for Mathematics in Industry, B.G. Teubner, 1998.

[22] C.W. Gear, "Maintaining solution invariants in the numerical solution of odes", Journal on Scientific and Statistical Computing, Vol. 7, No.3, 1986.

[23] E. Hairer and G. Wanner, "Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems", Springer, 1996.

[24] S.E. Mattsson and G. Soderlind, "Index reduction in differential-algebraic equations using dummy derivatives", SIAM Journal of Scientific Computing. 14(3), pp. 677-692, 1993.

[25] R. Nikoukhah, "A simulation environment for efficiently mixing signal blocks and modelica components", 12'th International Modelica conference, 2017.

[26] M. Otter and H. Elmqvist, "Transformation of differential algebraic array equations to index one form", Proceedings of the 12th International Modelica Conference, Prag, Czech Republic, 2017.

[27] C. C. Pantelides, "The consistent initialization of differential-algebraic systems", SIAM Journal on Scientific and Statistical Computing, 213–231, 1988.

[28] H. Olsson S.E. Mattsson and H. Elmqvist, "Dynamic selection of states in dymola", Modelica Workshop 2000, Lund, Sweden, pp. 61-67, 2000.

[29] L. Shampine, "Conservation laws and the numerical solution of odes", Comput. Math. Appls, Part B., 12, pp. 1287-1296, 1986.

Chapter 11

Physical component modeling in Twin Activate

11.1   Introduction

Twin Activate blocks communicate data with other blocks through ports that are specified as inputs and outputs. Produced data is placed on a block output port and retrieved through the input port of one or more other blocks. Even though it is possible to model many physical systems containing components in **Twin Activate**, this generally requires translating the implicit equations representing the behavior of the physical components into explicit form and as a result, the resulting **Twin Activate** diagrams may differ significantly from the original component based diagrams.

In **Twin Activate**, the behavior of physical components is specified through symbolic equations. Unlike regular blocks that may be regarded as black boxes, physical components expose the implicit equations of the corresponding component. **Twin Activate** then can, through symbolic manipulation, use these equations to produced efficient code for models obtained from the interconnection of physical components. The symbolic specification of the behavior of physical components in **Twin Activate** is done using the Modelica language.

11.2   Causal vs. acausal modeling

Twin Activate allows using components for construction of models. The ports of a component are not labeled, a priori, as inputs and outputs, and the block imposes a dynamical constraint on the values on its ports, contrary to regular blocks which compute explicitly their outputs as functions of their inputs. Components are essential for constructing models which include physical components such as resistors, capacitors, etc., in electricity, or pipes, nozzles, etc., in hydraulics. They also arise in many other areas such as mechanics and thermodynamics.

Modular model construction can be classified in two major categories: causal and acausal. To make an analogy with computer programming languages, causal modeling corresponds to the use of assignment statements where the right-hand sides of the equations are evaluated and the result of the evaluation is assigned to the variables on the left-hand side of the equations. This corresponds closely to the evaluation of the outputs based on the inputs in a causal module. Acausal modeling on the other hand is closer to *equations* in the mathematical sense of the term. For an equation in a set of mathematical

equations, it is not possible to classify (at least a-priori) its variables as inputs and outputs. In this case, for constructing the solution to the set of equations, the choice of the variable which should be computed in terms of the rest of the variables, in one particular equation, depends on the complete set of equations. In other words, a mathematical equation contains potentially a number of assignment statements; the choice of the statement which should be used for constructing the solution of the complete problem depends on all the equations.

An acausal module is like a mathematical equation. It has ports of communication with other modules but these ports are not labeled a-priori as inputs and outputs.

Most physical components are more naturally modeled as acausal modules simply because physical laws are expressed in terms of mathematical equations. Consider a resistor in an electrical circuit. If the resistor is to be used as a module, it can only be an acausal module because depending on the way the resistor is used, the input can be a current and the output a voltage, or the input a voltage and the output a current.

Often in modeling and simulation softwares, only causal modules (blocks) are considered. This situation is not only easier to implement but it provides a stronger notion of modularity which corresponds closely to the notion *subsystem* used by systems and control engineers. This is the reason why such way of modeling is referred to as *system level modeling*, as opposed to *component level modeling* where acausal modules (blocks) are used.

It is in general possible to convert a component level model into a system level model by rewriting the equations and finding the appropriate causality structure in each module, but this task is time consuming, and error prone. Furthermore the resulting (system level) model has very little resemblance with the original model making subsequent modifications difficult to realize. Indeed, a small change in the original model may require a complete redesign of the system level model.

To illustrate the difference between component level and system level modeling, consider the simple electrical circuit shown in Figure 11.1.
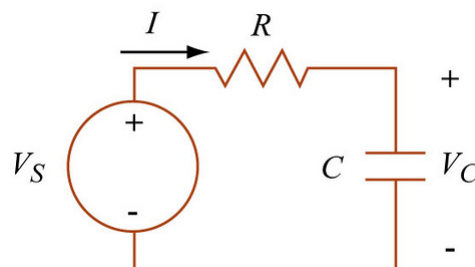

Figure 11.1: The original electrical system to be modeled.

The circuit in Figure 11.1 clearly corresponds to a component level model. This is what an electrical engineer would like to work with. Using elementary models for each electrical component, we obtain a corresponding equivalent system level model depicted in Figure 11.2. We see clearly an important drawback in using such a model: there is no one to one correspondence between original circuit elements and the blocks in the diagram. It is not difficult to see that for example replacing the resistor with a diode, would require starting over the construction of the system level model. This is an important point because even for this very simple example, the analysis to convert the intuitive physical model to a system level model is non-trivial and cumbersome.

It turns out however that in many situations, in particular in systems and control applications, the use of
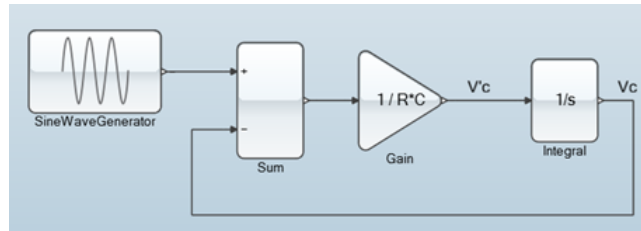
Figure 11.2: The system-level (causal) implementation of model depicted in figure 11.1

causal modules is very useful. As an example in figure 11.3, a control system has been illustrated. This system is composed of several parts such as the plant, feedback, and the controller. In such control systems it is much easier for the control engineer to have a system level model to use for the design of the controller and to optimize the control parameters.
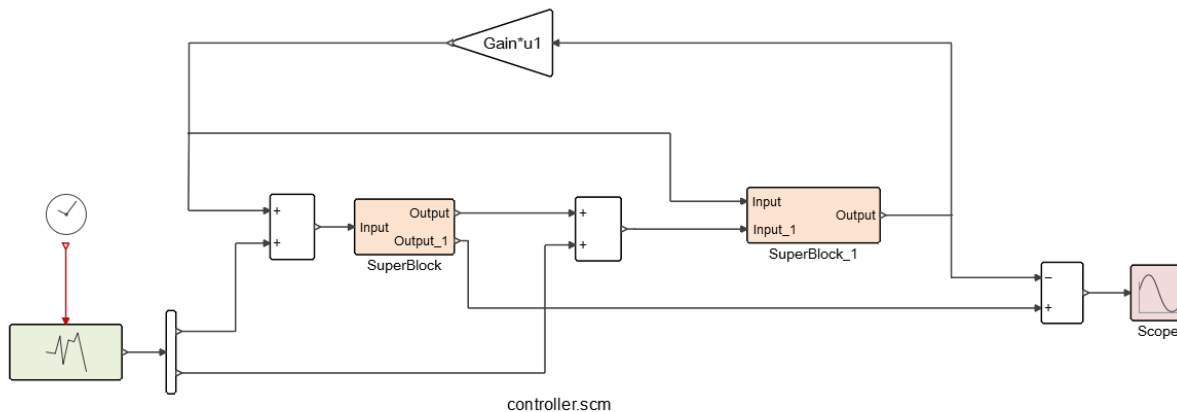

Figure 11.3: A typical system level modeling and design model

Even though causal models can in some sense be considered as special cases of acausal models, the restrictions imposed by causality implies that the behavior of the module can be implemented using an almost block box program which evaluates its outputs as a function of its inputs. The implementation of the behavior of the module in the acausal case is more complicated and requires either the construction of multiple causal modules or a formal description. In the latter case, the construction of the complete model requires formal manipulation of equations obtained from the description of acausal modules and their connection topology. It is for this reason that in **Twin Activate** we allow the co-existence of both type of modules: causal and acausal.

11.3   Modelica: a standard in component level modeling

In order to extend the capacity of **Twin Activate** to allow component level modeling, we needed a language for describing the algebraic-differential constraints imposed on the input/outputs of the acausal blocks. We found Modelica [1] to be an excellent choice.

Modelica is primarily a modeling language, sometimes called hardware description language, for specifying mathematical models of physical systems, in particular for the purpose of computer simulation. Modelica is a modern object-oriented programming language based on equations instead of assign-

ment statements (it can thus be classified as a declarative language)[3]. This makes it particularly useful for the purpose of acausal modeling, and this is not a surprise because Modelica is developed exactly for such applications. As an example the electrical circuit in figure 11.1 can be expressed in modelica as follows:

```
class RC_circuit
  Resistor   Res(R=1);
  Capacitor  Cap(C=1, v(start=0));
  Ground     Gnd;
  VsourceAC  VS(VA=1, f=50);
equation
  connect (VS.n,GND.p);
  connect (VS.p,Res.n);
  connect (Res.p,Cap.p);
  connect (cap.n,Gnd.p);
end RC_circuit;
```

where each component has been defined in the Modelica's library. For example here is the model of a simple Inductor.

```
class Capacitor "Ideal capacitor"
      Pin p, n;
      Real v;
      Real i;
      parameter Real C "Capacitance";
equation
      i = C*der(v);
      v = p.v - n.v;
      0 = p.i + n.i;
      i = p.i;
end Capacitor;
```

We are not going to present Modelica here in this document. More information can be found on Modelica website and in [1].

Modelica language is very rich and has the capacity also to handle discrete time systems. Our use of Modelica however, at least for the moment, is limited to the continuous-time part. We think that, except for very special cases, modeling discrete events can be done more naturally at the system level, where **Twin Activate** provided already the necessary functionalities for modeling and simulation. We thus have found that the best compromise is to allow the usage of acausal blocks in the existing **Twin Activate** environment while keeping the traditional causal blocks the way they were. Both types of blocks of course should be usable in the construction of diagrams.

Since the information available on the links connecting acausal models corresponds in general to physical quantities such as (voltage, current), (flow,pressure), etc, and are very different from information on links connecting causal models which transport signals, even a simple derivation placed on a link has a very different significance in two cases: a link that splits in two in the causal setting simply means that the information is duplicated whereas in the acausal setting, say in an electrical circuit, such a split must be implement using the Kirchof's voltage and current laws. Clearly it would be meaningless to allow the direct connection of a port of an acausal model to that of a causal model. That is why, special blocks were introduced in **Twin Activate** to be used as interface between the causal and the acausal

world. Such blocks which have very clear physical significance, may have input ports, output ports *and* "acausal ports". An example of such a block is a voltmeter. This block has two acausal ports; physically they correspond to the place where the two wires are to be connected. But it also has an output where the measured value of the voltage is sent out of the block. In general, sensors and actuators make the interface between the component level part of the model and system level part.

11.3.1   Example: modeling a DC motor

Consider a DC motor as shown in figure.11.4. The DC motor can be modeled using some basic electrical and mechanical components, such as resistor, inductor, and rotational inertia, as shown in Figure.11.5.
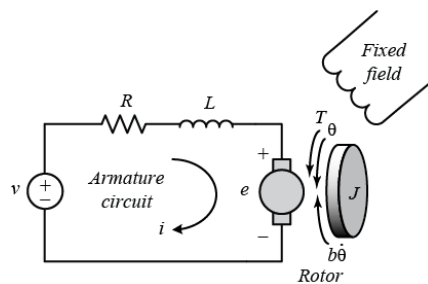
Figure 11.4: Representation of a DC motor.
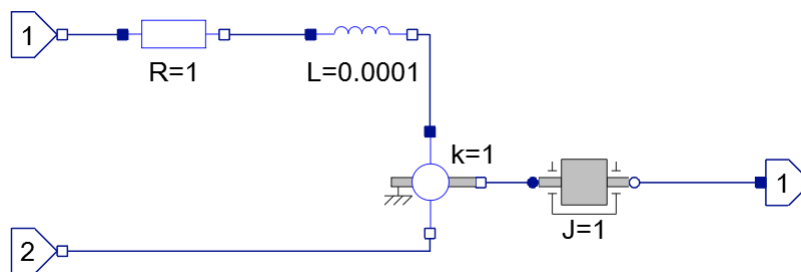
Figure 11.5: The model of a DC motor in **Twin Activate**.

Figure 11.6: Representation of the DC motor component in **Twin Activate**.

From electrical engineering courses you may remember that a DC motor can be either used as ordinary motor or as DC voltage generator. Since the ports of the components are not input or outputs, the components in Figure.11.6 can be used as motor (see Fig.11.7) or as generator (see Fig.11.8).
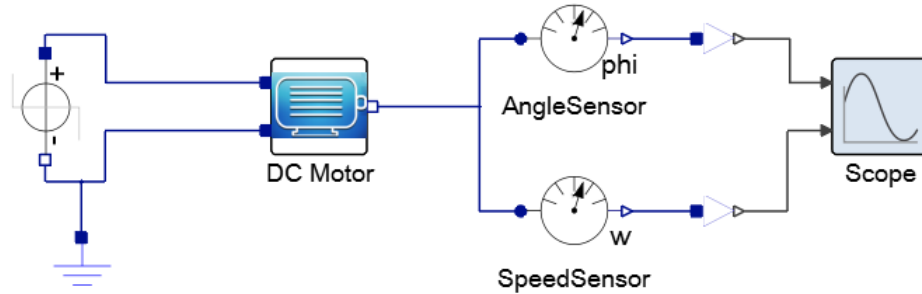
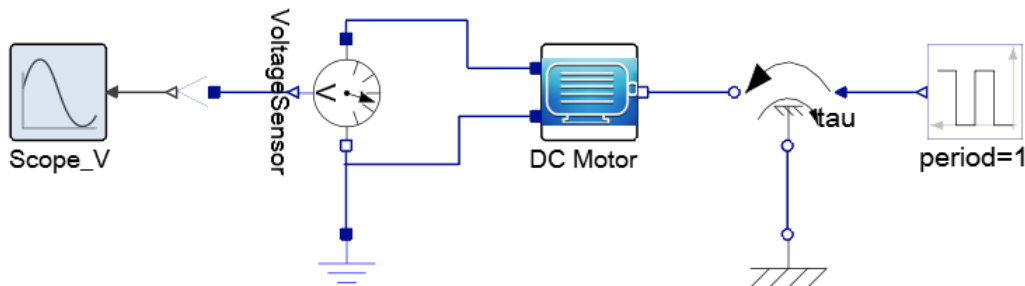Figure 11.7: Representation of the DC motor component (motor.scm)

Figure 11.8: Representation of the DC motor component (motor.scm)

11.4 Implementation in Twin Activate

The way acausal blocks are handled in **Twin Activate** is as follows. First, all the acausal blocks (including the interfacing blocks) are grouped and replaced with a single causal block. This operation is of course totally transparent to the user. For example, the model in Fig.11.7 is seen by the **Twin Activate** compiler as shown in Fig.11.9.
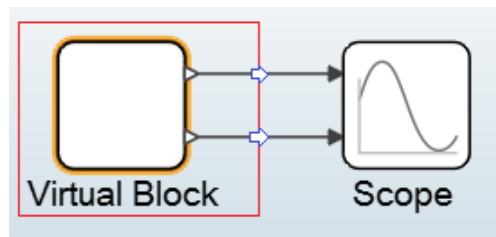
Figure 11.9: Aggregation of acausal blocks (physical components) into a causal block

Once all acausal blocks (physical components) are aggregated, only the explicit ports of the interfacing functions which are at the boundaries of the two worlds are left visible and thus the result is a causal block.

The behavior of this new causal block is defined by the Modelica program resulting from the Modelica programs of each of the acausal blocks and the way they are interconnected. This program is automatically generated, compiled and incrementally linked to the model.

This new block is almost a standard **Twin Activate** block and can be treated like any other **Twin Activate** block so that the way compilation and simulation were performed can be carried out as before.

There is however one small difficulty. So far in **Twin Activate**, not only we had assumed that the blocks where causal but also that if a block had an internal state, the dynamics of this system were explicit. If we note the inputs, outputs and the state of the block respectively by u, y and x, this condition amounts to imposing the following structure on the internal dynamics of the block

$$
\begin{aligned}
\dot{x} &= f(t, x, u) \\
y &= g(t, x, u)
\end{aligned}
$$

But the equations governing the new block can very well not be explicit. In many applications in particular in mechanics and electronics, we end up with differential-algebraic systems of equations which cannot always be converted into explicit form.

$$
\begin{aligned}
0 &= f(t, \dot{x}, x, u) \\
y &= g(t, x, u)
\end{aligned}
$$

Allowing internal implicit dynamics did not change the overall design of the **Twin Activate** compiler but did require the implementation of DAE solvers; Consequently `Ida`, `Daskr`, and `Radau` solvers were added to **Twin Activate** [4, 5, 6].
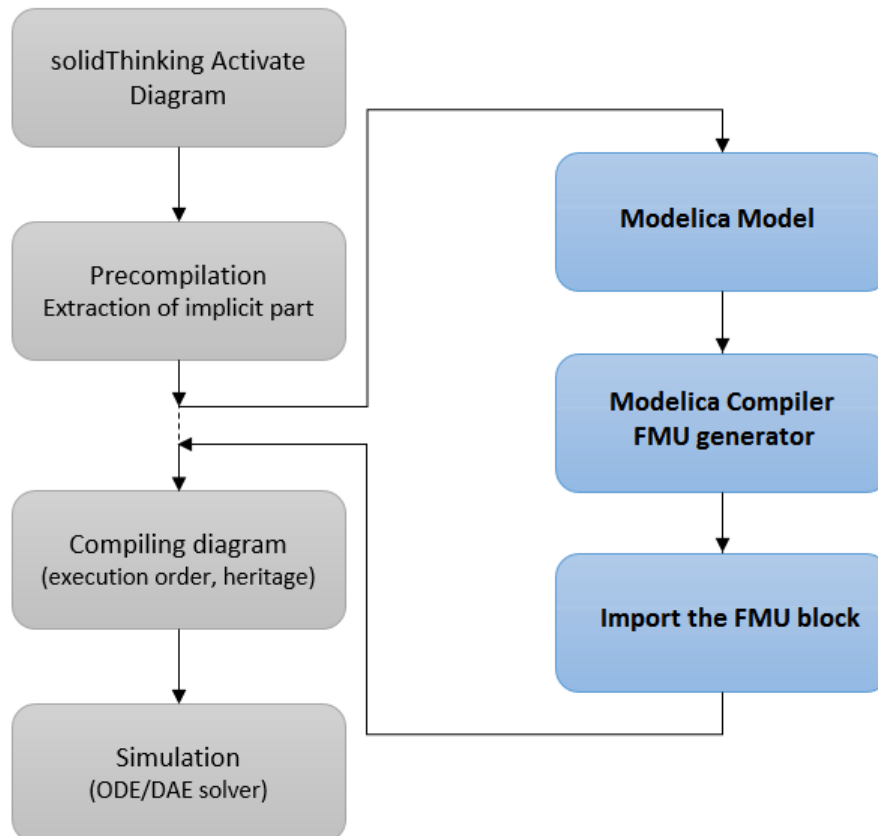
Figure 11.10: **Twin Activate** Compiler flowchart

Important Note - The ability to use Modelica-based blocks in **Twin Activate** and in particular the use of a Modelica compiler to convert the selected Modelica-based **Twin Activate** blocks into one **Twin**

Activate block, in particular an FMU block, as described below, is provided by an additional external library.

Once a **Twin Activate** diagram edited, to simulate this system several operation are performed in an automatic and transparent way. First, the precompiler removes the super blocks to obtain a flat diagram. Then if model contains physical blocks representing components, precompiler extracts them and their connecting links to generate a Modelica model. The Modelica model is then saved in a temporary Modelica file to be treated by the Modelica compiler. A Modelica compiler is used to translate the Modelica file(s) into an FMU block. The FMU block is connected to the rest of original **Twin Activate** diagram in place of removed parts. After these operations, **Twin Activate** diagram is a standard one and is compiled and simulated as usual.

11.5   Modelica custom block

Beside the Modelica components available inside the Modelica toolbox, **Twin Activate** provides a simple way to write Modelica code and test it. **Twin Activate** provides **Modelica Custom block** available in `CustomBlocks` palette. This block allows developing Modelica models from scratch.

In order to illustrate the way this block can be utilized, an example will be given in this section. Consider the model of a simple differential equation as follows:

$$\begin{cases} \dot{x} & = & 2 - 15y + u \\ \dot{y} & = & x + 6 \end{cases} \tag{11.1}$$

where the initial value of x and yy is 5.0 and 2.0 respectively. The Modelica code for this ODE follows.

```
model simpleODE
   Real x(start =5);
   Real y(start=2);
   Real u;
equation
   der(x)=2-15*y+u;
   der(y)=x+6;
end simpleODE;
```

This simple ODE can be easily modeled and simulated with the Modelica Custom block. As shown in the Flg.11.11, in the GUI of the block, the user can provide the number of explicit inputs and outputs of the block, as well as their names used in the Modelica code.
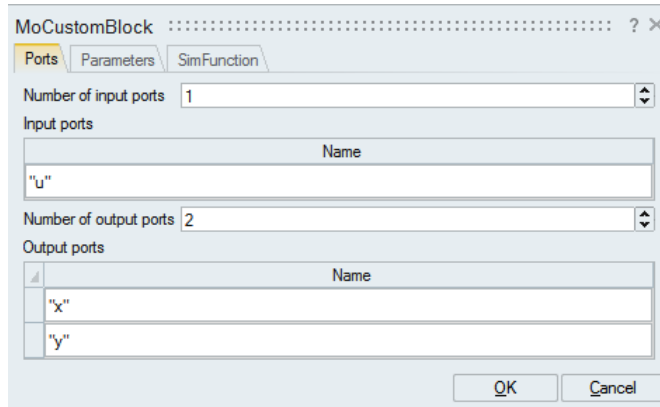
Figure 11.11: the GUI of the Modelica custom block for simulating 11.1.

Then in the Simfunction tab of the GUI, the function name and the Modelica code can be given, as shown in Fig.11.12 and Fig.11.13 .
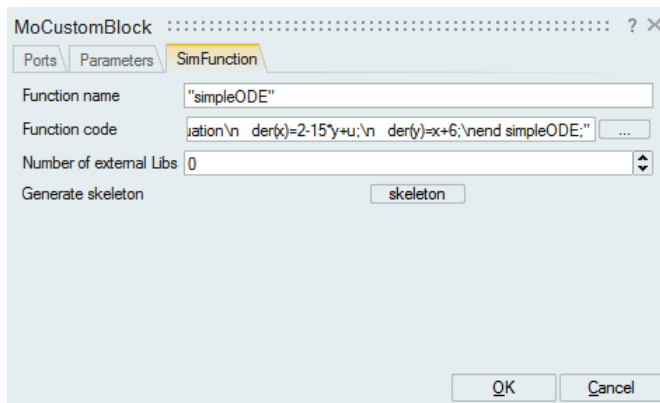

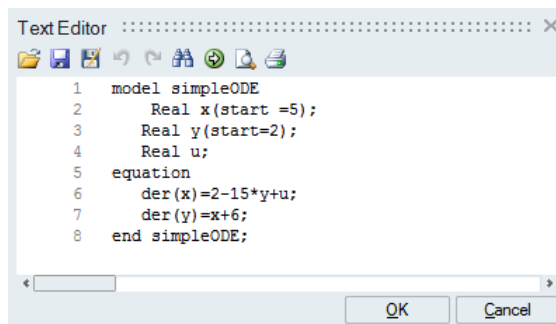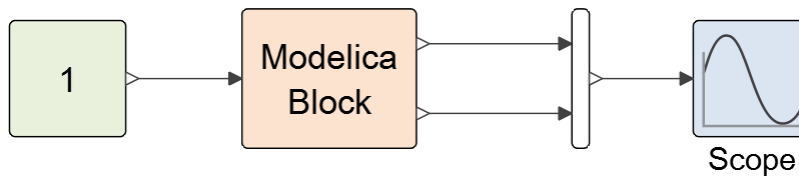Figure 11.12: the simFunction tab of the GUI of the Modelica custom block for simulating 11.1.

Figure 11.13: The Modelica code for simulating 11.1.

After building the model, as shown in Fig.11.14, it can be simulated. The simulation result is given in Fig.11.15.

simpleODE.scm

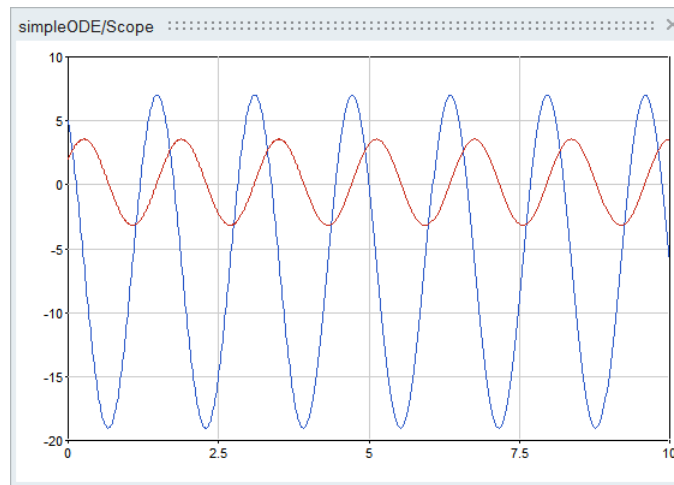Figure 11.14: Complete model for simulation of 11.1.

Figure 11.15: Simulation result for model 11.14.

Note that the input and output ports of this block is explicit and they can be connected to any other standard **Twin Activate** block. In this example, the input u is just a constant block.

Bibliography

[1] P.Frintzson, Object oriented Modeling and simulation with modelica, Wiley Interscience, 2004.

[2] A. C. Hindmarsh, "LSODE and LSODI, Two New Initial Value Ordinary Differential Equation Solvers", ACM-Signum Newsletter, Vol. 15, 1980, pp. 10–11.

[3] S.E. Mattsson, H. Elmqvist, M. Otter, and H. Olsson, "Initialization of Hybrid Differential-Algebraic Equations in Modelica 2.0", 2nd Inter. Modelica Conference 2002, Dynasim AB, Sweden and DLR Oberpfaffenhofen, Germany, 2002, pp. 9–15.

[4] L. R. Petzold, "Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations", SIAM J. Sci. Stat. Comput., No. 4, 1983.

[5] L. R. Petzold. "A Description of DASSL: A Differential/Algebraic System Solver", In Proceedings of the 10th IMACS World Congress, Montreal, 1982, pp. 8-13.

[6] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold, "Consistent Initial Condition Calculation for Differential-Algebraic Systems", SIAM J. Sci. Comp., no. 19, 1998.

[7] Masoud Najafi, Azzedine Azil, Ramine Nikoukhah, "Extending scicos from system to component level simulation", ESMc2004 international conference, Paris, France, October 2004.

[8] Masoud Najafi, Sebastien Furic, Ramine Nikoukhah, "SCICOS: a general purpose modeling and simulation environment", Modelica Conference 2005, March 7-8, 2005.

[9] Masoud Najafi, Ramine Nikoukhah, Serge Steer, Sebastien Furic, "New features and new challenges in modeling and simulation in scicos", Proceedings of the 2005 IEEE Conference on Control Applications Toronto, Canada, August 28-31, 2005.

Chapter 12

FMI (Functional Mock-up Interface)

12.1 Introduction

Automotive OEMs, who may have several hundreds of suppliers, have difficulties when they need to integrate many sub-systems from those suppliers.

Since they are specialized sub-systems, such as hydraulics, mechanical, electrical, ... the suppliers use different modeling and simulation programs for developing their sub-systems or components. In order to make sure that all of these subsystems work actually all together, the OEM should simulate them all together and in combination. For that, the system integrator should be able to work with all programs and also programs should be able to work with others. This is not of course only a problem in automotive industry but it is prominent in this industry.

A solution would be to share the source code of models and compile them in the host simulator, but very often these models have proprietary rights and can only be delivered in binary form or as libraries, which means they need special tools.

Co-simulation between simulators is a popular solution to make simulators work together in parallel. This solution works into some extent. In order to implement a co-simulation interface between simulators, a specialized interface should be developed. That means an interface per every two simulators. There are many tools and it is usually very hard to replace one by another or do all simulations in a single tool. There are some programs specialized in Co-Simulation that can interface several simulation tools with several versions, but it is not practically easy to interface all simulation tools.

Developing export and import facilities to exchange models between tools seems also to be a good solution, but that needs developing special export and import between tools. Also imported models do not usually work as well as in the original simulator due to difference in the simulator characteristics and lack of a standard format. There are several well-known interface formats, such as Modelica language, `s-function` from MathWorks, user routines in ADAMS, etc. But most of them are proprietary and cannot be used as standard for communicating between tools.

Even though Modelica is a tool independent language, it cannot be used as a standardized interface for model exchange because it needs a compiler to convert the language into the executable code.

In 2008, a large number of software and automotive companies as well as several European research centers worked in a cooperation project named MODELISAR to create the **FMI**[1] (Functional Mock-up

---

[1] More information, Detailed interface specification and images used here can be found at www.fmi-standard.org.

Interface) standard. The development of the FMI specifications was initiated and coordinated by the *Daimler* AG company. The FMI standard defines a general interface to other simulators. A component (model) which implements this interface is called **FMU** (Functional Mock-up Unit). The goal of FMI is that the calling of an FMU in a simulation environment becomes standardized and remains reasonably simple. The FMI functions can be imported and called by any external simulation environment such as **Twin Activate**. The host environment can create one or more instances of the FMU and simulate them. This is very similar to the way a model composed of several blocks can be created and simulated in **Twin Activate**. An FMU may either need a simulation environment to import it and perform numerical integration (FMI for Model Exchange) or have its own solvers/simulator (FMI for Co-Simulation).

In 2010 the first version of FMI-1.0 for Model Exchange and for Co-Simulation was released[1, 2].

In 2014, FMI-2.0 standard including both Model Exchange and Co-Simulation, was released[3]. FMI-2.0 fixed few issues in FMI-1.0 and is not backward compatible with FMI-1.0. In this document we concentrate on the FMI-2.0 version. Currently, the development of FMI is conducted by the Modelica Association.

In 2022, the FMI-3.0 standard was released. The new features that are used or implemented in **Twin Activate** are

- introduction of Clocks to more exactly control timing of events and evaluation of model partitions across FMUs,

- introduction of more integer types (signed and unsigned 8-bit, 16-bit, 32-bit, and 64-bit)

- introduction of a binary type to support non-numeric data handling, such as complex sensor data interfaces,

- extension of variables to arrays for more efficient and natural handling of non-scalar variables,

- introduction of structural parameters that allow description and changing of array sizes.

- To allow implementation of more robust and efficient co-simulation algorithms, a few new features such as Event-mode have been added to FMI for Co-Simulation.

12.1.1 FMI interface types

FMI standard which was introduced to cope with tool interface problems, provides two major solutions for interfacing tools:

1. **Interface as Model Exchange (in this document often abbreviated as ME):** FMI standardizes the model export as binary or as source code. In this interface type the model can be a system of algebraic-differential equations (at most index-1 DAE) with discrete equations. With this approach a large range of transient dynamical systems can actually be modeled. In this approach, the numerical solver which is external to the model should be compatible with the model. This interface type will be discussed more in section 12.3.

Figure 12.1: FMI for Model exchange

2. **Interface as Co-Simulation (in this document often abbreviated as CS):** FMI standardizes the co-simulation between simulation tools. In this interface type, the FMU can contain the model, the numerical solver, and the simulator. It is also possible that FMU only provide the link between two tools without containing the solver or simulator. In this approach, the tool that imports the FMU, provides the input to the FMU and receives the output of the FMU. This approach is more general and any kind of simulator and any format such as physical models, 1D-3D and 0D for Controls can be interfaced. This interface type will be discussed more in section 12.4.
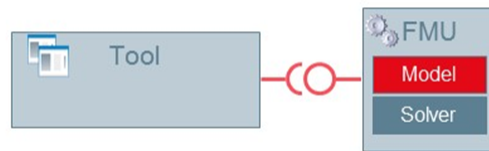
Figure 12.2: FMI for Co-simulation

12.1.2 Advantages of using FMI in industry

FMI is a free and open standard published under a BSD license and currently supported by most important actors in the simulation software domain. Up to now more than 70 tools support this standard according to the FMI-standard web-page. Both of FMI interface types allow preserving the intellectual property of the supplier over the exported model. FMI compliant tools often allow export of models freely licensed for distribution in the organization or to partners. As a result, exported FMUs most often do not require extra license from the export tool to be accessed or executed[2]. In industry, there are always only handful of specialists on modeling who develop detailed high fidelity models using modeling tools. They develop the model and deliver it to a large number of users who exploit the model for other applications such as

- Model parameter identification,

- Model sizing,

- Controller design,

- Model specific analysis such as frequency analysis,

- Pre and post-processing,

None of these applications needs the modeling tool which very often requires costly license and a good knowledge to use. In this situation, it is extremely useful to embed the model in an FMU and share it with other users. The FMU or model can be deployed from simulation specialists to designers, domain specialists, control engineers in other departments. That is the advantage of FMI, less costly modeling tool and more less expensive execution tools. As an example consider the model of the engine developed by the mechanical engineering department. This FMU can be deployed to Electrical department for ECU design, to Thermal department for thermal requirements verifications, or to control department for design appropriate controllers. In each of these departments, several engineers can work on the same FMU with different FMI-compliant tools and each engineer can run several variants of FMUs at the same time.

An FMU can be used by different groups in different domains, i.e., multi-domain collaboration. Engineers in different domains such as mechanical, Electrical, system, control, fluid, and thermal can work

---

[2]Except for FMI for Co-Simulation with external solver where the remote simulator is still required.

together with FMUs. They can share FMU models but staying with their tool of choice. Now, with advent of a new standard, every tool supporting FMI can virtually be connected without developing new interface.

Figure 12.3: Using FMUs to share models

It is, however, important to highlight that the FMI is basically developed for system level modeling where signals are scalar. As a result, FMI is not really suitable for interfacing finite-element or finite-volume problems where huge number of variables should be interfaced. If the number of connecting variables in finite-element or finite volume problems such as CFD is kept low, they can be interfaced by this standard. An example of such connection would be the placement of flow or temperature sensors in a fuel system modeled by a CFD tool.

12.2   Internal structure of an FMU

An FMU implementing the FMI interface is a model distributed in one zip-file containing several files with a structure similar to the diagram shown in Fig.12.4:
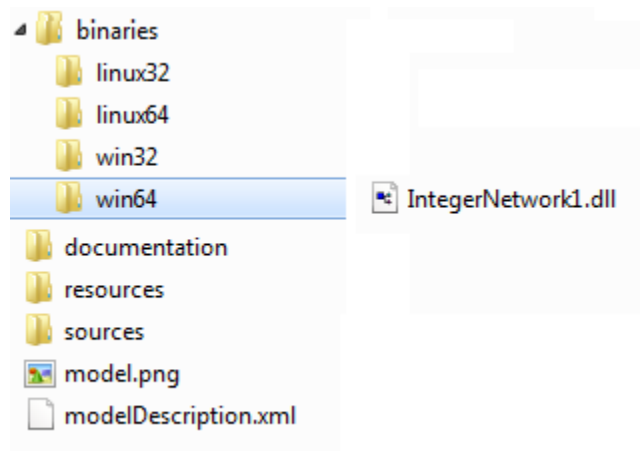
Figure 12.4: Internal structure of an FMU.

An FMU is composed of several files that will be discussed as follows:

- **An XML file (modelDescription.xml):** This XML file contains information on the model and the definition of all variables of the FMU in a standardized way. XML file indicates the FMI version, whether the FMU is for Model Exchange and/or for co-simulation, and the features of FMI standard supported by the FMU. The XML file also lists the exposed model variables including inputs,

outputs, states, state derivatives, and model parameters. Model parameters can be changed at the beginning of the simulation of even during the simulation for `tunable` parameters. An important information included in the XML is the model structural information, i.e., the variables who are output, derivatives, or unknowns at initial time. The dependency of output, derivative and initial unknowns on other variables are also provided. This information is crucial for detecting algebraic loops and correctly handle them in a model aggregated by several FMUs. The XML can also contain vendor information. It is important to indicate that information stored in the XML is not needed for execution of the model. That allows having an executable code with a minimum overhead. Below is an example of an FMI model description issued from Modelica. For more details on the XML file, interested readers are referred to the section `FMI description Schema` in [3].

```xml
<?xml version="1.0" encoding="UTF8"?>
<fmiModelDescription
  fmiVersion="1.0"
  modelName="ModelicaExample"
  modelIdentifier="ModelicaExample_Friction"
...
  <UnitDefinitions>
     <BaseUnit unit="rad">
        <DisplayUnitDefinition displayUnit="deg" gain="23.26"/>
     </BaseUnit>
  </UnitDefinitions>
  <TypeDefinitions>
     <Type name="Modelica.SIunits.AngularVelocity">
        <RealType quantity="AngularVelocity" unit="rad/s"/>
     </Type>
  </TypeDefinitions>
  <ModelVariables>
     <ScalarVariable
        name="inertia1.J"
        valueReference="16777217"
        description="Moment of inertia"
        variability="parameter">
        <Real declaredType="Modelica.SIunits.Torque" start="1"/>
     </ScalarVariable>
...
  </ModelVariables>
</fmiModelDescription>
```

- **C source file or compiled C dynamic linked libraries:** These files define the model equations converted into causal form. If the source code is provided, the source code is put in the fmu://sources folder. If the model is provided in binary form, it is placed inside the folder fmu://binaries/win64. Binaries for several platforms, such as `win32`, `linux32`, and `linux64`, can be provided by an FMU.

- **Documentation (optional):** Additional optional documentation can be included in the FMU inside the fmu://documentation folder.

- **resources (optional):** Additional optional data can be included in the FMU inside the fmu://resources folder. This folder can be used for storing tables, maps or any additional data.

12.3 FMI for Model Exchange (ME)

The FMI for Model Exchange (ME) interface defines an interface to the model of a hybrid dynamic system described by differential, algebraic and discrete-time equations. In this kind of the interface, ordinary differential equations with events and zero-crossings are handled. Algebraic equation systems

might be contained inside the FMU. Also, the FMU might consist of discrete-time equations activated by events, for example a sampled-data controller. Since the overhead is small, this kind of FMU can be used for very large models as well as for small embedded systems. An FMU for Model Exchange operates in three main modes, *Initialization mode*, *Continuous-time mode*, and *event-mode*. The state-machine of the FMU/ME has been illustrated in Fig.12.5.
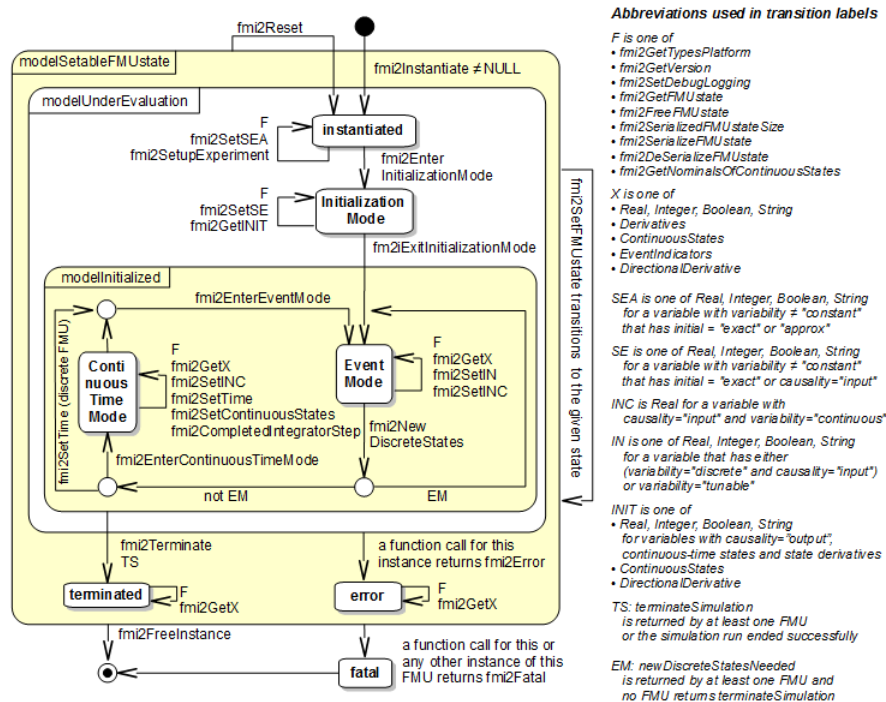

Figure 12.5: State Machine for FMI-2.0-Model Exchange.

In *initialization mode* the initial values of continuous-time states, the previous (internal) discrete-time states, are computed at the start time. In this mode some extra equations that may not be present in other modes may be used. As an example, suppose that an FMU should start at equilibrium point. The equations defining the start value of state at equilibrium are defined only in this mode.

The *continuous-time mode* is used to compute the values of all continuous-time variables between events by numerically solving ordinary differential and algebraic equations. All discrete-time variables are fixed during this phase and the corresponding discrete-time equations are not evaluated.

The *Event Mode* is used to compute new values for all continuous-time variables, as well as for all discrete-time variables that are activated at the current event instant, given the values of the variables from the previous instant. This is performed by solving algebraic equations consisting of all continuous-time and all active discrete-time equations.

The operating mode jumps from continuous-time into event-mode, if an event happens. There are four event kinds in FMI:

- **State-event (zero-crossing event):** Event which is triggered at the time instant where the event indicator function changes from $z > 0$ to $z \leq 0$, or vice versa. Discontinuities in a model are usually defined as zero-crossing functions (event-indicators).

- **Time events:** Event that is triggered at a predefined time instant. Since the time instant is known in advance, the integrator can select its step size so that the event instant is directly reached.

Therefore, this event can be handled efficiently.

- **Step-event:** Event that might occur when the simulator finishes a completed integrator step. This event is not usually defined by the FMU. The simulator may use it, for example, to dynamically switch between different states, or storing data into a file.

- **External event:** This kind of event is triggered by the environment. This event type is triggered if at the current time instant, at least one discrete-time input changes its value, a continuous-time input has a discontinuous change, or a tunable parameter changes its value.

Upon happening any of these events, the continuous-time integration is stopped and the FMU goes into the event-mode.

12.4   FMI for Co-Simulation (CS)

Modeling complex heterogeneous systems in engineering usually leads to hybrid systems of differential and algebraic system of equations with discrete-time equations. Such complex multi-disciplinary systems cannot often be modeled and simulated in one simulation tool alone. Sub-system models are often available only for a specific simulation tool. For example special tools for CFD (Computational Fluid Dynamics) models or integrated circuit systems. In many situations, the sub-systems shall be simulated with the simulator which suits best for the specific domain. Thus for the simulation of multi-disciplinary models it is often reasonable or even necessary to couple different simulation tools with each other or with real world system components. Co-simulation is a general approach for the joint simulation of models developed with different tools where each tool treats one part of a modular coupled problem. The data exchange (input and output variables, status information) between sub-systems is restricted to discrete communication points. In the time between two communication points, the sub-systems are solved independently from each other by their individual solver.

Main-secondary is a common method in Co-simulation. In a main-secondary approach, the secondary solvers simulate sub-models whereas the main (solver) is responsible for both coordinating the overall simulation as well as transferring data. In the main-secondary approach, instead of coupling the simulation tools directly, it is assumed that all communication (input/output data exchange) is handled via a main. Main algorithms control the data exchange between sub-systems and the synchronization of all secondary simulation solvers. Besides distribution of communication data, the main checks the connection graph, chooses a suitable simulation algorithm and controls the simulation according to that algorithm.

The secondary solvers are simulation tools, which are prepared to simulate their subtask. They are able to communicate data, execute control commands and return status information.

FMI for Co-Simulation unifies the main-secondary interface and supports from simple co-simulation algorithms to more sophisticated ones. FMI provides a small set of C-functions to implement the interface. It is important to note that the main algorithm itself is not part of the standard FMI for CoSimulation. The main is free to choose the appropriate co-simulation algorithm.

FMI for Co-Simulation supports two possible ways to provide sub-systems for co-simulation: sub-systems (models) packaged along with their respective solver, which can be simulated as stand-alone components, see Fig.12.6, or a wrapper FMU which calls its respective simulation tool to simulate the sub-system (model), which is installed externally to the FMU, see Fig.12.7. The FMI import in **Twin Activate** supports both implementations of FMI for Cosimulation. In FMI export in **Twin Activate**, the solver is embedded in the FMU and no **Twin Activate** installation is required to run the FMU in other
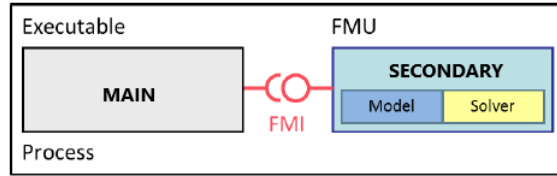
Figure 12.6: Co-simulation with generated code on a single computer.

environments.

In the first case, the model and its solver are exported together as runnable code, whereas in the second, the FMU plays as a wrapper to communicate with the exporter simulator.
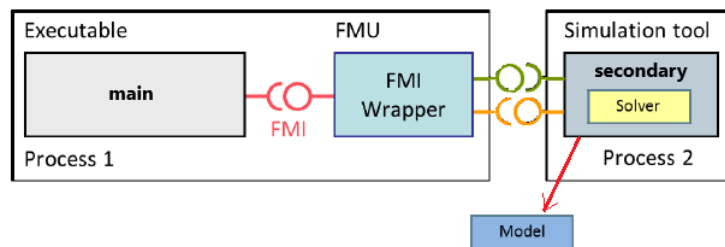

Figure 12.7: Co-simulation with tool coupling on a single computer.

In FMI for co-simulation, all information relevant for the communication in the co-simulation environment is provided in the XML-file. The XML file includes a set of capability flags to characterize the ability to support advanced main algorithms, e.g. the usage of variable communication step sizes, higher order signal extrapolation, or others. In **Twin Activate**, the simulator is considered as the main solver and each FMU block as a secondary solver.

In FMI for co-simulation, a secondary system operates in two main modes, *initialization mode*, and *step mode*. The *initialization mode* is used to compute the initial values for internal variables of the co-simulation secondary systems at the start time.

This mode supports a few special cases:

1. some extra equation may be used that are not active during the rest of the simulation.

2. algebraic equations can be solved.

3. If the secondary system is connected in loops with other FMUs, iterations over the FMU equations are possible.

The *step mode* is used to compute the values of all variables at communication points. In this mode, the main (solver) provides the input of secondary solvers and requests their outputs.

12.4.1 Co-simulation algorithms

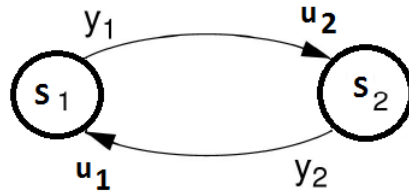Consider a model composed of two parts (submodels) defined in two tools as shown in Fig.12.8.

Figure 12.8: Co-simulation with tool coupling on a single computer.

The outputs of submodels are computed as following equations.

$$\begin{cases} y_1 = F_1(y_2) \\ y_2 = F_2(y_1) \end{cases}$$

In order to co-simulate this system, this equation should be solved at each communication instant. F_i functions are usually complex, non-linear, and time-dependent relationships, as a result, iterative methods are the only choice for solving such equations. Three common methods used for solving them are Gauss-Seidel and Gauss-Jacobi and Newton methods. The Gauss-Seidel iteration method is represented as follows.

$$y_1^{i+1} = F_1(y_2^i)$$

$$y_2^{i+1} = F_2(y_1^{i+1})$$

$$i = 0, 1, 2, ...$$

The Gauss-Jacobi iteration method is defined as

$$y_1^{i+1} = F_1(y_2^i)$$

$$y_2^{i+1} = F_2(y_1^i)$$

$$i = 0, 1, 2, ...$$

These iterative methods are simple fixed-point methods and their convergence(at most linear) depends on F_i functions. If the Fixed Point converges very slowly or diverges the Newton-type methods can be used. These methods are, however, more complicated, because in each step the Jacobian must be computed which results in higher number of simulator calls. The method which is used actually in **Twin Activate** is GS1, i.e., the Gauss-Seidel method with only one iteration. This method has been illustrated in Fig.12.9;
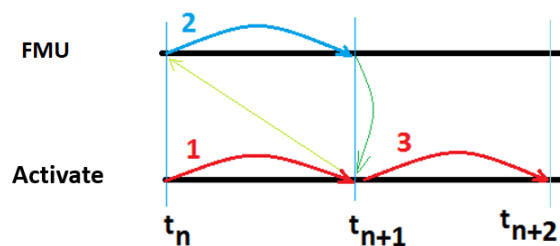

Figure 12.9: Co-simulation method used in **Twin Activate**.

The simulator of **Twin Activate** which acts as the main performs the following method for the co-simulation and synchronization between tools, as shown in 12.9.

1. The simulator advances the time from t_n to t_{n+1}, and updates the output of all blocks, except those of external tools (secondary). Optionally, the secondary system (FMU) outputs are extrapolated at a desired order if available from the FMU.

2. External tools are called one by one and asked to advance the their time from t_n to t_{n+1} and their output is requested at t_{n+1}. Optionally, the secondary system (FMU) inputs are linearly interpolated.

3. The **Twin Activate** simulator takes a new step from t_{n+1} to t_{n+2} using the new output value of secondary systems computed at t_{n+1}.

4. goto 1

Usually in order to get a correct result, the communication step-size (time instant between two communication instants) should not be larger than two times the highest frequency content (one-half the smallest time constant; Nyquist theorem) of the secondary systems, and in practice should usually be smaller than this. Smaller communication step-size results in better convergence but longer simulation time. In **Twin Activate** the user can control the communication step-size, see 12.5.5.

12.4.2 Advanced: FMI import preserving full output/input dependency property

[3] A challenge in importing an FMI generated is the treatment of output/input dependencies. In an Activate basic block output/dependencies are expressed as a vector of dependencies specifying the inputs that affect any of the outputs. So the dependency is solely a property of an input port. A basic block computes all its outputs in the same call, so all its dependent inputs must be up to date when the call is made. An FMU, on the other hand, specifies output/input dependencies as matrix specifying which output depends on each input. This input/output dependency matrix may also be different during the initialization. It provides routines that allow the computation of output ports separately and take advantage of variable caching.

A way to deal with this situation, which was the way **Twin Activate** imported Modelica parts, is to simply project the matrix of dependencies into a vector. This conservative approach properly assigns dependencies in **Twin Activate** but "loses" information along the line. This may lead in particular to detection of algebraic loops by the **Twin Activate** compiler that are not true algebraic loops (artificial algebraic loops). Even though there are ways to break algebraic loops in a **Twin Activate** model, we decided to detect artificial algebraic loops and treat them properly. A very simple example that illustrates the problem is shown in Figure 12.10.

After compiling the Modelica part, a virtual model similar to what is shown in Figure 12.11 is obtained. In the generated FMU, there is a direct dependency between the `SignalCurrent` input port (**in**) and the `CurrentSensor` output signal (**A**). The dependency is depicted by a red dashed line if Figure 12.11. If the dependency matrix is converted into a vector, output ports **A** and **V** depend on input port **in**. This direct dependency results in an artificial algebraic loop that does not really exist.

In order to solve this problem, the output/input dependency property of the basic blocks can only be set for ports separately so the imported FMU cannot be represented by a single FMU; but it can be properly represented as a network of basic blocks. The topology of this network depends on the number of inputs and outputs of the FMU and the FMUs output/input dependency matrix. A programmable super block is thus used for importing the FMU. The block main parameter is the FMU filename. By reading and parsing the XML inside the FMU, the block generates a network of basic blocks, as shown in Figure

---

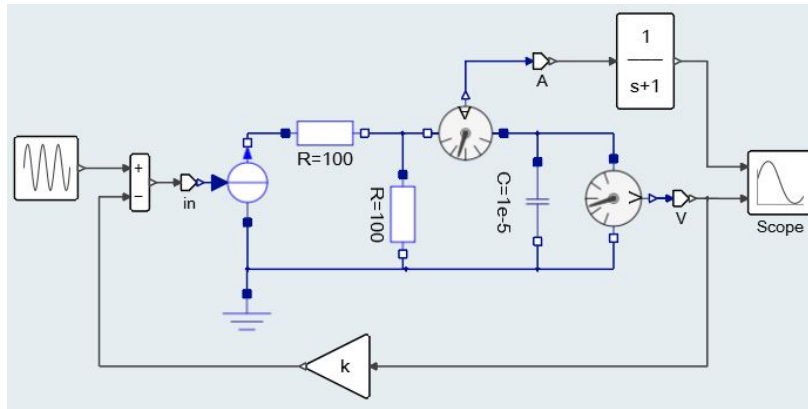[3]This section can be skipped without losing the continuity.

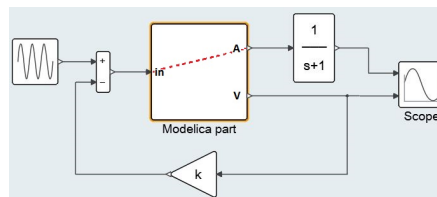Figure 12.10: A simple model mixing Modelica and **Twin Activate** blocks

Figure 12.11: The model in Figure 12.10, after converting the Modelica part into an FMU block.

12.12, during the model compilation stage. The network contains a central basic block, always present, and other blocks associated with each output port. Figure 12.12 illustrates the resulting network for an imported FMU with two inputs and four output ports.
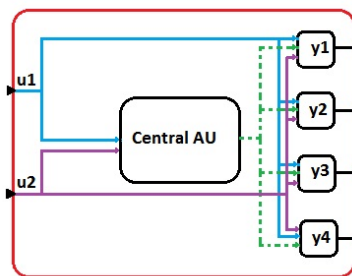

Figure 12.12: Automatically generated network of basic blocks from FMU import for an FMU with one input and four output ports.

In resulting network of basic blocks, each output unit keeps the direct-feedthrough property of the corresponding FMU output port. The central unit on the other hand does not have the direct feed-through. In order to communicate within this network, the central block outputs the internal structure which contains the FMU context and shares it with other blocks of the network. Note that this process is completely transparent for the user, who never sees the network in Figure 12.12. Hence generating a programmable network of blocks instead of single block for FMU import, avoids creation of artificial algebraic loops.

12.5   FMI Import in Twin Activate

Twin Activate can simulate FMI-1.0, FMI-2.0, and FMI-3.0 block of both Model-exchange and co-simulation kinds. The **FMU Import** block (see Fig.12.13) available in the **CoSimulation** palette is used to import and simulate FMU blocks. This block automatically detects the FMI version and FMI kind and chooses appropriate methods to simulate them. By double clicking on the block, a block dialog (GUI) is opened as shown in Fig.12.14

Figure 12.13: **FMU Import** block from **CoSimulation** palette
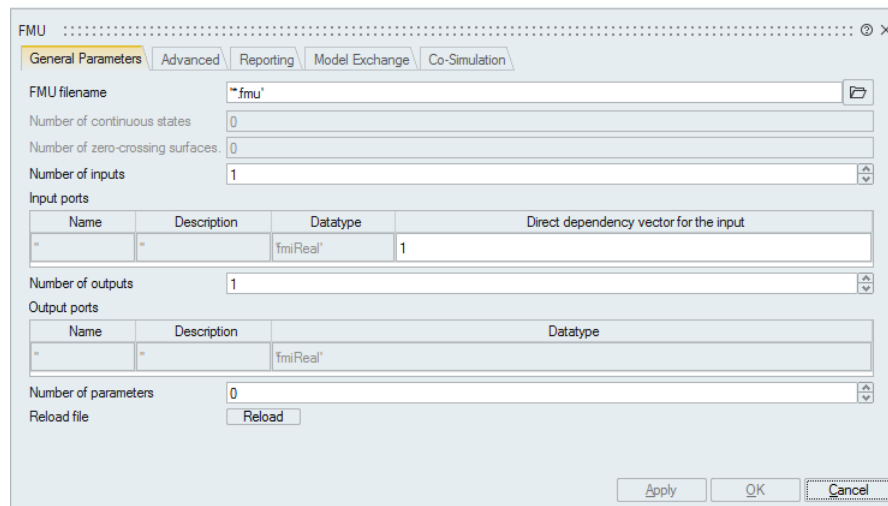
Figure 12.14: Block Dialog for **FMU Import** (empty)

This dialog allows for selecting and loading an FMU. Once selected, the user can either fill the FMU information fields such as number of input and outputs, or click on the **Reload** button. The button temporarily reads the FMU and fills the information fields of the dialog automatically, as shown in Fig.12.15.

In the Parameter field, the FMU variables whose start value can be changed are displayed. The user can provide new values for these variables. The new values are taken into account before the start of the simulation.

There are, however, few features of FMI that are not supported by the FMI Import block in **Twin Activate**.

- Tunable parameters are not supported.

- The FmuSetState and FmuGetState features of FMU are not fully exploited for repeating the steps and solving algebraic loops.
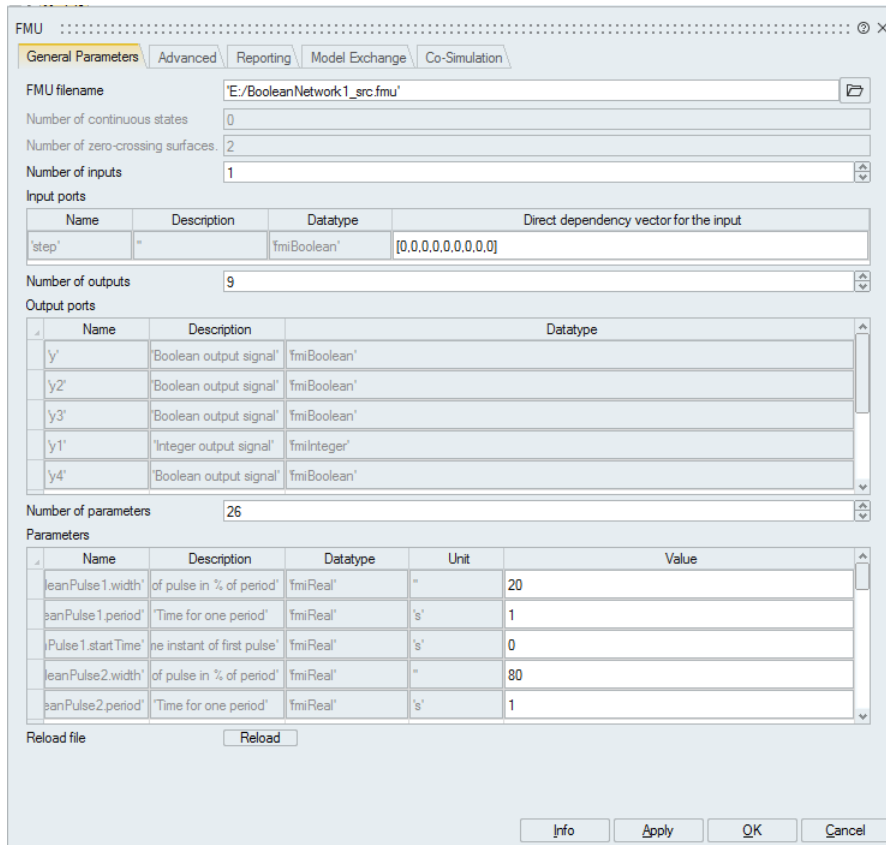
Figure 12.15: Graphical user interface of the block **FMU-Import** (filled)

12.5.1 Direct dependency vector for inputs (Feedthrough)

An FMU block usually provides the dependency of output variables on its variables including its input variables. This information is transformed into feedthrough information of the FMU block in **Twin Activate** (see 9.1.1). This information is displayed in the FMU GUI, as shown in Fig.12.15.. For each input port, a dependency vector is provided which indicates the output ports that depend on the this input port. In this vector zero(0) means no dependency and one(1) means direct dependency of output on input.

12.5.2 Advanced tab

`Run as "Model Exchange":` In FMI-2.0 and FMI-3.0, the FMU can provide both ME and CS implementations for a model. In this case, the user can choose the implementation that should be simulated. The "Run as Model Exchange" check-box, if checked, forces the simulator using the Model Exchange implementation.

`Tolerance controlled:` This check-box allows choosing if the FMU should be tolerance-controlled or not. If the checkbox is On, then the FMU model is called with a numerical integration scheme where the step size is controlled by using the relative error tolerance for error estimation. In such a case, all numerical algorithms used inside the model operate with an error estimation of an appropriate smaller relative tolerance. If the fixed-step solver is used, this checkbox may be Off.
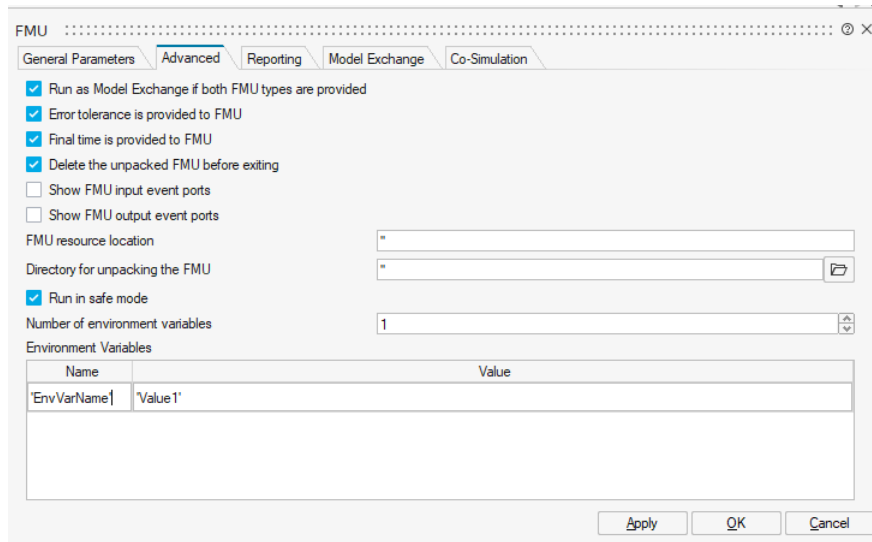
Figure 12.16: Graphical user interface of the block (Advanced tab)

`Delete the unpacked FMU before exit:` The simulator unzips the FMU in a folder on each run and deleted the folder once the simulation finished. This check-box allows keeping the folder after finishing the simulation.

`Show FMU input event ports:` This check-box allows displaying/hiding the input event ports of the block. The FMU import block has got two input event ports, one for external events and another for End-event. The external event which can be activated by discrete events of **Twin Activate** pushes the FMU (of type ME) into the event mode. This is important if, for example, the input of the block is discontinuous. The End-event pushes the block into the terminate mode, where the output of the block remains unchanged until the end of the simulation.

`Show FMU output event ports:` This check-box allows displaying/hiding the output event ports of the block. The FMU import block has got two output event ports, one for discrete event that happen inside the FMU and another for End-event. The discrete event is triggered for example when a time event is programmed inside the FMU block. The output End-event is triggered when the FMU requests a termination. This event can be used to terminate the simulation.

`FMU resource location:` This field is used to provide the location of the resource folder to be given to the FMU. If this field is left empty, the FMU is the default location which is fmu://resources.

`Directory for unpacking the FMU:` This field allows unzipping the FMU in a specified folder. If this field is left empty, the FMU is unzipped in the temporary folder of **Twin Activate**.

`Run in safe mode:` If enabled, the FMU is loaded inside another process. This protects **Twin Activate** from possible crashes in the FMU. The simulation is slower in this mode. Both FMI-1.0 and FMI-2.0 of ModelExchange and CoSimulation kinds can be simulated in this mode.

`Environement variables:` If safe-mode is activated, the user can define environement variables to be visible only in the new process by the FMU.
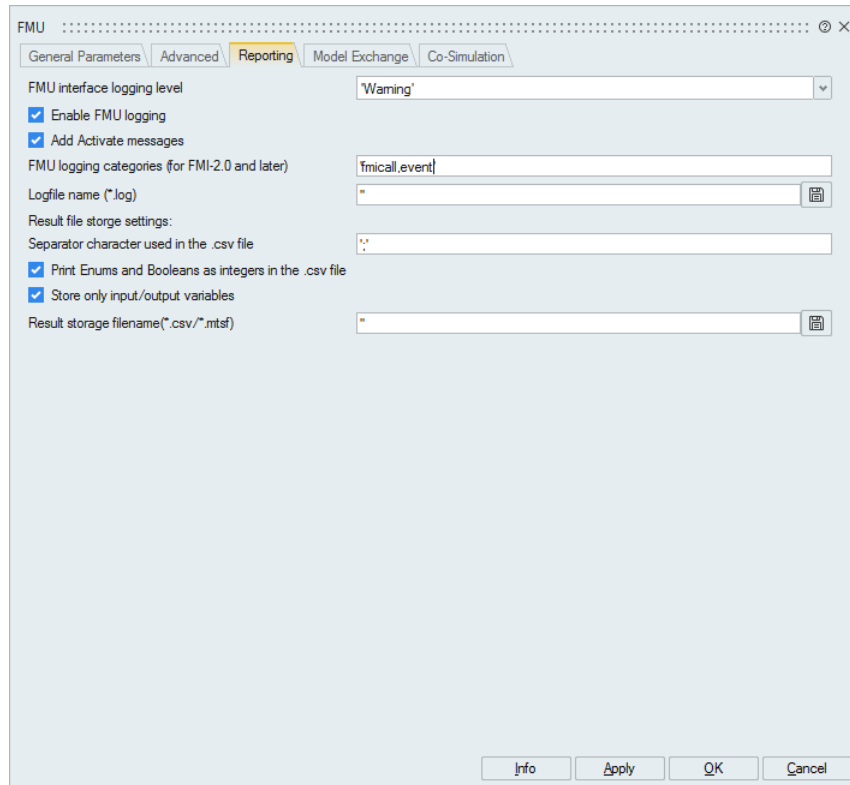
Altair Twin Activate 2023   Extended Definitions 246

Figure 12.17: Graphical user interface of the block (Reporting tab)

12.5.3 Reporting tab

`FMU interface logging level:` The list-box allows choosing the logging level for the interface to the FMU block. For example, if the user chooses the *Warning* level, all logger messages emitted by FMU with status *fmi2Warning*, *fmi2Error* and *fmi2Fatal* will be stored in the logfile. Possible values *Nothing*, *Fatal*, *Error*, *Warning*, *Info*, *Verbose*.

`Enable FMU logging:` This check-box enables or disables the whole logger.

`Add` **Twin Activate** `messages:` If this check-box enabled, the messages emitted by **Twin Activate** interface to the FMU will be added to the logfile. If you need to catch only the messages emitted by the FMU, you may turn this checkbox off.

FMU logging categories (for FMI-2.0 and FMI-2.0): In FMI 2.0, the FMU is given the possibility to filter logger messages. The user can enter the required log categories. Log categories should be protected by double quote and separated by comma, for example "fmiCall;events;logger2". Only messages with given categories will be stored in the logfile. If left empty, all messages are stored, regardless of their category.

`Logfile name (*.log):` The log filename to be created. If this field is left empty, no logfile is created. `stderr` as a filename is supported.

`Separator character used in CSV file:` The separator character in the output CSV file.

`Print Enums and Booleans as integers in CSV file:` If this check-box is activated, the Enum values are stored as integer, otherwise their text values are stored.

`Store only input/output variables:` If this check-box is activated, the CSV or HDF file will store only input and output variables. This allows reducing the result file volume.

`Result storage filename(*.csv/*.mtsf):` The full path of the output result file. The output file can either an CSV file or an HDF file (with extension *.mtsf)[4].

12.5.4 ModelExchange tab

`Solver mesh-points events (step events):` If checked, whenever a step event happens, the value of variables at that time instant is stored in the CSV file.

`Zero-crossing events (state events):` If activated, whenever a state event (zero-crossing event) happens, the value of variables at that time instant is stored.

`Time events:` If activated, whenever a time event happens, the value of variables at that time instant is stored.

`Every output update:` If activated, whenever the block outputs are updated, the value of variables at that time instant is stored.

`Meshpoints points:` If activated, the value of variables at that solver mesh-points are stored.

`Super dense time instants:` If activated, the value of variables at super dense time in discrete event iterations are stored.

12.5.5 CoSimulation tab

`Preferred fixed communication step size:` An FMU of kind co-simulation is simulated by its internal solver. The FMU and **Twin Activate** exchange (update) their respective output and input values at intervals called communication points. The interval between communication points is called communication step-size which may vary during the co-simulation. The communication step size can be set by the "Preferred fixed communication stepsize" parameter in the FMU block under the Co-Simulation tab as shown in Fig.12.18.
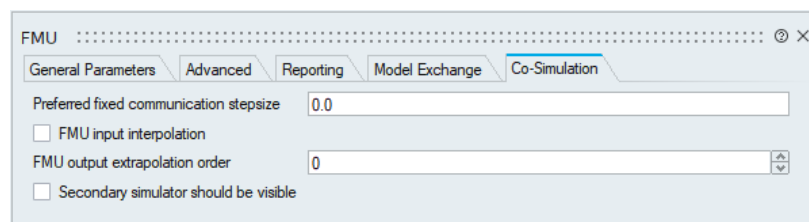

Figure 12.18: Setting co-simulation parameters in the FMI import block

The default value is set to zero (0), which means that the communication instants are governed by the time-step of the **Twin Activate** solver. Therefore, in order to control the communication interval you can adjust the "maximum step size" parameter for the **Twin Activate** solver. Note that, if the **Twin Activate** solver is a fixed-step size solver, the "maximum step size" parameter will be the co-simulation communication step size. In **Twin Activate**, "maximum step size" parameter is found under the Solvers tab of the Simulation Parameters dialog.

Alternatively, you can change the "Preferred fixed communication step size" to a non-zero value and this will set the rate of communication between the FMU and **Twin Activate**, regardless of the step-size

chosen by the **Twin Activate** solver. Note that, in this case the communication step size is fixed. If two sub-systems (FMU and **Twin Activate** part) tightly interact, fixed communication points methods does not work and the co-simulation may diverge. In this case, Newton methods should be employed, although this is not yet supported in **Twin Activate**.

`FMU Input interpolation`: In order to enable the secondary system to interpolate the continuous real inputs between communication steps, **Twin Activate** can provide the first derivatives of the inputs with respect to time using the numerical differentiation method. If this checkbox is activated, the first derivative of inputs is computed and is given to the FMU. This may be useful if the FMU can exploit the derivative of its input. Note that since the derivative is computed numerically, if the signal is not smooth enough, the computed derivative is poor. In this case the checkbox is better to be off.

`FMU Output interpolation`: FMU can provide higher order derivatives of their real outputs to allow extrapolation of the output variables between communication steps. In order to exploit the derivative and read smoother signal from the FMU, the user can choose the extrapolation order to be used. Based on the chosen extrapolation order, the derivative of outputs of FMU/CS block is used for extrapolation of outputs. Note that no extrapolation order bigger what is provided by FMU can be used, otherwise a warning message will be raised. The FMU provides the highest order available and you can set the extrapolation to this order or any lower order.

`Secondary simulator should be visible`: If activated, the FMU/CS is co-simulated in interactive manner.

12.5.6 Example

In order to demonstrate the way an FMU can be imported, export a model as FMU with another simulator, e.g., MapleSim form MapleSoft. You can also choose an FMU from the FMI public SVN server[4]. Make sure that the FMU you choose is compatible with the **Twin Activate** binaries. For example if the **Twin Activate** is compiled for a win64 machine, your FMU should contain win64 DLLs.

Consider for example the bridge rectifier modeled in MapleSim as shown in 12.19. With input signal $u = 10\sin(20\pi t)$, the simulation result is given in Fig.12.20.

In MapleSim, export this model as FMI-2.0 for ModelExchange or Co-Simulation and load it in **Twin Activate** as shown in Fig.12.21 and Fig.12.22.

Choose appropriate simulation parameters in **Twin Activate**, i.e., Final time=1.0, Maximum step-size=0.0001. and run the model. You should get the same result as in MapleSim, as shown in Fig.12.23.

12.6 FMI export in Twin Activate

Twin Activate can export super blocks as FMI-2.0 and FMI-3.0. The FMU exported in **Twin Activate** is standalone, i.e., in the form depicted in Fig.12.6. In the FMU, the numerical solver used in the model setting is embedded in the FMU. So no **Twin Activate** installation is required to run the FMU in other environments.

In order to export a model or a part of a model, the interested region should first be placed into a super block. Once converted into a super block, right-click on the `Code Generation and Export` menu. The CodeGeneration GUI is opened, as shown in Fig.12.24.

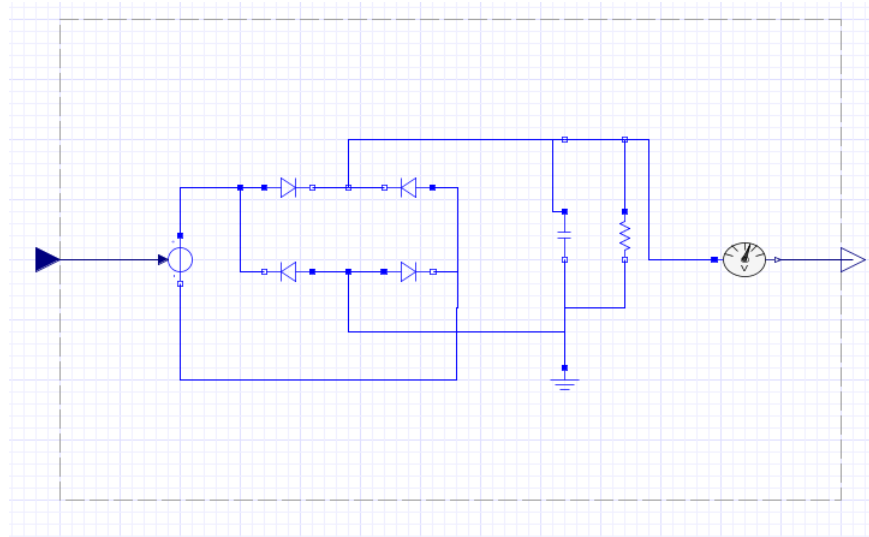---

[4]https://svn.fmi-standard.org/fmi/branches/public
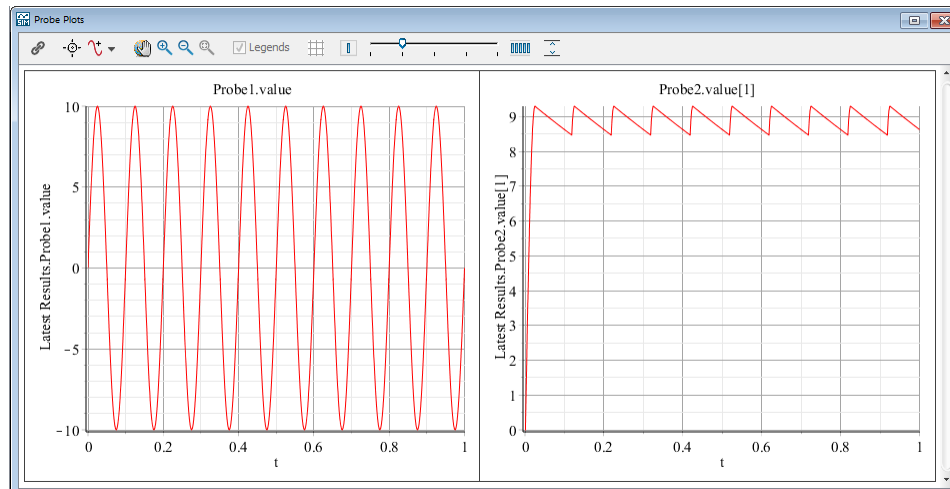
Figure 12.19: A bridge rectifier modeled in MapleSim.

Figure 12.20: Simulation result of model in Fig.12.19 in MapleSim.

By selecting FMU as target, two export methods will be available.

- Standard Code Generation: In this method the same libraries used in **Twin Activate** will be packaged inside the FMU. This method supports more **Twin Activate** blocks for FMU export, but since all necessary libraries binaries are included in the FMU, the FMU size may be big.

- Inline Code Generator: In this method an optimized C code is generated for **Twin Activate** block. The generated C code can be exported in the FMU and recompiled for another OS. Some **Twin Activate** blocks are not supported by Inline code generator. For example, the Modelica blocks cannot be exported as FMU for Model-Exchange. Note that Modelica blocks can be exported in Standard code generation, as well as in Inline code generator for Co-simulation. Inline Code Generator also can export as FMI-3.0 and supports arrays and clocks.

Note that The name and identifier of the exported FMU will be the name of the super block. Hence, if the name of the super block contains unsupported characters, they are converted to `"_"`. For example
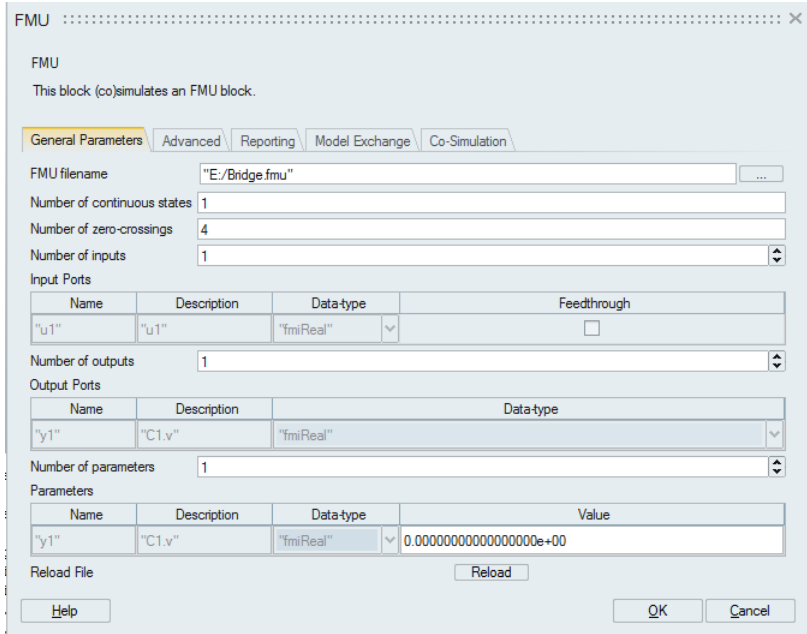
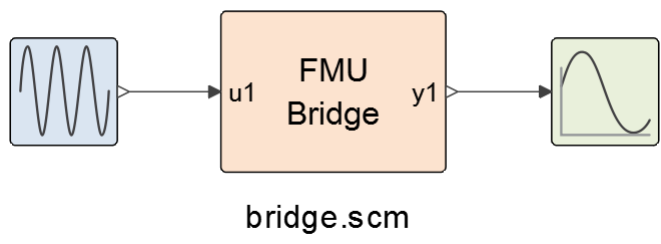Figure 12.21: FMI Block dialog in **Twin Activate**

bridge.scm

Figure 12.22: Bridge model (bridge.scm)

if the name of the super blocks is `"SuperB !%lock"`, `"sb_SuperB___lock"` will be used instead.

In the Code generator GUI, the super block can be exported with Model-Exchange and/or Co-simulation interface types. If the super block is exported as FMI for co-simulation, the **Twin Activate** simulator as well as the simulation parameter values in the "setup" menu will be used in the exported model. If a model is implicit, i.e., contains implicit blocks, such as ImplicitDerivative, Constraint, or Automaton, it can only be exported as FMU for CoSimulation. In this case, an implicit solver such as, Ida, Daskr, or Radau(DAE) is used inside the FMU.

12.6.1 Nested FMU

A superblock containing other FMUs (FMI-1.0 or FMI-2.0 of kind ModelExchnage or cosimulation) can be exported as a new FMU.
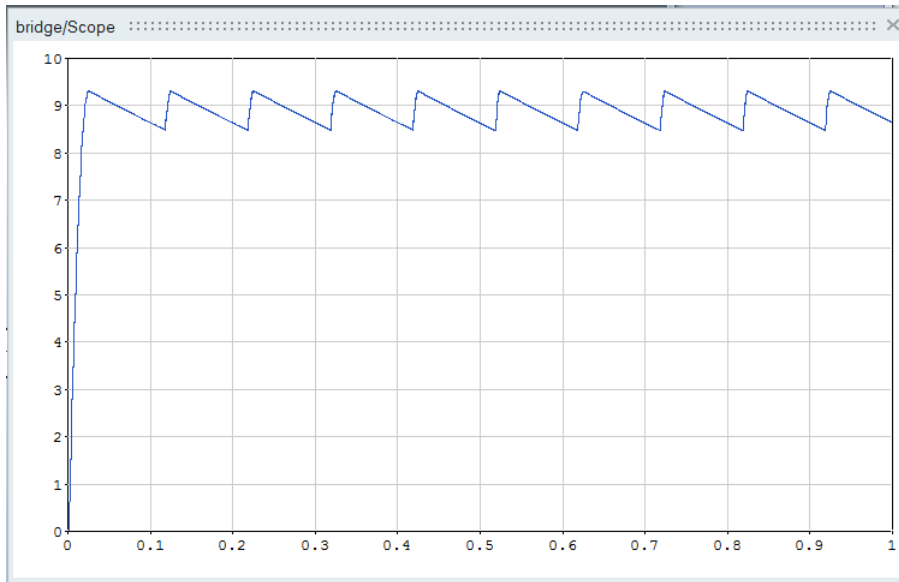
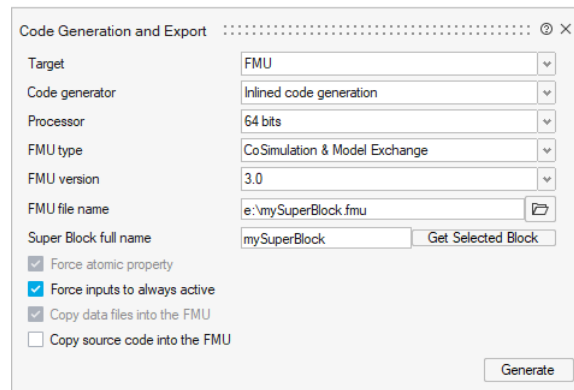Figure 12.23: Result in **Twin Activate**.

Figure 12.24: Code Generation and Export GUI in **Twin Activate**.

12.6.2 Exporting Modelica

A superblock containing Modelica components can be exported as FMU. Note that the superblock cannot have Modelica ports (implicit **Twin Activate** ports). All input and output ports should be of explicit signal type to be exported as FMU. Note that Modelica blocks can be exported in Standard code generation for FMU export, as well as in Inline code generator for Co-simulation. Inline Code Generator does not support FMI-Export with Model-Exchange interface type for Modelica blocks.

12.6.3 Shortcomings

Current version of the FMI export of **Twin Activate** does not support following features:

- Signal import blocks FromBase and SignalIn are not supported.
- Signal export blocks ToBase and SignalOut are not supported.
- Signal Viewer blocks such as Scope, Display, etc. are ignored.

- Co-simulation blocks such as MotionSolve or Flux blocks are not supported.
- Memory block is not supported.

12.6.4 Requirements

The FMI export facility needs a MicroSoft Visual Studio C++ compiler installed on the system. Otherwise the generated **C** code cannot be compiled and no binary will be available for the FMU and FMI export fails. Note that **Twin Activate** automatically detects the **C** compiler on the system. In order to detect the **C** compiler, you may use the command: `vssGetCompilerName()`

Note that the `tcc` compiler cannot be used in FMI export.

12.6.5 Example

In order to demonstrate the way a model in **Twin Activate** can be exported, here is an example. Open the model `Controller_Me_CS.scm` available in the product installation directory, under tutorial_models/Extended_Book_Models/Chapter_11_FMI folder. Suppose that we want to export the indicated region in the model, as shown in Fig.12.25.
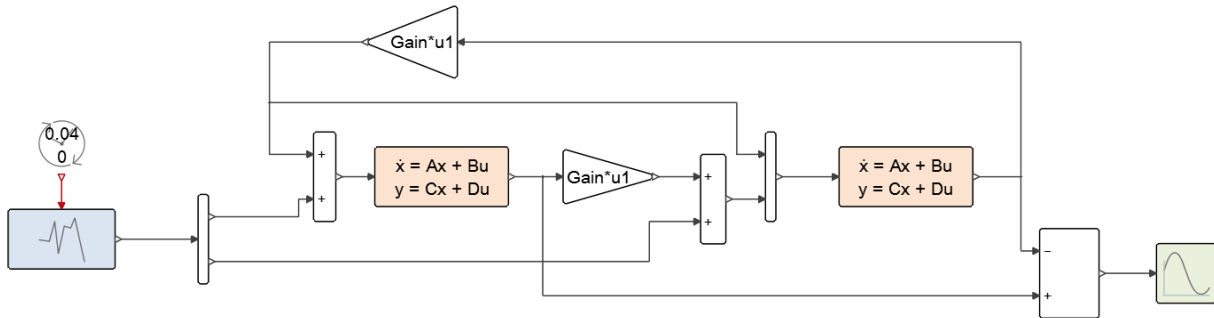

Figure 12.25: Controller model (AFM_controller.scm)

- Select the region and convert it into a super block as shown in Fig.12.26. Rename the super block as needed to the name of the FMU to be generated.
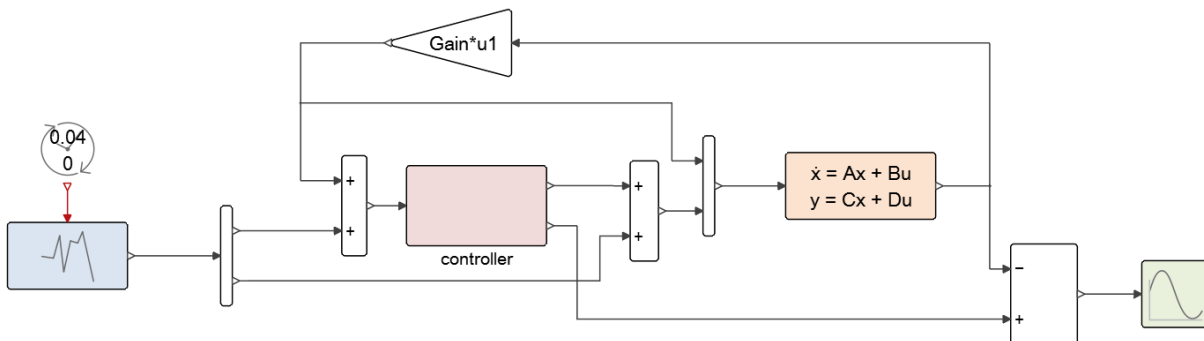

Figure 12.26: Controller model after superblock creation (SFM_controller.scm)

- If the superblock contains Input ports, inside the super block, click on the Input blocks and activate the "Time-Dependency" checkbox and fill the input dimensions with correct values (as shown in Fig.12.27). Otherwise an FMU with only discrete-time variables will be generated and the size of input ports will be one.

- Get outside of the super block and after selecting the super block, select the menu Tools/FMU 2.0 and choose the export type: Model Export, Co-Simulation, or both.

- The C code and the binaries are generated in the **Twin Activate** model temporary folder.

- Once the FMU is generated, a dialog window opens and asks the user to choose a folder to copy the FMU into.

- The generated FMU can be imported as an FMU with the "FMI import" block and be tested to verify if it gives the same result.
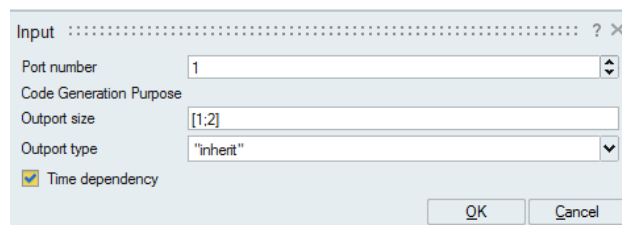
Figure 12.27: Input-port block dialog.

12.7 Advanced user topics:

The co-simulationalgorithm is not a part of the FMI standard and tools are free to choose the best method to perform the co-simulation with an FMU. The FMI import block is basically a **Twin Activate** block that interfaces an FMU (ModelExchange or CoSimulation) with **Twin Activate**. Similar to any other **Twin Activate** block, the FMI import block is called with several flags during the simulation. The model is updated through the propagation of block outputs throughout the whole model. In the output propagation, the input/output dependency of the blocks plays an important role. For example, if at least one output of a block depends directly on the ith input of the block, the block is not called (i.e., its outputs are not updated) unless i^{th} input is updated. This is guaranteed by the **Twin Activate** compiler to ensure that at any time instant and for a given continuous-time state, the outputs are correctly evaluated. Once all input and outputs are evaluated, other jobs, like reading derivative of continuous-time states, evaluating the zero-crossing surfaces, programming a discrete event, or in the case of co-simulationwith an FMU calling `fmixDoStep` can be done. For more details about the simulation flags and calling blocks the reader is referred to the chapter 9.

For the sake of simplicity, consider a **Twin Activate** model containing only one FMU, as shown in Figure12.28. The FMU has only one input and one output port. The whole **Twin Activate** part is abstracted into a single block. In co-simulation, the standard part of the **Twin Activate** model and the FMU exchange their input and output signals at some time instants called communication points (CP). Between two subsequent CPs **Twin Activate** simulator engine and FMU simulate separately with their respective solver. At the communication step t_i, the **Twin Activate** simulator engine takes the lead and performs the simulation until the next communication step, i.e., simulate the interval $[t_i, t_{i+1}]$. During the integration between t_i and t_{i+1}, the outputs of the FMU can be extrapolated. Mathematically speaking, the **Twin Activate** simulator engine performs the simulation of this model for the interval $t_i \leq t \leq t_{i+1}$,
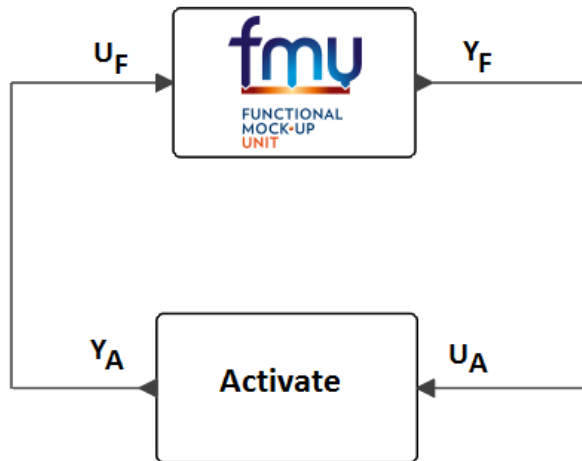
Figure 12.28: Simple co-simulationmodel

$$\begin{cases} \dot{x}_A &= f(x_A, u_A(t)) \\ u_A(t) &= y_F(t_i) + (t - t_i)y'_F(t_i) + \frac{(t-t_i)^2}{2}y''_F(t_i) + ... \end{cases} \qquad (12.1)$$

where x_A is **Twin Activate** continuous-time states and u_A is the input of the **Twin Activate** part. y_F, y'_F and y''_F are output of the FMU block, and their first and second time derivatives, respectively. If extrapolation is disabled (i.e., zero order extrapolation method is selected) then $u_A(t) = y_F(t_i)$.

During the integration of the **Twin Activate** part, the FMU may be invoked several times to retrieve $y_F(t_i)$, $y'_F(t_i)$, $y''_F(t_i)$, etc.

Once **Twin Activate** engine simulated the interval $[t_i, t_{i+1}]$, and $y_A(t_{i+1})$ is evaluated, it is the FMU turn to advance and simulate the same interval. Right before invoking `fmixDoStep`, the input values of the FMU at t_i (i.e., $u_F(t_i)$ and possibly its first order derivative (i.e., $der(U_F(t_i))$) are given to FMU (using `fmi2SetXXX` and `fmi2SetRealInputDerivatives` commands). Also, the value of $u_F(t_{i+1})$ is read and saved for subsequent `fmixDoStep` calls.

The derivative of input is computed as follows:

$$der(U_F(t_i)) = \frac{u_F(t_{i+1}) - u_F(t_i)}{t_{i+1} - t_i}$$

Once FMU inputs are set, the FMU is called to perform `fmixDoStep`(cp=t_i, h=$t_{i+1} - t_i$).

Controlling the co-simulationcommunication step

In the co-simulation with an FMU, the size of communication steps is governed by the numerical solver of **Twin Activate** (see the model settings panel). If a variable step-solver is chosen, the communication step will be variable and with a fixed-step solver, the communication steps will be fixed. In some cases, this behavior is not desired. Consider, for example, a complex **Twin Activate** model where a variable step-solver is needed but calling `fmixDoStep` in an FMU is costly. In this case, the user has the

possibility of choosing a fixed-communication step size in the FMU parameter settings (section 12.5.5)
.

Unlike the variable communication step size where every **Twin Activate** numerical solver step is followed by a `fmixDoStep` in the FMU, if a fixed-Communication step is chosen by the user, `fmixDoStep` is called periodically, regardless of **Twin Activate** Solver steps. Upon dependence of the output of the FMI block on the rest of the model, **Twin Activate** solver meshpoint may or may not coincide with the communication points of FMI. For example, if the output of the FMI block does not go to any block with continuous-time state, the solver mesh-point and communication points do not coincide.

Calling `fmixDoStep`

As described in section 12.4.1, **Twin Activate** and the FMU for co-simulation alternate between calling their respective solvers to solve and advance time between communication points and coupling shared data (inputs/outputs) at these communication points. At any particular time (t_i), **Twin Activate** will take a time step from t_i to t_{i+1} first and then call to the FMU to advance the time with `fmixDoStep`. **Twin Activate** calls `fmixDoStep` for either the flag `VssFlag_OutputUpdate` or `VssFlag_EventScheduling` depending on the model and FMU parameter settings.

Blocks in **Twin Activate** are called in a predefined order with flag `VssFlag_OutputUpdate`, in order to read the output ports. The simulator propagates the updated output port values all over the model. Once all input and output ports of blocks are updated, the blocks can be called (maybe in an arbitrary order) to do the update job with flag `VssFlag_EventScheduling` and `VssFlag_StateUpdate` (see the section 9.1.1 for more details).

In one case for calling DoStep in FMU for co-simulationblocks, after the input/output update phase at t_{i+1}, the input value of the FMU block at t_i (and possibly their derivatives) are given to the FMU. Then `fmixDoStep` is called in flag `VssFlag_EventScheduling` to advance the FMU time from t_i to t_{i+1} [5].

It is important to note that if `fmixDoStep` is called in flag `VssFlag_EventScheduling`, the output of the FMU are not immediately read because this is the purpose of the flag `VssFlag_OutputUpdate` which was called prior to `VssFlag_EventScheduling`. The outputs of the FMU will be updated in the next model input/output update with flag `VssFlag_OutputUpdate` .

This usually introduces a communication step delay in the output signal versus the input signal. In some situations, it is possible to perform `fmixDoStep` during the input/output update phase, i.e., with flag `VssFlag_OutputUpdate` . In this case, the newly updated output can be propagated through the model and the delay be avoided. The situations where `fmixDoStep` will be called in flag `VssFlag_OutputUpdate` are:

- If the FMU block does not have any input port, no need to wait all input/output update propagation to call `fmixDoStep`. At the block invocation to retrieve $u_A(t_{i+1})$ (which is equal to $y_F(t_i)$), `fmixDoStep` will be called and immediately $y_F(t_{i+1})$ can be obtained and propagated into the model. In this way, the delay is avoided.

- If the variability of all inputs is *discrete* or none of input data-types is *Real*, $u_F(t)$ will be equal to $u_F(t_i)$ all over the interval $[t_i, t_{i+1})$). As a result, `fmixDoStep` will be called at input/output update with flag `VssFlag_OutputUpdate` .

- If the input interpolation is off or the FMU does not support interpolation, $u_F(t)$ will be equal

---

[5]The flags and the job that the block should do in each flag has been explained in chapter 9

to $u_F(t_i)$ all over the interval $[t_i, t_{i+1}])$. As a result, `fmixDoStep` will be called at input/output update with flag `VssFlag_OutputUpdate` .

- Once all input values of FMU at t_{i+1} are read during the input/output update phase with flag `VssFlag_OutputUpdate`. As an example, consider the case where an output of the FMU depends directly on all input ports of the FMU. Once the value of the output port is requested by the simulator, it means that all inputs are already updated by the simulator at t_{i+1}. As a result, `fmixDoStep` can be immediately called and the FMU output $y_F(t_{i+1})$ can be propagated for the FMU outputs whose value are not already propagated.

Input interpolation and output extrapolation in co-simulation

Twin Activate is a hybrid simulation tool which means it can simulate a model with continuous-time and discrete-time parts. The discrete-time part is activated by discrete-time events and the continuous-time part of the model is integrated by the numerical solvers of **Twin Activate**. Discrete-time events cause discontinuities in signals. As a result, signals in a hybrid model may be piece-wise continuous. Since several discrete-time events may happen at the same time, a signal may have several values at the same time instants, as shown in Fig.12.29.
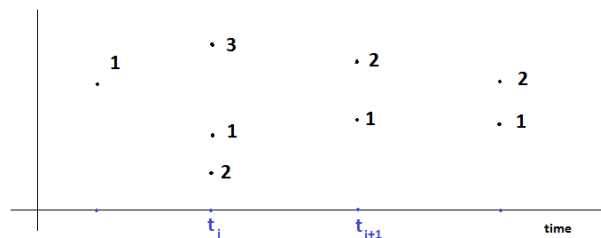

Figure 12.29: evaluation of a model at different time instants

The multiple time instants at an event time is also called super dense time instants. During the integration of the continuous-time part, the numerical solver takes steps to advance the time. At the beginning of each step, the time dependent part of the model are evaluated with the flags `VssFlag_OutputUpdate`, `VssFlag_EventScheduling` and `VssFlag_StateUpdate` . The time instant where the model is evaluated at the beginning of the solver step is also called a solver 'mesh-points'. Discrete-time events are classified in two categories, critical and non-critical events. Critical events are discrete-time events that may cause discontinuities in the continuous-time part of the model. If a critical event happens, the numerical solver needs to be cold-restarted[6]. Following one or several critical events, there is always a solver mesh-point to update the continuous-time parts. In other words, after the evaluation of a model at critical events, the continuous-time part should be evaluated before starting the integration of the continuous-time part. Unlike critical events, after a non-critical events, there is not necessarily a mesh-point. Non-critical events may happen between two mesh-points.

If the *"Preferred fixed communication stepsize"* in the co-simulation tab of the FMU block, is left zero, then the **Twin Activate** solver mesh-points are taken as the co-simulation communication points, i.e., in Fig.12.30 t_i and t_{i+1} are **Twin Activate** mesh-points and also communication points. Since the model may have critical events, the model may get evaluated several times at communication instants. Before calling `fmixDoStep` at the mesh-point t_{i+1}, the input of the FMU at the mesh-point (i.e., $u_F(t_i)$) is

---

[6]For more details about events, and solver, interested readers are referred to the chapter on solvers.

given to the FMU. If the derivative of the input is also provided to the FMU, **Twin Activate** computes the first order derivative of the input by using the following equation.

$$der(u_F(t_i)) = \frac{u_F(t_{i+1}) - u_F(t_i)}{t_{i+1} - t_i}$$

Note that in order to compute the derivative of the input, the input value at the mesh-point evaluation, as well as the first super dense time at the next communication points is used, see Fig.12.30.
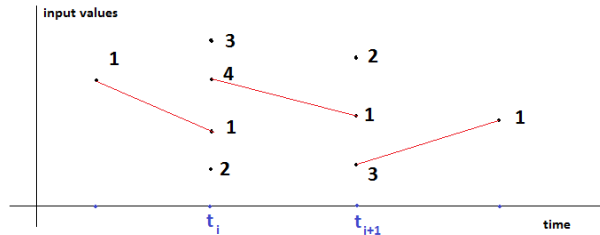

Figure 12.30: First order interpolation of FMU inputs in presence of events at communication steps.

If the *"Preferred fixed communication stepsize"* in the co-simulation tab of the FMU block is non-zero, then the communication points are periodic. i.e., in Fig.12.31, t_i and t_{i+1} are communication points. Between communication points, there may be several **Twin Activate** mesh-points, see Fig.12.31. The FMU is activated by a periodic events where at each event time, `fmixDoStep` is called. In this case, before calling `fmixDoStep` at the event time of t_{i+1}, the input of the FMU at the previous communication point (i.e., $u_F(t_i)$) is given to the FMU.
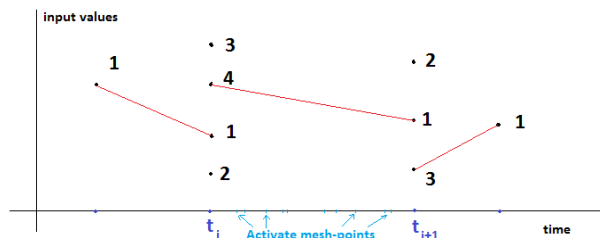

Figure 12.31: First order interpolation of FMU inputs in presence of events at communication steps (fixed communication step).

Based on FMU attributes, **Twin Activate** chooses to call `fmixDoStep` at t_{i+1} either during the call with flag `VssFlag_OutputUpdate`, or during the call with flag `VssFlag_EventScheduling`. When the **Twin Activate** solver integrates the interval $[t_i, t_{i+1}]$, the output of the FMU block is computed with the following formula (if a linear output extrapolation is used).

$$y_F(t) = y_F(t_i) + (t - t_i)y'_F(t_i)$$

At t_{i+1}, the output of FMU is $y_F(t^1_{i+1}) = y_F(t_i) + (t_{i+1} - t_i)y'_F(t_i)$ until `fmixDoStep` is called at the mesh-point and the output of FMU is retrieved. The output is retried and displayed at the next communication point.
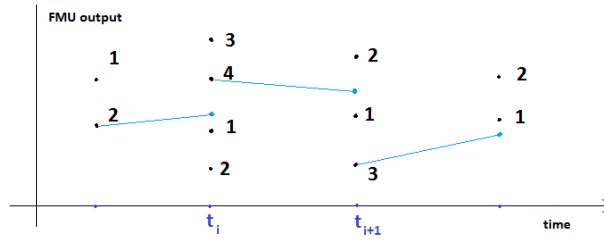
Figure 12.32: First order extrapolation of FMU output in presence of events at communication steps.

Bibliography

[1] Functional Mock-up Interface for Model Exchange. Version-1.0, 2010.

[2] Functional Mock-up Interface for Co-Simulation. Version-1.0, October 2010.

[3] Functional Mock-up Interface for ModelExchange and Co-Simulation, Version-2.0, 2014.

[4] Proposal for a Standard Time Series File Format in HDF5, Andreas Pfeiffer,Ingrid Bausch-Gall, Martin Otter, Linköping Electronic Conference Proceedings, 2012.

[5] A Simulation Environment for Efficiently Mixing Signal Blocks and Modelica Components,Ramine Nikoukhah,Masoud Najafi,Fady Nassif, Modelica Conference, Prague, 2017

Chapter 13

Co-Simulation with Multi-body Simulation

13.1 Introduction

Modeling complex heterogeneous systems in engineering usually leads to hybrid systems of differential and algebraic system of equations with discrete-time equations. Such complex multi-disciplinary systems cannot often be modeled and simulated in one simulation tool alone. Subsystem models are often available only for a specific simulation tool. For example special tools for CFD (Computational Fluid Dynamics) models or integrated circuit systems. In many situations, the sub-systems shall be simulated with the simulator which suits best for the specific domain. Thus for the simulation of multi-disciplinary models it is often reasonable or even necessary to couple different simulation tools with each other or with real world system components. Co-simulation is a general approach for the joint simulation of models developed with different tools where each tool treats one part of a modular coupled problem. The data exchange (input and output variables, status information) between subsystems is restricted to communication points. In the time between two communication points, the subsystems are solved independently from each other by their individual solvers. If intermediate input or output are needed by one of simulators, it is very often computed through interpolation or extrapolation.

Main-Secondary is a common method in Co-simulation. In a main-secondary approach the secondary solver simulates the sub-model whereas the main solver is responsible for both coordinating the overall simulation as well as transferring data. The secondary solvers are the simulation tools, which are prepared to simulate their subtask. The secondary solvers are able to communicate data, execute control commands and return status information. Several MSplant blocks representing different **Altair MotionSolve** models can be instantiated in a **Twin Activate** model.

13.2 Co-Simulation with MBS

The Co-simulation interface lets the user simulate a complex system that includes one or more multi-body systems (defined by **MotionSolve** blocks) and several subsystems modeled with **Twin Activate** blocks. In order to effectively simulate the entire system, the MBS subsystems are simulated with **MotionSolve** models [1] while the control subsystem is simulated with **Twin Activate**.

---

[1]The installation of Altair **MotionSolve** ™ software is required for co-simulation. **MotionSolve** is a multi-body modeling, analysis, visualization and optimization solution for performing multi-disciplinary simulations that include kinematics and dynamics, statics and quasi-statics, linear and vibration studies, stress and durability, loads extraction, co-simulation, effort estimation and packaging synthesis.

In the **Twin Activate**-**MotionSolve** interface, **Twin Activate** is the main solver and **MotionSolve** (MS) is secondary solver. The co-simulation interface has been implemented in a block called **MS Plant** which is available in the `Advanced/CoSimulation` palette, as shown in Fig.13.1. **MS Signals** is a variant of **MS Plant** block that displays the name of MS signals at the input/output ports of the blocks.
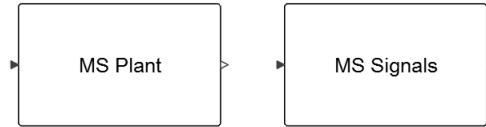

Figure 13.1: Blocks in **Twin Activate** to simulate a **MotionSolve** model.

The block provides a small set of C++ functions to implement the **Twin Activate**-**MotionSolve** interface. The path of the **MotionSolve** model is given to MS Plant and **MotionSolve** solver is called to simulate the model. In order to perform the simulation, it is necessary to have **MotionSolve** installed on the machine. This process is shown in fig.13.2.
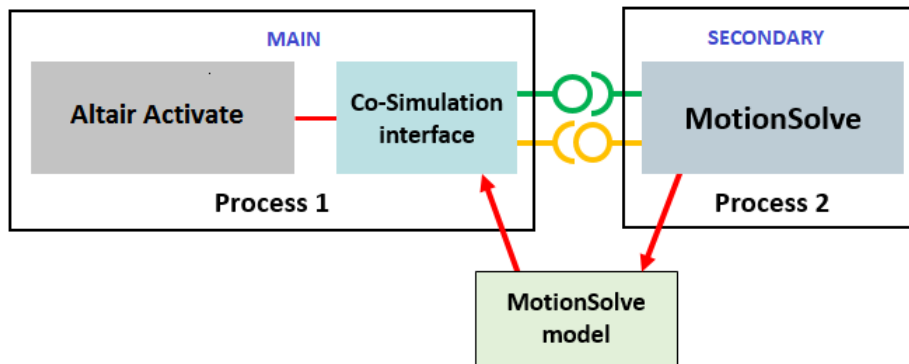

Figure 13.2: Co-simulation between **Twin Activate** and **MotionSolve**.

The co-simulation connection variables to define the inputs/outputs from the **MotionSolve** model are defined within the Control_PlantInput and Control_PlantOutput **MotionSolve** statements, which is a list of runtime (solver) variables for inputs and outputs to the mechanical plant, respectively. These also have settings for the hold_order, sampling_period, and offset_time.

In co-simulation, after instantiation and initialization of the **MotionSolve** solver, inputs to the **Motion-Solve** model is provided and the output of the **MotionSolve** block is requested. In order to explain the way a co-simulation is performed, consider a model composed of two simulators as shown in Fig.13.3.
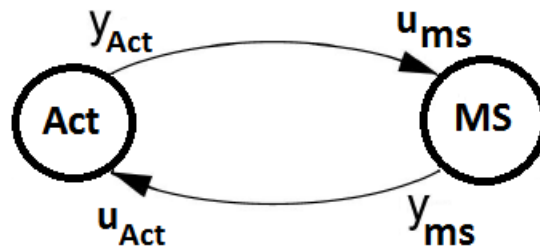

Figure 13.3: Co-simulation between **Twin Activate** and **MotionSolve**

If the output of each model is computed as following equations:

$$\begin{cases} y_{act} & = F_{act}(y_{ms}) \\ y_{ms} & = F_{ms}(y_{act}) \end{cases}$$

in order to co-simulate this system, this equation should be solved at each communication instant. F_i functions are usually complex, non-linear, and time-dependent relationships, as a result, iterative methods are the only choice for solving such equations. Three common methods used for solving them are Gauss-Seidel and Gauss-Jacobi and Newton methods. The Gauss-Seidel iteration method is represented as follows.

$$y_{act}^{i+1} = F_{act}(y_{ms}^i)$$

$$y_{ms}^{i+1} = F_{ms}(y_{act}^{i+1})$$

$$i = 0, 1, 2, ...$$

The Gauss-Jacobi iteration method is defined as

$$y_{act}^{i+1} = F_{act}(y_{ms}^i)$$

$$y_{ms}^{i+1} = F_{ms}(y_{act}^i)$$

$$i = 0, 1, 2, ...$$

These iterative methods are simple fixed-point methods and their convergence(at most linear) depends on F_i functions. If the Fixed Point converges very slowly or diverge the Newton-type methods can be used. These methods are, however, more complicated, because in each step the Jacobian must be computed which results in higher number of simulator calls. **Twin Activate** does not support roll back in time, as result, the method which is used actually in **Twin Activate** is GS1, i.e., the Gauss-Seidel method with only one iteration. In this CoSimulation scheme there is no error control.

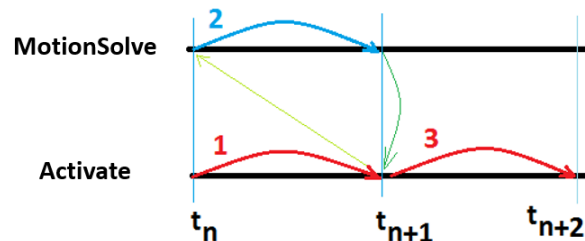This method has been illustrated in Fig.13.4;

Figure 13.4: CoSimulation method used in **Twin Activate**.

The **Twin Activate** simulator performs the following steps for the co-simulation and synchronization between secondary solvers, as shown in 13.4.

1. The **Twin Activate** simulator advances the time from t_n to t_{n+1}, and updates the output of all blocks (except secondary solvers).

2. Secondary solvers are called one by one and asked to advance their time from t_n to t_{n+1} and their output is requested at t_{n+1}.

3. The **Twin Activate** simulator takes a new step from t_{n+1} to t_{n+2} using the new output value of secondary solvers computed at t_{n+1}.

4. Goto 1

Usually in order to get a correct result, the communication step-size (time between two communication instants) should not be larger than the smallest time-constant of secondary solvers. Often smaller communication step-size results in better convergence but longer simulation time.

During the CoSimulation, **Twin Activate** is the lead solver and takes the lead and advances the time, as shown in 13.4. Once **Twin Activate** took its step and reached t_{n+1}, then **MotionSolve** solver which is the lag solver takes its own internal steps to reach at least t_{n+1}.

13.2.1   Under the hood

The CoSimulation method shown in 13.4 is a rough representation of the CoSimulation method between **Twin Activate** and **MotionSolve**. Actually, on the **MotionSolve** side there are two buffers for keeping input and output values exchanged at communication points, t_n, t_{n+1}, etc. , as shown in 13.5.
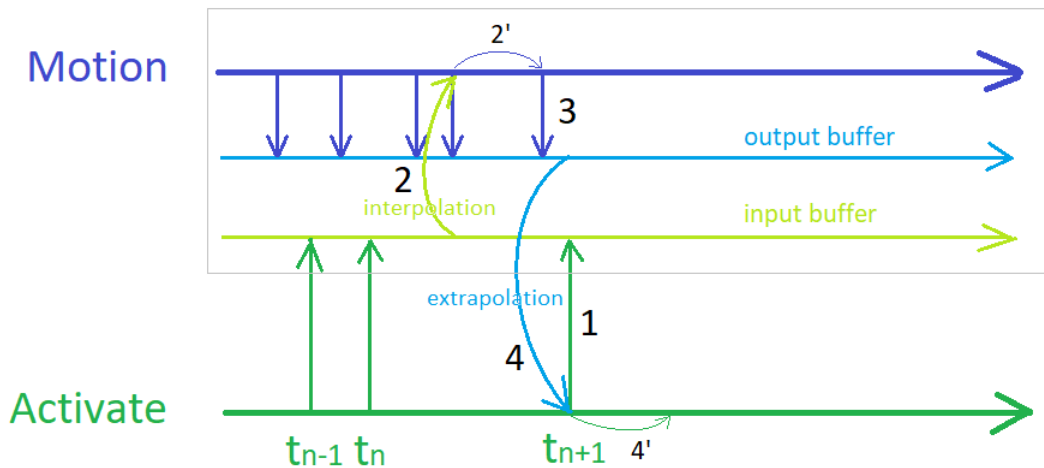

Figure 13.5: Input and output buffers in **MotionSolve**.

1. **Input-buffer:** At communication points t_n, **Twin Activate** deposits the inputs to **MotionSolve** into this buffer whenever **Twin Activate** calls the **MotionSolve** block. The deposit time instants are completely independent of **MotionSolve** internal steps. **MotionSolve** solver can query this buffer to get the data it needs at a specific time. Since the **MotionSolve** solver is the lag solver, the input buffer will always interpolate the data deposited by **Twin Activate** to provide the input to the **MotionSolve** solver. The interpolate order is 2.

2. **Output-buffer: MotionSolve** deposits the outputs from **MotionSolve** into this buffer at the frequency it is supposed to. The deposit time instants are completely independent of **Twin Activate** internal steps or communication points. **Twin Activate** solver can query this buffer to get the data it needs at a specific time. Since **Twin Activate** is the lead solver, the output buffer will extrapolate at communication points $t_n < t \leq t_{n+1}$ to provide the output to the **Twin Activate** solver. The order of extrapolation is defined in the **MotionSolve** xml model, as will be explained as follows.

The writing into and reading from these buffers follows three basic rules in **MotionSolve**:

1. At each step for any solver, either **Twin Activate** or **MotionSolve**, makes an attempt to deposit data into its corresponding output buffer, i.e., **output-buffer** for **MotionSolve** and **input-buffer** for **Twin Activate**, and the output buffer immediately returns the control back to the solver. The data is accepted if the step is a successful step, and rejected if it's not. Then, the solver will request data from its input buffer.

2. When input buffer gets request from a solver, the **MotionSolve** solver will need to wait if there is not enough data for interpolation. The **Twin Activate** solver has to wait if the request time is more than one step (the **Twin Activate** solver's step) larger than the last time the **MotionSolve** solver made an attempt to deposit or request data to the same buffer.

3. A buffer-in-use lock should be enforced to prevent buffer from being overwritten (by new deposit) while the same buffer is preparing and returning request for another solver.

In general, one should choose **Twin Activate** solver to be the solver with smaller stepsize and **MotionSolve** to be the solver with larger stepsize. This is because **Twin Activate** solver usually gets extrapolated inputs and **MotionSolve** always gets interpolated inputs, and extrapolating by a large step usually leads to worsen results. Also, the inputs to the **Twin Activate** solver should be something less prone to noise, like displacements. The inputs to the **MotionSolve** solver, on the other hand, can be less restrictive including noisier signals such as forces or torques.

When the **Twin Activate** solver requests the output-buffer the output at t_n, the **MotionSolve** solver cannot integrate beyond t_n. Actually, whether **MotionSolve** solver can integrate beyond t_n or not is not determined by when the **Twin Activate** solver requests the output-buffer for a data. It is determined by when the last time **Twin Activate** solver deposits a data at the input-buffer. Since, this last time is usually t_n, **MotionSolve** solver cannot integrate beyond this point because the input-buffer will keep the **MotionSolve** solver waiting until there is enough data for interpolation (rule #2).

These two buffers are not visible from **Twin Activate**'s side and filled and queried only by **MotionSolve**. **Twin Activate** communicates with **MotionSolve** only at communication points t_n. When outputs of the **MotionSolve** block are computed at communication points, **MotionSolve** extrapolates the outputs. The order of extrapolation, is defined by the parameter *HOLD_ORDER* defined in the **MotionSolve** or MotionView model. This parameter which defines the degree of the polynomial used for extrapolation takes the following values:

1. *HOLD_ORDER = 0* means the last value is returned.

2. *HOLD_ORDER = 1* means a straight line is fitted through the last 2 points. The line is extended to the new time to return the signal value.

3. *HOLD_ORDER = 2* means a parabola is fitted through the last 3 points. The parabola is extended to the new time to return the signal value.

4. *HOLD_ORDER > 2* is not supported, because extrapolation is inherently dangerous and higher order extrapolations can lead to large (and unexpected) values.

The decoupling of the solvers through the buffer mechanism allows each solver to pick its own step. In other words, **Twin Activate** and **MotionSolve** solver settings do not directly affect each other's time steps. This leads to a faster simulation. The only way the simulators can affect each other's time step is through the model. If the **MotionSolve** output is smooth enough, that will allow **Twin Activate**'s numerical solver take larger step sizes. On the other hand, if in the **MotionSolve** side, a lower order extrapolation order is chosen or the **MotionSolve** output shows noisy or high frequency behavior, the **Twin Activate** numerical solver needs to choose smaller steps to reduce the error. When **MotionSolve**

output is noisy, data got from 2nd order extrapolation at larger time step will be way off. This is the reason that user needs to choose order 1 or 0. So, in the **MotionSolve** xml, a lower extrapolation order needs to be chosen when the **MotionSolve** output shows noisy or high frequency behavior, otherwise the **Twin Activate** numerical solver needs to choose smaller steps to reduce the error.

Interested users are referred to the **MotionSolve** Users Guide for more information.

13.3   MSplant block parameters

The MSplant block implements co-simulation interface between **Twin Activate** and **MotionSolve**. Simulations with this block require that Altair **MotionSolve** solver is installed. The path to the solver and its licensing must be set within **Twin Activate** preferences panel, section Co-Simulation.
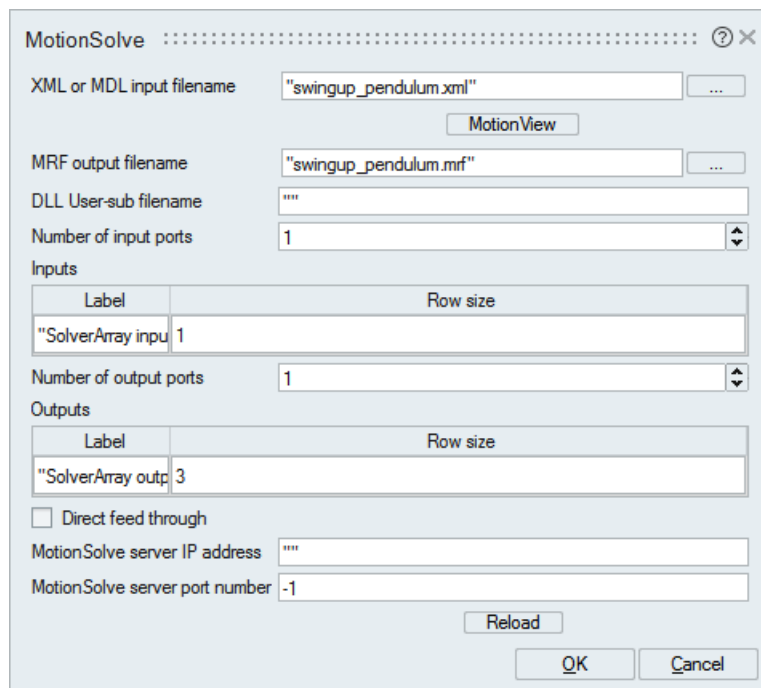
Figure 13.6: Graphical user interface of the MSplant block.

- **XML or MDL input filename:** In this field the name of the **MotionSolve** model (with extension .xml) or MotionView (with extension .mdl) can be given. The filenames can be either given by an absolute path, or a relative path. If **Twin Activate** and **MotionSolve** models are in the same folder, no path is required. In the later case the absolute path is obtained from where the **Twin Activate** model is located. If a MotionView is given the (.mdl) file is automatically converted into a **MotionSolve** (.xml) file.

- **Launch MotionView:** If a MotionView (.mdl) file is provided as the input **MotionSolve** filename, it is possible to visualize and edit the model in MotionView by clicking on this button. Note: When MotionView is open **Twin Activate** remains inactive.

- **MRF output filename:** This field indicates the name of the MotionView result filename. Note that the folder where is file should be generated is not write-protected. If the corresponding feature is active in the **MotionSolve** (.xml) model, the MotionView result file (.mrf) will be converted into an H3D file (.h3d). The .mrf and .h3d files can be visualized in MotionView.

- **DLL User-sub filename (optional):** If the **MotionSolve** model requires extra libraries for execution, the path of these DLLs can be provided in this field. The DLLs provided here will be loaded by **MotionSolve** at the beginning of the co-simulation and will be released at the end.

- **Number of input ports:** The number of input ports of the block. This number must correspond to the number of plant inputs in the **MotionSolve** model. This field is automatically filled if the Reload button is pressed.

- **Inputs:** Each row corresponds to an input port. The label of input as well as the size of the input should be provided here. These fields are automatically filled if the Reload button is pressed.

- **Number of output ports:** The number of output ports of the block. This number must correspond to the number of plant outputs in the **MotionSolve** model. This field is automatically filled if the Reload button is pressed.

- **Outputs:** Each row corresponds to an output port. The label of output as well as the size of the output should be provided here. These fields are automatically filled if the Reload button is pressed.

- **Direct feedthrough:** The **MotionSolve** model does not indicate if an input of the MSplant block directly affects the outputs (feedthrough information). This check box allows setting the feedthrough value. If this check-box is active, it is ensured that the input of the block is always up-to-date when the output of the block is requested. Otherwise a very short delay may be observed in the output of the block.

- **Reload button:** The information about the input and output of the **MotionSolve** model and their names and their sizes are all available in the **MotionSolve** (.xml) file. Instead of filling the above fields manually, the user can fill them automatically by pressing this button. Filling manually the fields is useful when the xml file is not still available and the user needs to build the **Twin Activate** model.

- **MotionSolve server IP address:** This field is not currently used.

- **MotionSolve server port number:** This field is not currently used.

13.4   Example: Pendulum Swing-Up

The inverted pendulum is a popular system for illustrating control and multi-body problems. In [1], a control strategy for swinging up a pendulum from downward position to the upright position has been presented. This strategy has been explained and implemented in 17.3.1 where the pendulum and the controller are implemented in **Twin Activate**. Here we will explain the way the pendulum is implemented in **MotionSolve** and the controller in **Twin Activate**. The resulting models are then co-simulated to bring the pendulum in upward position.

The inverted pendulum has been implemented in **MotionSolve** as shown in Fig.13.7. This model is available as a **Twin Activate** model in the `Main_Demos` folder. The model has one input signal `sa_in[1]` of size one and one output vector signal `sa_out[3]` of size three, as shown inFig.13.8. The force applied to the pendulum cart's translational y-axis the input signal to the MS' model (`force.y` (u)), as shown in Fig.13.9.

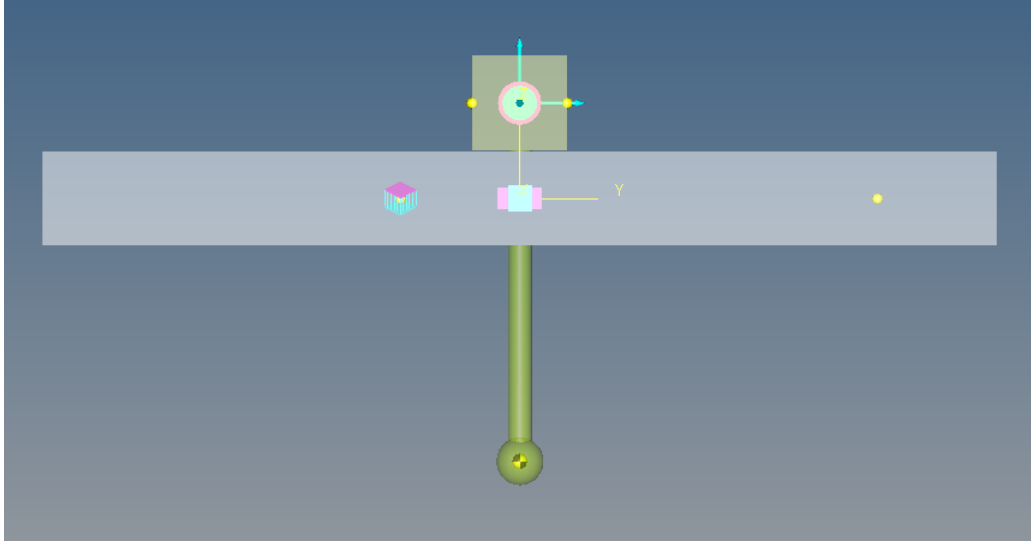The output signals used which go to **Twin Activate** are

- `angle_ball_globalz`(θ),

Figure 13.7: Model of inverted pendulum on a cart realized in **MotionSolve**.

- omega_ball_globalz(ω),
- cart.y(y)

as shown in Fig..

The controller is much easier to implement in **Twin Activate**, because it is a more adapted tool for building complex controllers. The controller which receives the angle and the angular velocity of the pendulum has been implemented in **Twin Activate**, as shown in Fig.13.13. The block parameters of the MSplant block is shown in Fig.13.12. The co-simulation result is shown in Fig. 13.14. The evolution of pendulum angle and angular velocity (θ and ω) are given in the first and the second subplots, respectively. The third subplot is the cart displacement which is not controlled. The fourth subplot is the input signal to the MSplant block (applied axial force to the pendulum).
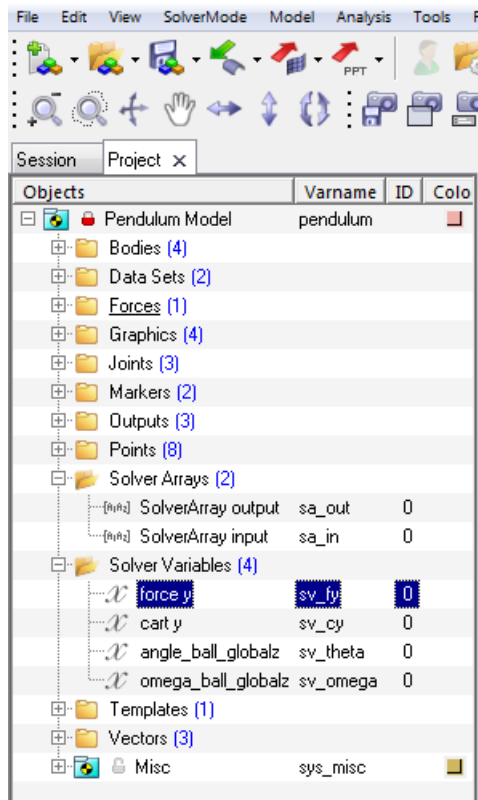
Figure 13.8: Input and output variables of the **MotionSolve** pendulum model.
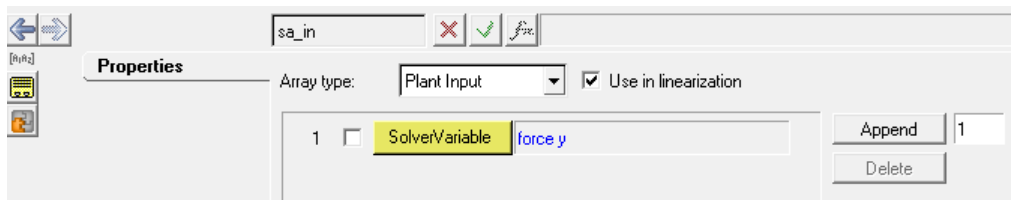

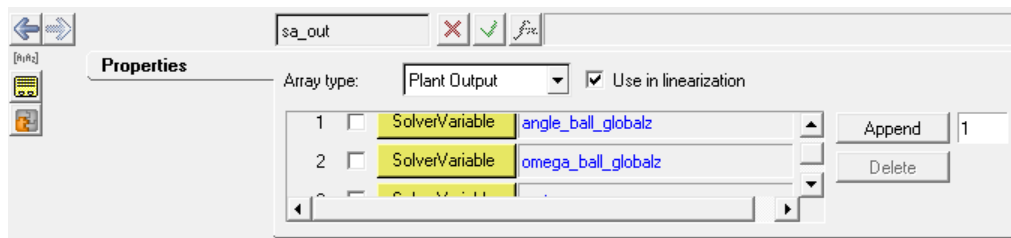Figure 13.9: Input variable of the **MotionSolve** pendulum model.

Figure 13.10: Output variables of the **MotionSolve** pendulum model.
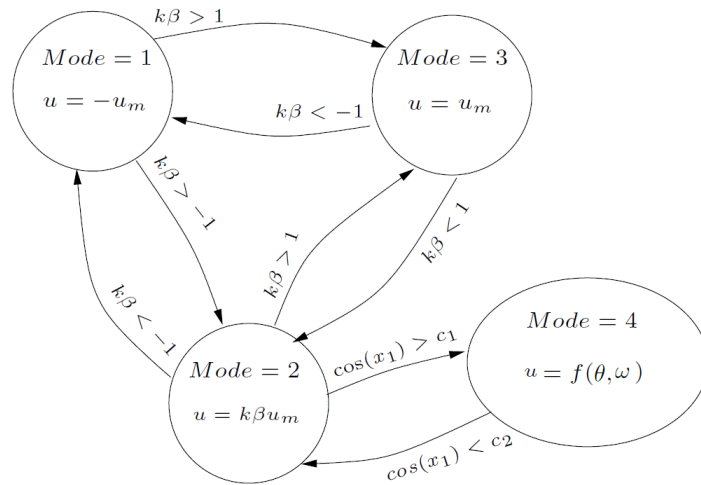
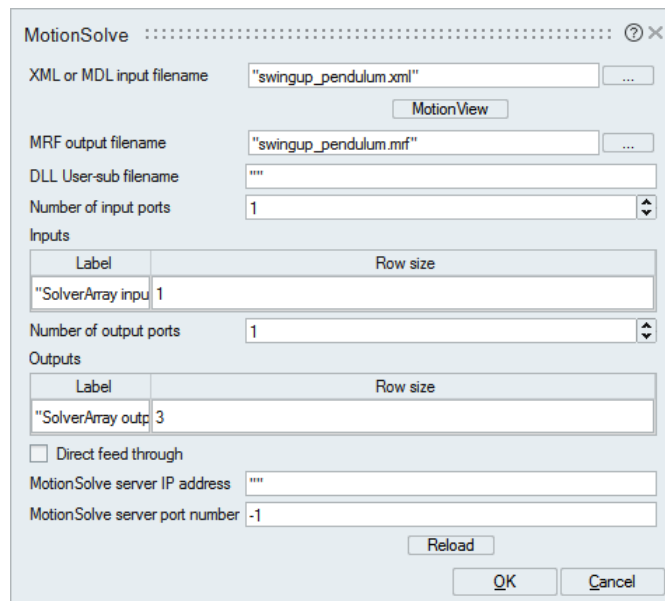Figure 13.11: Control strategy to swing up and stabilize an inverted pendulum on a cart.

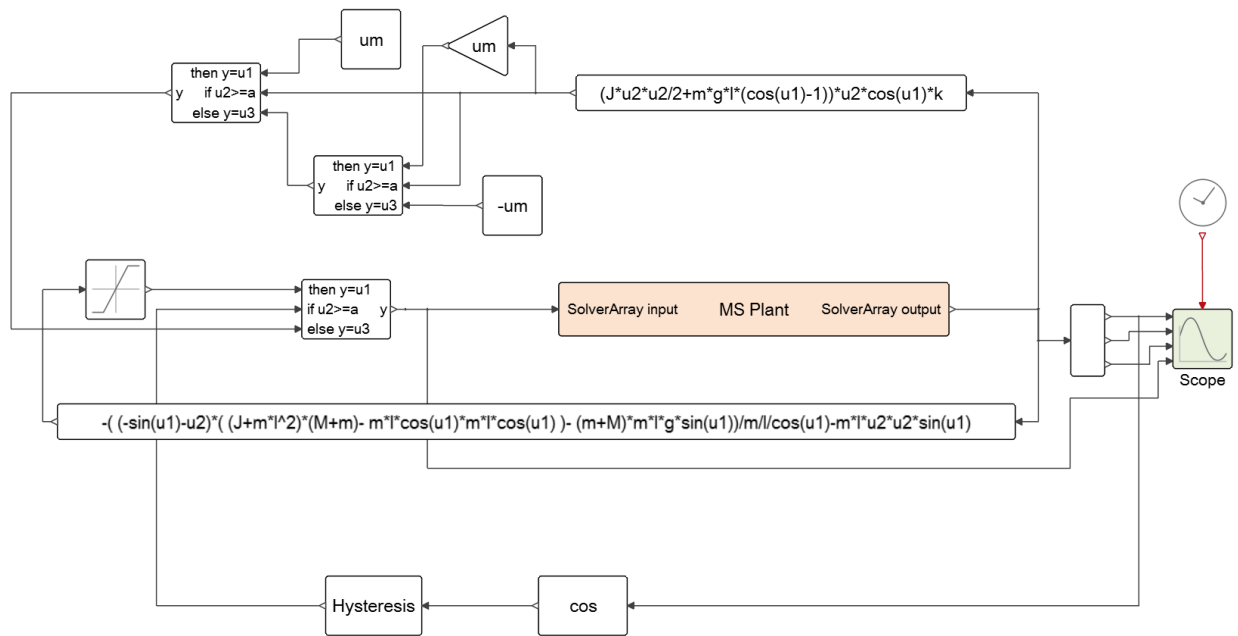Figure 13.12: Graphical user interface of the MSplant block.

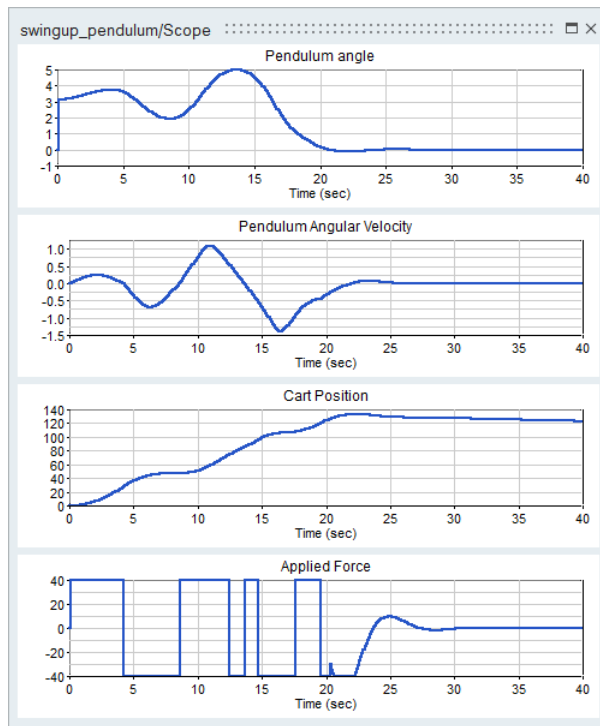Figure 13.13: The controller built in **Twin Activate**.

Figure 13.14: The co-simulation result displayed in **Twin Activate**.

Bibliography

[1] Swinging up a pendulum by energy control, K. J. Astrom and K. Furuta, Automatica, Volume 36, pp 278-285, 2000.

Chapter 14

CoSimulation with Electromagnetic and Thermal Models

14.1   Introduction on Flux

Modeling and simulation of complex electromagnetic systems in engineering usually leads to hybrid systems of differential and algebraic system of equations with discrete-time equations. Such complex systems cannot often be modelled and simulated in one simulation tool. The system and control part is simulated in **Twin Activate** and the electromagnetic and thermal parts are simulated in **Altair Flux**. **Flux** is a finite element software for low-frequency electromagnetic and thermal simulations. In order to simulate such multi-disciplinary models, CoSimulation is the principal solution. CoSimulation is a general approach for the joint simulation of models developed with different tools where each tool treats one part of a coupled problem. The data exchange (input and output variables, status information) between subsystems is restricted to communication points. In the time between two communication points, the subsystems are solved independently from each other by their individual solvers.

Twin Activate provides an interface to **Flux** which allows reading sensor variables such as current, position, and rotor speed defined in the **Flux** model and applying control variables computed by **Twin Activate** such rotor angle position.

In this chapter, the **Twin Activate**-**Flux** interface and the **Flux** block for CoSimulation will be discussed[1].

14.2   CoSimulation with Flux

Similar to the **Twin Activate**-**MotionSolve** CoSimulation interface (13.1), a main-secondary method is used to co-simulate between **Twin Activate** and **Flux**. In a main-secondary approach the secondary solver simulates the sub-model whereas the main solver is responsible for both coordinating the overall simulation as well as transferring data. The secondary solvers are the simulation tools, which are prepared to simulate their subtask. The secondary solvers are able to communicate data, execute control commands and return status information. Several **Flux** blocks representing different **Flux** models can be used in a **Twin Activate** model.

The CoSimulation interface lets the user simulate a complex system that includes one or more elec-

---

[1]For more information of on **Flux**, interested users are referred to the **Flux** User's Guide.

tromagnetic and thermal systems (defined by **Flux** blocks) and several subsystems modeled with **Twin Activate** blocks.

In the **Twin Activate-Flux** interface, **Twin Activate** is the main solver and **Flux** is secondary solver. The CoSimulation interface has been implemented in a block called **Flux** which is available in the `Activate/CoSimulation` palette, as shown in Figure 14.1.

Figure 14.1: **Flux** block in **Twin Activate**.

In order to perform the simulation, it is necessary to have **Flux** installed on the machine. The installation path of **Flux** should be provided to **Twin Activate** in the `Preferences` of **Twin Activate**, as shown in Figure 14.2.
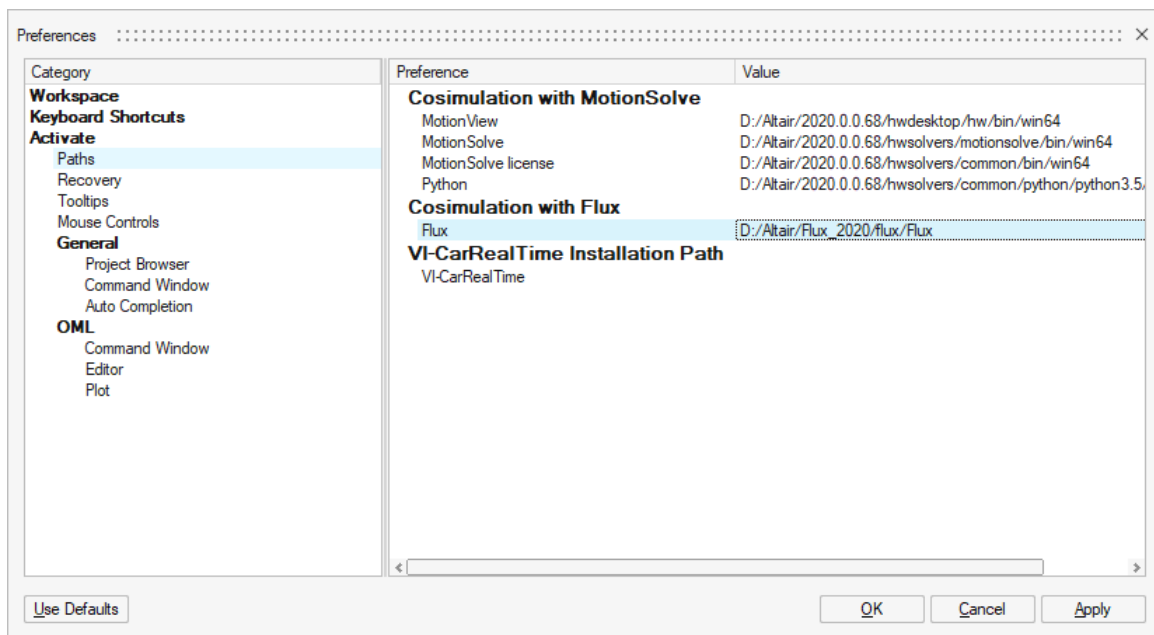

Figure 14.2: Defining the **Flux** installation path in **Twin Activate**.

In **Flux**, once a project is created, the interface variables, i.e., input and output to **Twin Activate** are defined. Then a CoSimulation interface is exported as an `*.f2sta` component[2]. The `.f2sta` component contains the **Flux** project information including the **Flux** version, the type of **Flux** solver, and the name of input and output ports. The information inside the `*.f2sta` file is loaded by **Twin Activate** to setup the CoSimulation.

In order to explain the way a CoSimulation is performed, consider a model composed of two simulators as shown in Figure 14.3.

---

[2]The reader is referred to the **Flux** User's Gide for more details on exporting `.f2sta` components.
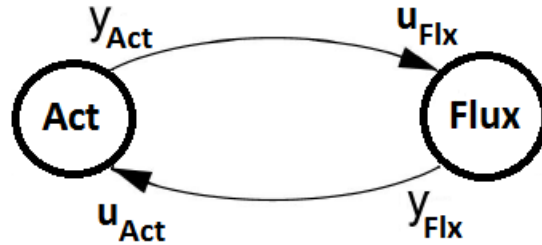
Figure 14.3: CoSimulation between **Twin Activate** and **Flux**.

During the CoSimulation, after instantiation and initialization of the **Flux** solver, inputs to the **Flux** model is provided and the output of the **Flux** block is requested. Suppose that the output of each part is computed with following equations:

$$\begin{cases} Y_{Act} & = F_{Act}(Y_{Flx}) \\ Y_{Flx} & = G_{Flx}(Y_{Act}) \end{cases}$$

In order to co-simulate this system, this equation should be solved at each communication instant. F_{Act} and G_{Flx} functions are usually complex, non-linear, and time-dependent relationships, as a result, iterative methods are the only choice for solving such equations.

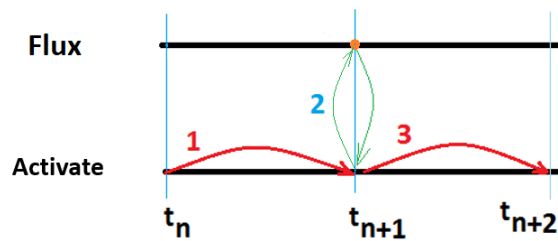This method has been illustrated in Figure 14.4;

Figure 14.4: CoSimulation method used in **Twin Activate**.

The **Twin Activate** simulator is the main solver and advances the time. The following steps for the CoSimulation and time synchronization between **Twin Activate** and **Flux** is done, as shown in Figure 14.4.

1. The **Twin Activate** simulator updates the input of the **Flux** block at instant t_n and requests the **Flux** output at t_n.

2. The **Flux** block reads its inputs and performs either a complete computation or a simple extrapolation to compute the output at t_n.

3. The **Twin Activate** simulator advances the time to t_{n+1} and repeats.

Usually in order to get a correct result, the communication step-size (time between two communication instants) should not be larger than the smallest time-constant of secondary solvers. Often smaller communication step-size results in better convergence but longer simulation time.

The CoSimulation method shown in 14.4 is a rough representation of the CoSimulation method between **Twin Activate** and **Flux**. There are subtle details about the way **Flux** block is called during the

CoSimulation. Actually, each time the **Flux** engine performs a computation, the whole finite element computation is done in **Flux** with the recent input received from **Twin Activate**. But, each **Flux** engine computation is very costly in time. Then, it is crucial to reduce the number of computation of **Flux** to reduce the CoSimulation time.

In order to reduce the simulation time there are two solutions. The first way is done inside the **Flux** engine, i.e., the **Flux** engine does not perform the computation, if it is not necessary. In fact, a simple intuitive method to reduce the number of **Flux** computation is to not performing the computation inside **Flux**, if input values of the **Flux** block, provided by **Twin Activate** have not changed within a range. In this way, a parameter in the **Flux** block GUI is used to define the percentage in input variations. The parameter **'Minimal input variation'** specifies the percentage of input variation between two consecutive calculations of **Flux**. If the **Flux** engine performs a computation at time t_n, the next **Flux** engine computation at t_{n+1} will be carried out only if the change in inputs exceeds the designated value. If the **Flux** engine calculation is not carried out, **Flux** block outputs are extrapolated based on the 'Output extrapolation order' value.

This method is useful when the signal incoming from **Twin Activate** is almost steady or does not change very fast with respect to **Twin Activate** signals. Note that this solution may become troublesome, if the **Flux** component contains time-dependent elements with fast time constants. Suppose there is a time dependent **Flux** model whose inputs provided by **Twin Activate** are Constant values. If the **'Minimal input variation'** parameter is specified, since the block inputs does not change during the simulation, the **Flux** engine will perform the computation only once, which provides incorrect result. Because, since the **Flux** model is time dependent, there is no computation to show the signal variations inside the **Flux** component. If a minimum input variation parameter is specified, in order to force the **Flux** engine to perform a computation, the **'Maximum computation interval'** parameter is used. If this parameter is set to T_m value, the **Flux** engine performs at least one computation each $T_m \pm 10\%$ seconds to ensure capturing the signal variations inside the **Flux** component. In order to disable the parameter **'Minimal input variation'**, zero value or a very high value such as 1000% may be used. The parameter **'Maximum computation interval'** is ignored if the parameter **'Minimal input variation'** is not used. As a rule of thumb, the time step-size used by the **Flux** numerical solver would be a appropriate value for this this parameter.

The second way to reduce the simulation time is to reduce the number of calls to the **Flux** engine. In order to advance the time in **Twin Activate** and take a step, the **Twin Activate** simulator engine calls blocks in model several times for several reasons, *e.g.*, at solver mesh-points, at discrete time-events, at zero-crossing time instants, at internal numerical solver points. At all of these instants, the **Flux** block is invoked and its outputs are read by **Twin Activate**. For some of these invocations, it is useless to invoke the **Flux** engine to perform a computation and then update the outputs. For example, at internal updates required by the numerical solver, the updated **Flux** output is not required and an extrapolation of the previous outputs is sufficient. In the **Flux** block GUI, the *Call Flux at Activate event instants* checkbox in the **Flux Activation** tab 14.3.2 indicates if at discrete event time instants of the **Twin Activate** simulator, the **Flux** engine computation is required. By default, the **Flux** block is invoked only at simulator mesh-points. If this checkbox is checked, the **Flux** block is called at mesh-points and at all discrete-time events of the simulator. If the checkbox is not checked, at discrete-event time instant, the **Flux** engine performs an extrapolation and no costly computation is done. Note that even if this extra point checkbox is checked, it does not mean that there will be a **Flux** computation at a point. Actually, it is the **Flux** engine who decides if a computation or an extrapolation should be done, based on the **'Minimal input variation'** and **'Maximum computation interval'** values.

It is important to indicate that when the **Flux** block is externally activated (*i.e.*, **External Activation'**

checkbox is checked), the **'Minimal input variation'** and **'Maximum computation interval'**, as well as the *Call Flux at Activate event instants* checkbox are ignored. In fact, if the block is externally activated, **Twin Activate** forces the **Flux** engine to perform a computation, regardless of these parameters. The external activation is useful if, for example, a periodic update of the **Flux** block is desired. In the model shown in Figure 14.5, the Flux block is called at periodic interval of 0.01 seconds.
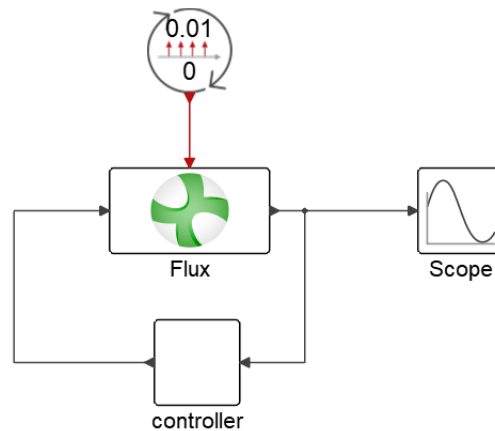

Figure 14.5: **Flux** block activated externally and called at periodic intervals.

14.3 Flux block parameters

The **Flux** block implements CoSimulation interface between **Twin Activate** and **Flux**.

14.3.1 Parameters tab

- **Flux to Activate component file-name:** In this field, the name of the **Flux** component with extension `.f2sta` is given. The **Flux** to **Twin Activate** component (*.F2TSA) is created by **Flux** and should have the same name as the **Flux** project. The file-name can be either given by an absolute path, or a relative path. If **Twin Activate** and **Flux** models are in the same folder, no path is required. In the latter case, the absolute path is obtained from where the **Twin Activate** model is located.

- **Reload button:** The information about the input and output of the **Flux** model, their names, and their sizes are all available in the `(*.f2sta)` file. Instead of filling the GUI fields manually, they can be filled automatically by pressing this button. Filling manually these fields is useful when the `*.f2sta` file is not still available and the user needs to build the **Twin Activate** model before getting the `*.f2sta` file.

- **Multiplexed inputs/outputs:** Select vector or scalar mode for the input/output. Inputs and outputs to **Twin Activate** can be defined as vector or scalars when the `*.f2sta` component is exported by **Flux**.

- **Number of input ports:** The number of input parameters stored in the `*.f2sta` file. This field is automatically filled if the Reload button is pressed.

- **Inputs:** Each row corresponds to an input port. The label of input as well as the size of the input should be provided here. These fields are automatically filled if the Reload button is pressed.
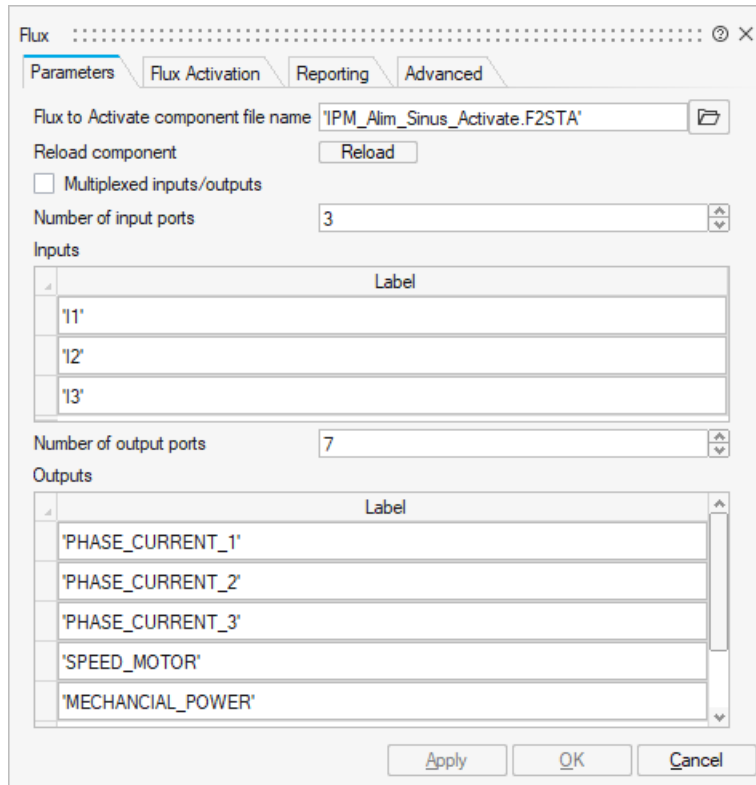
Figure 14.6: Graphical user interface of the **Flux** block: Parameters tab.

- **Number of output ports:** The number of output parameters stored in the `*.f2sta` file. This field is automatically filled if the Reload button is pressed.

- **Outputs:** Each row corresponds to an output port. The label of output as well as the size of the output should be provided here. These fields are automatically filled if the Reload button is pressed.

14.3.2 Flux Activation tab

- **External activation:** Specifies whether the block receives an external activation or inherits its activation through its regular input ports. When 'External Activation' is selected, an additional activation port is added to the block. By default, external activation is not selected. If external activation is selected, the **Flux** engine only performs a computation at external event instants, *e.g.*, at periodic events defined by an external clock. If checked, **'Minimal input variation'**, **'Maximum computation interval'**, and **'Flux Activation'** are disabled and ignored by **Flux** and **Twin Activate**.

- **Extra Flux calls at Twin Activate event instants: Twin Activate** only calls the **Flux** block at **Twin Activate** solver mesh-point to update its output. If this checkbox is checked, the **Flux** block is also called at other instants such as discrete-time events during the CoSimulation. Extra points provides more detailed outputs especially at event instants.

- **Minimal input variation (%) to run Flux computation:** Specifies the percentage of input variation between two calculation steps under which a **Flux** calculation should not occur. When the **Flux** calculation is not carried out, outputs are computed based on the 'Output extrapolation or-
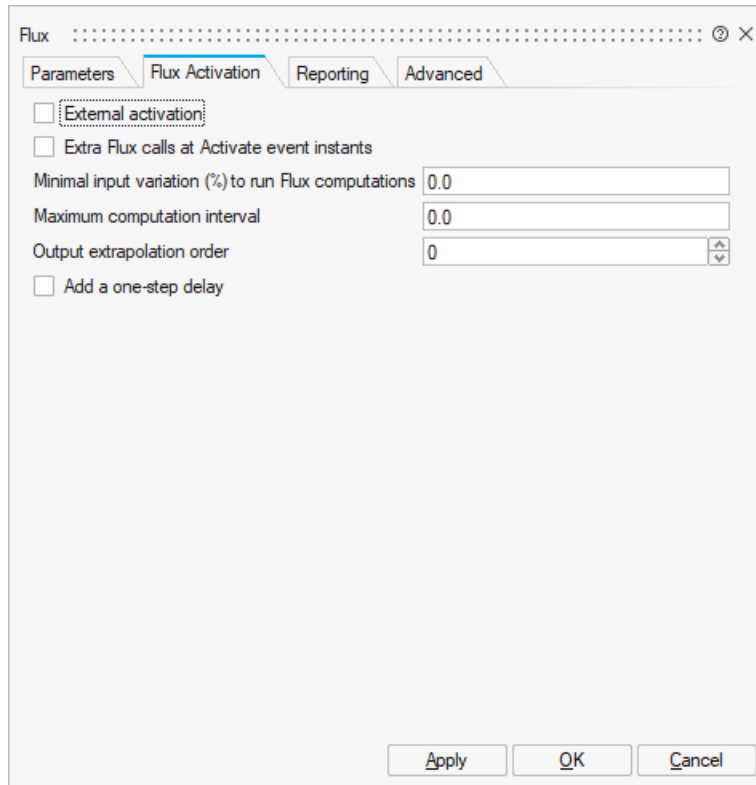
Figure 14.7: Graphical user interface of the **Flux** block: Flux Activation tab.

der' that is defined, therefore permitting several **Twin Activate** time steps to be carried out without any **Flux** calculations.

- **Maximum computation interval:** If a non-zero **'Maximum computation interval'** is provided, and a non-zero **'Minimal input variation'** is specified, the **Flux** engine is forced to perform a computation not later than the **'Maximum computation interval'** since the last computation, even if the input values have not changed.

- **Output extrapolation order:** Choose the output extrapolation order (0 : zero-order hold extrapolation, 1 : linear extrapolation). If **'Minimal input variation'**, **'Maximum computation interval'** parameters are defined and used, when the **Flux** engine does not perform the computation, the output of the **Flux** block is provided using extrapolation. In general, if the **Flux** simulator engine does not perform the complete calculation, *e.g.*, when the **Twin Activate** simulator needs the Flux output at internal solver points (try points), **Flux** provides the output by extrapolation.

- **Add one-step delay:** Add a one-step delay to help avoid the algebraic loops caused by the **Flux** block.

14.3.3   Reporting tab

- **Subsampling (Reduced result storage):** N (which should be greater than 2) means one out of N data is stored in the result data file in **Flux**. This feature is available in **Flux** version 2018 and later.

- **Flux console display:** Select the check box to turn on the **Flux** display console, though this may
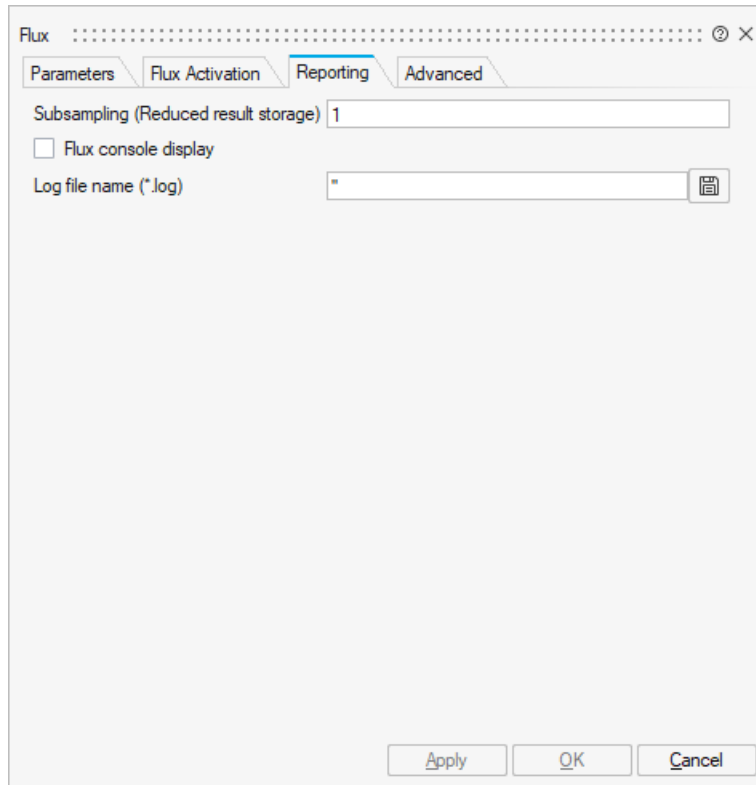
Figure 14.8: Graphical user interface of the **Flux** block: Report tab.

drastically slow down the computation.

- **Log file name (*.log): Twin Activate** messages as well as messages communicated from **Flux** to **Twin Activate** are logged into this file. 'stderr' and 'stdout' are valid values that output log messages into the standard output.

14.3.4 Advanced tab

- **Current Flux version:** Displays the **Flux** version in use. If the installation path of **Flux** is specified in `Preferences` of **Twin Activate**, this field indicates the version of that installation. If no installation path is specified, the user may specify the **Flux** installation path by defining the environment variable `'INSTALLFLUX'`. For example, in the OML command window of **Twin Activate**, the user can enter: `setenv('INSTALLFLUX','D:/AltairSofwtares/Flux-2020/flux/Flux')`

- **Flux component version: Flux** version used to generate the coupling component, *i.e*, the `*.f2sta` file. This information is stored in the `*.f2sta` file.

- **Flux solver:** Displays the **Flux** solver in use. This information is stored in the `*.f2sta` file.

- **Numerical memory (MB):** Specifies the memory used for the various modeling actions. 3D mesh and solving processes (Flux 2D-3D) include a high memory demand. Memory allocation is a function of the application type (real/complex) and matrix size for the solving process. The default value of this information is stored in the `*.f2sta` file.

- **Character memory (MB):** Character memory is used for storing entity names such as parameters, transformations, regions, and project names presented in the directory. The default value of
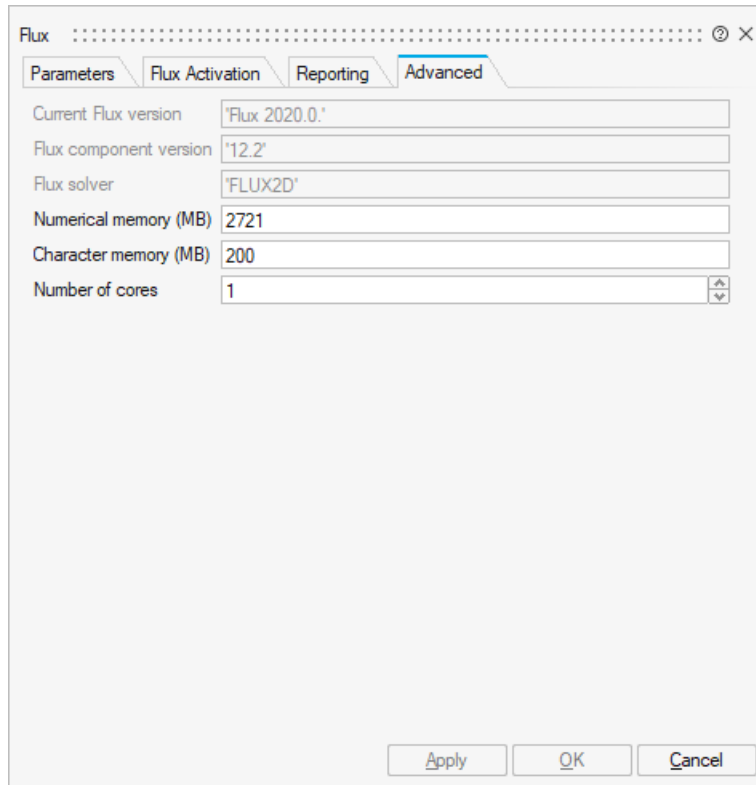
Figure 14.9: Graphical user interface of the **Flux** block: Advanced tab.

this information is stored in the `*.f2sta` file.

- **Number of cores:** Number of cores used by **Flux** during CoSimulation. This feature is available in **Flux** version 2018 and above. in Flux-2D, only 1 core is used.

14.4 Example: Brushless AC embedded permanent magnet motor

Consider the model in Figure 14.10 which is a model for a brushless AC embedded permanent magnet motor. The motor is derived by **Twin Activate** sine wave signals.

The simulation result is shown in Figure 14.11.

14.5 Example: Externally activated block

The Cosimulation with **Flux** block is usually slow due to numerous calls to the **Flux** engine. This CoSimulation of the example in Figure 14.10 takes about 190 seconds. In order to reduce the calls to the **Flux** engine, the Flux block can be activated with an external clock, instead of the **Twin Activate** numerical solver. In Figure 14.12, the model in Figure 14.10, is Activated externally by a clock with the period of 0.001 seconds. In this model, the Flux block perform a computation every one milli-second.

Now the simulation only takes 60 seconds. The reason is the reduced number of calls to Flux. The only drawback of using external activation is a result which is sampled at clock event instants, as shown in Figure 14.13. The sampled result (blue curve) is supoerposed on the result obtained in Figure 14.11
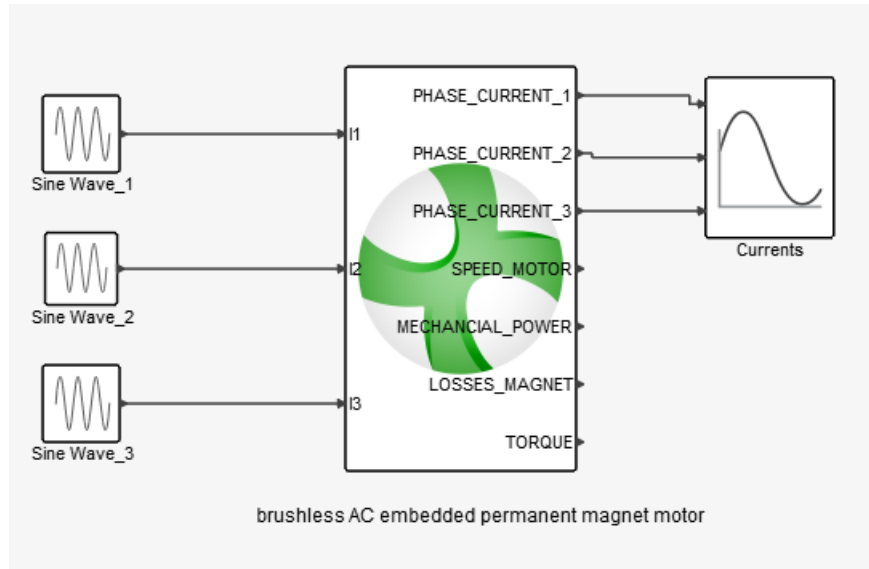
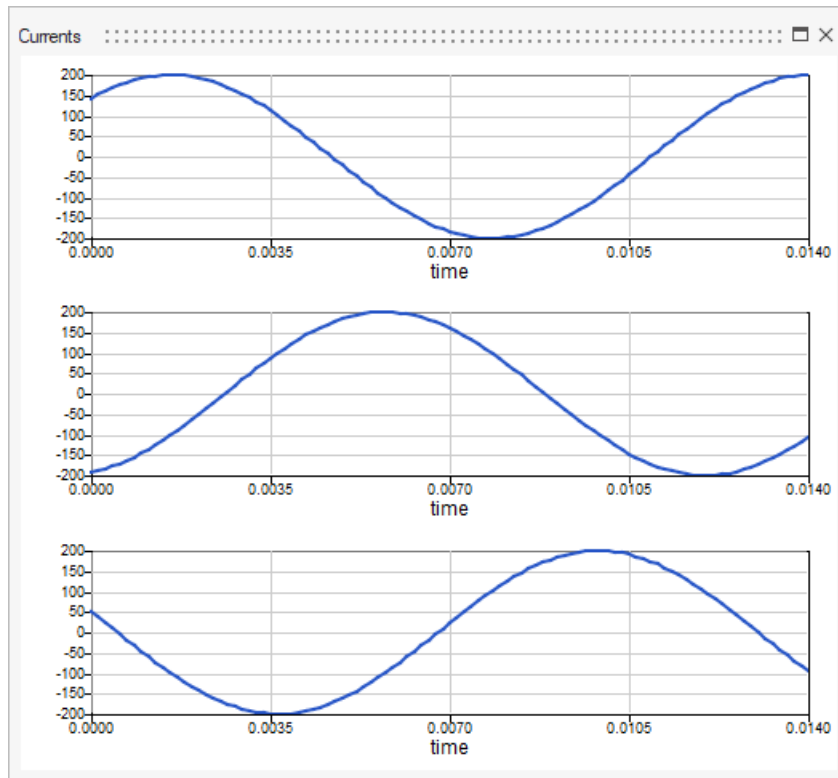Figure 14.10: The brushless AC motor

(red curve).

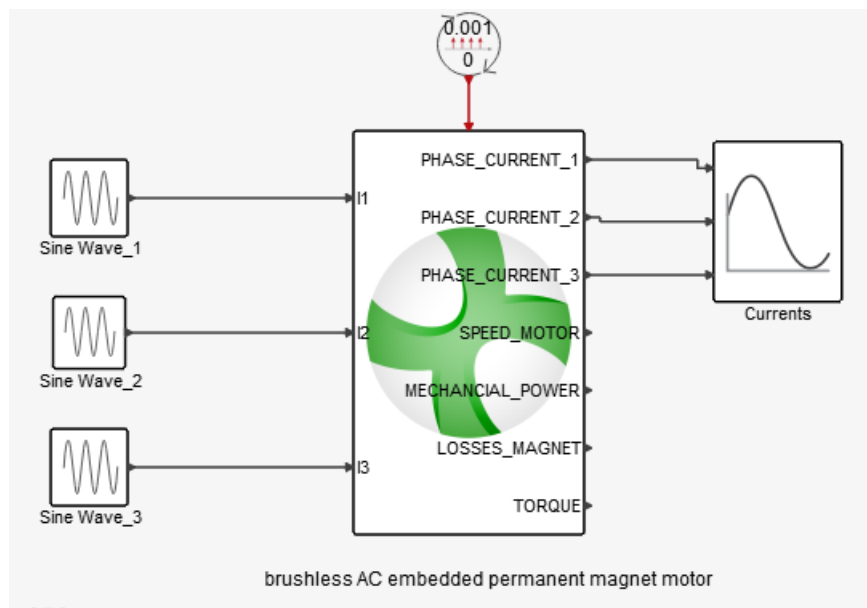Figure 14.11: The brushless AC motor: Simulation result.

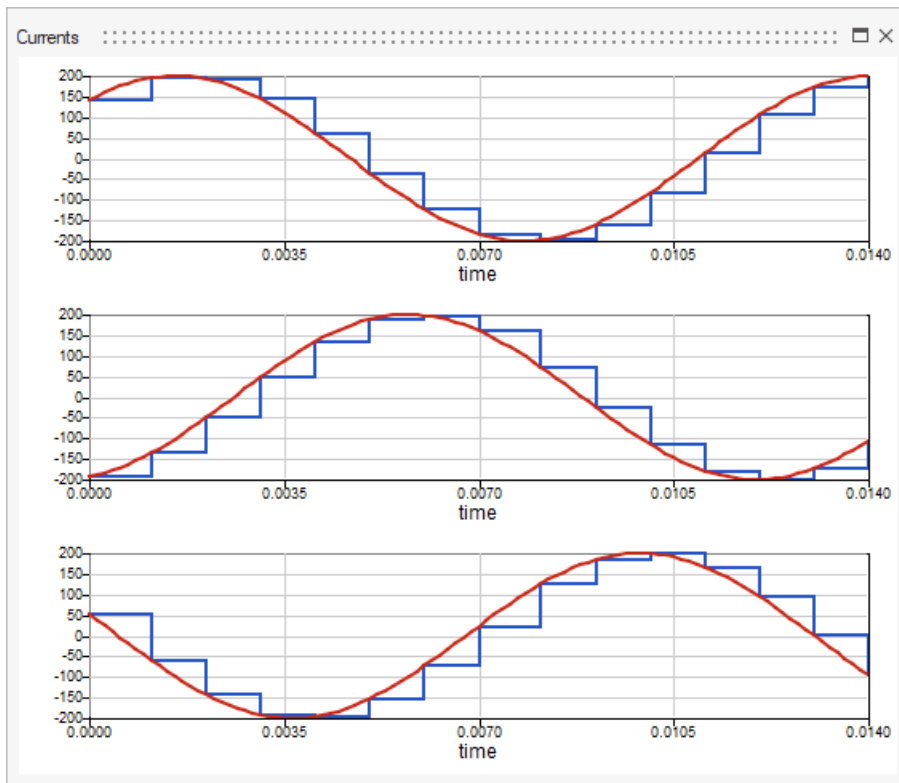Figure 14.12: The brushless AC motor: externally activated by a clock.

Figure 14.13: The brushless AC motor: Simulation result.

Chapter 15

Spice Environment in Twin Activate

15.1   Introduction

This chapter is dedicated to users who want to model and simulate any electronic circuits inside of a given system description. To cover this field, **Twin Activate** is proposing an environment and a simulator which is based on the SPICE language. Without being a specialist in electronic, a designer can easily describe and/or get electronic designs in order to simulate them as a black-box.

15.1.1   What Is SPICE?

SPICE (Simulation Program with Integrated Circuit Emphasis) is a software dedicated to simulating analog circuits with the highest accuracy. The language used as input of a SPICE simulator is a textual representation of the electronic circuit. The language itself is called the SPICE language. As it is a textual representation, there is no specific requirement to define a circuit. Moreover descriptions can be shared easily without any restrictions.

15.1.2   Why Use SPICE?

SPICE is proven to produce highly accurate simulation results and has been fully adopted by silicon foundries as well as analog circuit designers. Even for digital simulations, timing analysis and power consumptions, SPICE simulation has become essential for building libraries and obtaining accurate results.

15.2   Spice^{TM}

The Altair Spice solver^{TM} is a high performance, matrix-based circuit simulator based on the SPICE syntax and Spice3 standard for circuit description. The solver architecture is based on a hierarchical approach to circuit description that enables the simulation of complex circuitry with accuracy, speed, and minimal memory requirements.

The Spice solver performs simulation in three modes:

- DC : Operating point simulation
- Transient: time domain simulation

- AC: frequency response simulation

In terms of device models, the Spice solver uses the Berkeley public models. Devices models level 1, 2 and 3 as well as Bsim3 are supported by Spice.

Below is the list of all supported models:

Devices	AC/DC and Transient simulation
Resistor	X
Capacitor	X
Inductor	X
Mutual inductances	X
Transmission lines	X
Lossy Transmission lines	X
Vsources	X
Isources	X
Voltage Controlled Current Sources	X
Voltage Controlled Voltage Sources	X
Current Controlled Current Sources	X
Current Controlled Voltage Sources	X
Voltage Switches	X
Current Switches	X
S-Parameters	X
Diodes	X
BJTs	X
MOSs	Level 1, 2, 3 and Bsim 3
JFETs	Not Supported yet
MESFETs	Not supported yet

Altair Spice solver takes its input through netlists written with Spice syntax. The Spice components are described as follows. For more details refer to the public documentation at

http://bwrcs.eecs.berkeley.edu/Classes/IcBook/SPICE/UserGuide/elements_fr.html

Altair Spice solver is compatible with other Spice simulators such as Spice3, NGspice, LTspice and PSpice as well as some commercial Spice syntax.

15.3 Spice Language

15.3.1 Units

Spice language uses symbols to associate different unit factors. These characters can be used for any setting values. The following table shows the symbols and their representation:

V	Deg	Db	f	Hz	H	u	k	M	m	n	p	s
Volt	Degree	Decibel	Fento	Hertz	Henri	Micro	Kilo	Mega	Milli	Nano	Pico	Second

15.3.2 Comments

To create a comment in Spice, use the symbol '*' to set the complete line as a comment. However if a partial comment is needed, you can use the symbol '!'. This symbol means that the comment starts from the symbol and continues to the end of line. For example:

```
* This line is a entire comment line
```

```
R1 1 0 1k ! This line is a resistor of 1K, but after the ! it is a comment
```
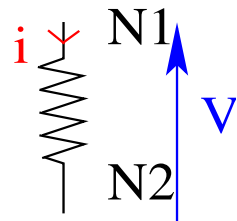
An important thing to take into account is that the Spice language considers the first line as a comment. You should either declare a comment for the first line or consider that the first line will be ignored.

15.3.3 Resistor

Spice Syntax

```
RXXXXXXX N1 N2 VALUE
```

Graphical representation


```
RXXXXXXX N1 N2 <VALUE> <MNAME> <L=LENGTH> <W=WIDTH> <TEMP=T>
```

```
RLOAD 2 10 10K RMOD 3 7 RMODEL L=10u W=1u
```

Equation:

$$V = R * i$$

Examples:

```
R1 N1 N2 100
```

```
RC1 12 17 1K
```

N1 and N2 are the two element nodes. VALUE is the resistance (in ohms) and may be positive or negative but not zero.

This is the more general form of the resistor presented in section 6.1, and allows the modeling of temperature effects and for the calculation of the actual resistance value from strictly geometric information and the specifications of the process. If VALUE is specified, it overrides the geometric information and defines the resistance. If MNAME is specified, then the resistance may be calculated from the process information in the model MNAME and the given LENGTH and WIDTH. If VALUE is not specified, then MNAME and LENGTH **must** be specified. If WIDTH is not specified, then it is taken from the default width given in the model. The (optional) TEMP value is the temperature at which this device is to

operate, and overrides the temperature specification on the .OPTION control line.

The resistor model consists of process-related device data that allow the resistance to be calculated from geometric information and to be corrected for temperature. The parameters available are:

name	parameter	units	default	example
TC1	first order temperature coefficient.	Ω /°C	0.0	-
TC2	second order temperature coefficient.	Ω / $°C^2$	0.0	-
RSH	sheet resistance	Ω /q	-	50
DEFW	default width	meters	1e-6	2.e-6
NARROW	narrowing due to side etching	meters	0.0	1.e-7
TNOM	parameter measurement temperature	°C	27	50

The sheet resistance is used with the narrowing parameter and L and W from the resistor device to determine the nominal resistance by the formula:

$$R = RSH * \frac{L - Narrow}{W - Narrow}$$

DEFW is used to supply a default value for W if one is not specified for the device. If either RSH or L is not specified, then the standard default resistance value of 1k is used. TNOM is used to override the circuit-wide value given on the .OPTIONS control line where the parameters of this model have been measured at a different temperature. After the nominal resistance is calculated, it is adjusted for temperature by the formula:

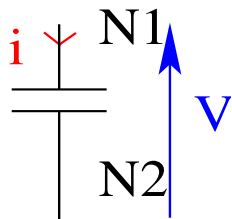$$R(T) = R(T_0) \left[1 + TC_1(T - T_0) + TC2(T - T_0)^2\right]$$

15.3.4 Capacitor

Spice Syntax

```
CXXXXXXX N+ N- VALUE
```

```
CXXXXXXX N1 N2 <VALUE> <MNAME> <L=LENGTH> <W=WIDTH>
```

Graphical representation

Examples:

```
Cap1 13 0 1n
```

```
CBYP 13 0 1UF COSC 17 23 10U
```

```
CLOAD 2 10 10P CMOD 3 7 CMODEL L=10u W=1u
```

N+ and N- are the positive and negative element nodes, respectively. VALUE is the capacitance in Farads.

The (optional) initial condition is the initial (time-zero) value of capacitor voltage (in Volts). Note that the initial conditions (if any) apply 'only' if the UIC option is specified on the .TRAN control line.

This is the more general form of the Capacitor presented in section 6.2, and allows for the calculation of the actual capacitance value from strictly geometric information and the specifications of the process. If VALUE is specified, it defines the capacitance. If MNAME is specified, then the capacitance is calculated from the process information in the model MNAME and the given LENGTH and WIDTH. If VALUE is not specified, then MNAME and LENGTH **must** be specified. If WIDTH is not specified, then it is taken from the default width given in the model. Either VALUE or MNAME, LENGTH, and WIDTH may be specified, but not both sets.

The capacitor model contains process information that may be used to compute the capacitance from strictly geometric information.

name	parameter	units	default	example
CJ	junction bottom capacitance	$F/meters^2$	-	5.e-5
CJSW	junction sidewall capacitance	F/meters	-	2.e-11
DEFW	default device width	meters	1.e-6	2.e-6
NARROW	narrowing due to side etching	meters	0.0	1.e-7

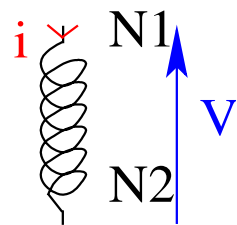The capacitor has a capacitance computed as:

$Cap = CJ(Length - Narrow)(Width - Narrow) + 2 * CJSW(Length + Width - 2 * Narrow)$

15.3.5 Inductor

Spice Syntax

```
LYYYYYYY N+ N- VALUE <IC=INCOND>
```

Graphical representation

Examples:

```
LLINK 42 69 1UH
```

```
LSHUNT 23 51 10U IC=15.7MA
```

N+ and N- are the positive and negative element nodes, respectively. VALUE is the inductance in Henries.

The (optional) initial condition is the initial (time-zero) value of inductor current (in Amps) that flows from N+, through the inductor, to N-. Note that the initial conditions (if any) apply only if the UIC option is specified on the .TRAN analysis line.

Equation:

$$V = L\frac{di}{dt}$$

15.3.6 Mutual Inductors

Spice syntax

```
KXXXXXXX LYYYYYYY LZZZZZZZ VALUE
```

Examples:

```
K43 LAA LBB 0.999
```

```
KXFRMR L1 L2 0.87
```

The K device is coupled Mutual Inductors. LYYYYYYY and LZZZZZZZ are the names of the two coupled inductors, and VALUE is the coefficient of coupling, K, which must be greater than 0 and less than or equal to 1. Using the 'dot' convention, place a 'dot' on the first node of each inductor.

The mutual inductance is equal to:

$$M = \sqrt{L\& * L2}$$

15.3.7 Mosfet

Spice syntax

```
MXXXXXXX ND NG NS NB MNAME <L=VAL> <W=VAL> <AD=VAL> <AS=VAL>
+ <PD=VAL> <PS=VAL> <NRD=VAL> <NRS=VAL> <OFF>
+ <IC=VDS, VGS, VBS> <TEMP=T>
```

Examples:

```
M1 24 2 0 20 TYPE1
```

```
M31 2 17 6 10 MODM L=5U W=2U
```

```
M1 2 9 3 0 MOD1 L=10U W=5U AD=100P AS=100P PD=40U PS=40U
```

ND, NG, NS, and NB are the drain, gate, source, and bulk (substrate) nodes, respectively. MNAME is the model name.
L and W are the channel length and width, in meters.
AD and AS are the areas of the drain and source diffusions, in m^2. Note that the suffix U specifies microns (1e-6 m) and P square-microns (1e-12 m^2). If any of L, W, AD, or AS are not specified, default values are used. The use of defaults simplifies input file preparation, as well as the editing required if device geometries are to be changed.
PD and PS are the perimeters of the drain and source junctions, in meters. RD and NRS designate the equivalent number of squares of the drain and source diffusions; these values multiply the sheet resistance RSH specified on the .MODEL control line for an accurate representation of the parasitic series drain and source resistance of each transistor.
PD and PS default to 0.0 while NRD and NRS to 1.0.
OFF indicates an (optional) initial condition on the device for dc analysis. The (optional) initial condition specification using IC=VDS, VGS, VBS is intended for use with the UIC option on the .TRAN control

line, when a transient analysis is desired starting from other than the quiescent operating point. See the .IC control line for a better and more convenient way to specify transient initial conditions.

The (optional) TEMP value is the temperature at which this device is to operate, and overrides the temperature specification on the .OPTION control line. The temperature specification is ONLY valid for level 1, 2, 3, and 6 MOSFETs, not for level 4 or 5 (BSIM) devices.

MOSFET Models (NMOS/PMOS)

SPICE provides several MOSFET device models, which differ in the formulation of the I-V characteristic. The variable LEVEL specifies the model to be used.

The models are declared using the syntax:

`.MODEL MOSMNAME <PMOS|NMOS> LEVEL=<level number> <PARAMETER=<VALUE>>`

At least for a MOS model the user should select the level number. For this he has to choose among the following models:

Level number	Mos type
0, 1	Mos Level 1
2	Mos Level 2
3	Mos Level 3
13	Bsim1
39	Bsim2
49, 53	Bsim3 v3.3(.2)

The DC characteristics of the level 1 through level 3 MOSFETs are defined by the device parameters VTO, KP, LAMBDA, PHI and GAMMA. These parameters are computed by SPICE if process parameters (NSUB, TOX, ...) are given, but user-specified values always override. VTO is positive (negative) for enhancement mode and negative (positive) for depletion mode N-channel (P-channel) devices. Charge storage is modeled by three constant capacitors, CGSO, CGDO, and CGBO which represent overlap capacitances, by the nonlinear thin-oxide capacitance which is distributed among the gate, source, drain, and bulk regions, and by the nonlinear depletion-layer capacitances for both substrate junctions divided into bottom and periphery, which vary as the MJ and MJSW power of junction voltage respectively, and are determined by the parameters CBD, CBS, CJ, CJSW, MJ, MJSW and PB.

Charge storage effects are modeled by the piecewise linear voltages-dependent capacitance model proposed by Meyer. The thin-oxide charge-storage effects are treated slightly different for the LEVEL=1 model. These voltage-dependent capacitances are included only if TOX is specified in the input description and they are represented using Meyer's formulation.

There is some overlap among the parameters describing the junctions, e.g. the reverse current can be input either as IS (in A) or as JS (in A/m^2). Whereas the first is an absolute value the second is multiplied by AD and AS to give the reverse current of the drain and source junctions respectively. This methodology has been chosen since there is no sense in relating always junction characteristics with AD and AS entered on the device line; the areas can be defaulted. The same idea applies also to the zero-bias junction capacitances CBD and CBS (in F) on one hand, and CJ (in F/m^2) on the other. The parasitic drain and source series resistance can be expressed as either RD and RS (in ohms) or RSH (in Ω/sq.), the latter being multiplied by the number of squares NRD and NRS input on the device line.

SPICE level 1, 2, 3 parameters:

	name	parameter	units	default	example
1	LEVEL	model	index	-	1
2	VTO	zero-bias threshold voltage (VT0)	V	0.0	1.0
3	KP	transconductance parameter	A/V^2	2.0e-5	3.1e-5
4	GAMMA	bulk threshold parameter ()	$V^{1/2}$	0.0	0.37
5	PHI	surface potential ()	V	0.6	0.65
6	LAMBDA	channel-length modulation	1/V	0.0	0.02
		MOS1 and MOS2 only			
7	RD	drain ohmic resistance	Ω	0.0	1.0
8	RS	source ohmic resistance	Ω	0.0	1.0
9	CBD	zero-bias B-D junction capacitance	F	0.0	20fF
10	CBS	zero-bias B-S junction capacitance	F	0.0	20fF
11	IS	bulk junction saturation current (IS)	A	1.0e-14	1.0e-15
12	PB	bulk junction potential	V	0.8	0.87
13	CGSO	gate-source overlap capacitance per meter channel width	F/m	0.0	4.0e-11
14	CGDO	gate-drain overlap capacitance per meter channel width	F/m	0.0	4.0e-11
15	CGBO	gate-bulk overlap capacitance per meter channel length	F/m	0.0	2.0e-10
16	RSH	drain and source diffusion sheet resistance	Ω/q	0.0	10.0
17	CJ	zero-bias bulk junction bottom cap. per sq-meter of junction area	F/m^2	0.0	2.0e-4
18	MJ	bulk junction bottom grading coeff.	-	0.5	0.5
19	CJSW	zero-bias bulk junction sidewall cap. per meter of junction perimeter	F/m	0.0	1.0e-9
20	MJSW	bulk junction sidewall grading coeff.	-	0.50(level1) 0.33(level2,3)	
21	JS	bulk junction saturation current per sq-meter of junction area	A/m^2		1.0e-8
22	TOX	oxide thickness	meter	1.0e-7	1.0e-7
23	NSUB	substrate doping	1/cm3	0.0	4.0e15
24	NSS	surface state density	$1/cm^2$	0.0	1.0e10
25	NFS	fast surface state density	$1/cm^2$	0.0	1.0e10
26	TPG	type of gate material: +1 opp. to substrate -1 same as substrate 0 Al gate	-	1.0	
27	XJ	metallurgical junction depth	meter	0.0	1
28	LD	lateral diffusion	meter	0.0	0.8
29	UO	surface mobility	cm^2/Vs	600	700

30	UCRIT	critical field for mobility degradation (MOS2 only)	V/cm	1.0e4	1.0e4
31	UEXP	critical field exponent in mobility degradation (MOS2 only)	-	0.0	0.1
32	UTRA	transverse field coeff. (mobility) (deleted for MOS2)	-	0.0	0.3
33	VMAX	maximum drift velocity of carriers	m/s	0.0	5.0e4
34	NEFF	total channel-charge (fixed and mobile) coefficient (MOS2 only)	-	1.0	5.0
35	KF	flicker noise coefficient	-	0.0	1.0e-26
36	AF	flicker noise exponent	-	1.0	1.2
37	FC	coefficient for forward-bias depletion capacitance formula	-	0.5	
38	DELTA	width effect on threshold voltage (MOS2 and MOS3)	-	0.0	1.0
39	THETA	mobility modulation (MOS3 only)	1/V	0.0	0.1
40	ETA	static feedback (MOS3 only)	-	0.0	1.0
41	KAPPA	saturation field factor (MOS3 only)	-	0.2	0.5
42	TNOM	parameter measurement temperature	C	27	50

For more information on models see the Berkeley device models documentation.

15.3.8 BJT

Spice syntax

```
QXXXXXXX NC NB NE <NS> MNAME <AREA> <OFF> <IC=VBE, VCE> <TEMP=T>

.MODEL MNAME NPN|PNP <Param=Value>
```

Examples:

```
Q23 10 24 13 QMOD IC=0.6

Q50A 11 26 4 20 MOD1
```

The BJT device is a Bipolar Junction Transistor.

NC, NB, and NE are the collector, base, and emitter nodes, respectively.
NS is the (optional) substrate node. If unspecified, ground is used.
MNAME is the model name, AREA is the area factor, and OFF indicates an (optional) initial condition on the device for the dc analysis. If the area factor is omitted, a value of 1.0 is assumed.
The (optional) initial condition specification using IC=VBE, VCE is intended for use with the UIC option on the .TRAN control line, when a transient analysis is desired starting from other than the quiescent operating point. See the .IC control line description for a better way to set transient initial conditions. The (optional) TEMP value is the temperature at which this device is to operate, and overrides the temperature specification on the .OPTION control line.

BJT Models (NPN/PNP)

The bipolar junction transistor model in SPICE is an adaptation of the integral charge control model of Gummel and Poon. This modified Gummel-Poon model extends the original model to include several effects at high bias levels. The model automatically simplifies to the simpler Ebers-Moll model when certain parameters are not specified. The parameter names used in the modified Gummel-Poon model have been chosen to be more easily understood by the program user, and to reflect better both physical and circuit design thinking.

The dc model is defined by the parameters IS, BF, NF, ISE, IKF, and NE which determine the forward current gain characteristics, IS, BR, NR, ISC, IKR, and NC which determine the reverse current gain characteristics, and VAF and VAR which determine the output conductance for forward and reverse regions.

Three ohmic resistances RB, RC, and RE are included, where RB can be high current dependent. Base charge storage is modeled by forward and reverse transit times, TF and TR, the forward transit time TF being bias dependent if desired, and nonlinear depletion layer capacitances which are determined by CJE, VJE, and MJE for the B-E junction , CJC, VJC, and MJC for the B-C junction and CJS, VJS, and MJS for the C-S (Collector-Substrate) junction.

The temperature dependence of the saturation current, IS, is determined by the energy-gap, EG, and the saturation current temperature exponent, XTI. Additionally base current temperature dependence is modeled by the beta temperature exponent XTB in the new model. The values specified are assumed to have been measured at the temperature TNOM, which can be specified on the .OPTIONS control line or overridden by a specification on the .MODEL line.

The BJT parameters used in the modified Gummel-Poon model are listed below. The parameter names used in earlier versions of SPICE2 are still accepted.

Modified Gummel-Poon BJT Parameters

	name	parameter	units	default	example	area
1	IS	transport saturation current	A	1.0e-16	1.0e-15	*
2	BF	ideal maximum forward beta	-	100	100	
3	NF	forward current emission coefficient	-	1.0	1	
4	VAF	forward Early voltage	V	infinite	200	
5	IKF	corner for forward beta high current roll-off	A	infinite	0.01	*
6	ISE	B-E leakage saturation current	A	0	1.0e-13	*
7	NE	B-E leakage emission coefficient	-	1.5	2	
8	BR	ideal maximum reverse beta	-	1	0.1	
9	NR	reverse current emission coefficient	-	1	1	
10	VAR	reverse Early voltage	V	infinite	200	
11	IKR	corner for reverse beta high current roll-off	A	infinite	0.01	*
12	ISC	leakage saturation current	A	0		8
13	NC	leakage emission coefficient	-	2	1.5	
14	RB	zero bias base resistance	Ω	0	100	*
15	IRB	current where base resistance falls halfway to its min value	A	infinite	0.1	*
16	RBM	minimum base resistance at high currents	Ω	RB	10	*
17	RE	emitter resistance	Ω	0	1	*
18	RC	collector resistance	Ω	0	10	*
19	CJE	B-E zero-bias depletion capacitance	F	0	2pF	*

20	VJE	B-E built-in potential	V	0.75	0.6	
21	MJE	B-E junction exponential factor	-	0.33	0.33	
22	TF	ideal forward transit time	sec	0	0.1ns	
23	XTF	coefficient for bias dependence of TF	-	0		
24	VTF	voltage describing VBC dependence of TF	V	infinite		
25	ITF	high-current parameter for effect on TF	A	0		*
26	PTF	excess phase at freq=1.0/(TF*2PI) Hz	deg	0		
27	CJC	B-C zero-bias depletion capacitance	F	0	2pF	*
28	VJC	B-C built-in potential	V	0.75	0.5	
29	MJC	B-C junction exponential factor	-	0.33	0.5	
30	XCJC	fraction of B-C depletion capacitance connected to internal base node	-	1		
31	TR	ideal reverse transit time	sec	0	10ns	
32	CJS	zero-bias collector-substrate capacitance	F	0	2pF	*
33	VJS	substrate junction built-in potential	V	0.75		
34	MJS	substrate junction exponential factor	-	0	0.5	
35	XTB	forward and reverse beta temperature exponent	-	0		
36	EG	energy gap for temperature effect on IS	eV	1.11		
37	XTI	temperature exponent for effect on IS	-	3		
38	KF	flicker-noise coefficient	-	0		
39	AF	flicker-noise exponent	-	1		
40	FC	coefficient for forward-bias depletion capacitance formula	-	0.5		
41	TNOM	Parameter measurement temperature	°C	27	50	

15.3.9 JFET

Spice Syntax

```
JXXXXXXX ND NG NS MNAME <AREA> <OFF> <IC=VDS, VGS> <TEMP>
```

Examples:

```
J1 7 2 3 JM1 OFF
```

The JFET device is a Junction Field-Effect Transistors (JFETs). ND, NG, and NS are the drain, gate, and source nodes, respectively. MNAME is the model name, AREA is the area factor, and OFF indicates an (optional) initial condition on the device for dc analysis. If the area factor is omitted, a value of 1.0 is assumed.

The (optional) initial condition specification, using IC=VDS, VGS is intended for use with the UIC option on the .TRAN control line, when a transient analysis is desired starting from other than the quiescent operating point. See the .IC control line for a better way to set initial conditions.

The (optional) TEMP value is the temperature at which this device is to operate, and overrides the temperature specification on the .OPTION control line.

JFET Models (NJF/PJF)

The JFET model is derived from the FET model of Shichman and Hodges. The dc characteristics are defined by the parameters VTO and BETA, which determine the variation of drain current with gate voltage, LAMBDA, which determines the output conductance, and IS, the saturation current of the two gate junctions. Two ohmic resistances, RD and RS, are included. Charge storage is modeled by nonlinear depletion layer capacitances for both gate junctions which vary as the -1/2 power of junction voltage and are defined by the parameters CGS, CGD, and PB.

Note that in Spice3f and later, a fitting parameter B has been added. For details, see [9].

	name	parameter	units	default	example	area
1	VTO	threshold voltage (VT0)	V	-2.0	-2.0	
2	BETA	transconductance parameter	A/V^2	1.0e-4	1.0e-3	*
3	LAMBDA	channel-length modulation	$1/V$	0	1.0e-4	
4	RD	drain ohmic resistance	Ω	0	100	*
5	RS	source ohmic resistance	Ω	0	100	*
6	CGS	zero-bias G-S junction capacitance (Cgs)	F	0	5pF	*
7	CGD	zero-bias G-D junction capacitance (Cgs)	F	0	1pF	*
8	PB	gate junction potential	V	1	0.6	
9	IS	gate junction saturation current (IS)	A	1.0e-14	1.0e-14	*
10	B	doping tail parameter	-	1	1.1	
11	KF	flicker noise coefficient	-	0		
12	AF	flicker noise exponent	-	1		
13	FC	coefficient for forward-bias	-	0.5		
14	TNOM	parameter measurement temperature	℃	27	50	

15.3.10   Switches

V-Switch

Spice syntax

```
SXXXXXXX N+ N- NC+ NC- MODEL <ON><OFF>
```

Examples:

```
.MODEL smodel1 RON=100K

Switch1 1 2 10 0 smodel1
```

The S- device is an ideal voltage switch. Nodes 1 and 2 are the nodes between which the switch terminals are connected. The model name is mandatory while the initial conditions are optional nodes, 3 and 4 are the positive and negative controlling nodes respectively.

Switch Model (SW)

The switch model allows an almost ideal switch to be described in SPICE.

The switch is not quite ideal, in that the resistance cannot change from 0 to infinity, but must always have a finite positive value. By proper selection of the on and off resistances, they can be effectively zero and infinity in comparison to other circuit elements.

The parameters available are:

name	parameter	units	default
VT	threshold voltage	Volts	0.0
VH	hysteresis voltage	Volts	0.0
RON	on resistance		1.0
ROFF	off resistance		$1/GMIN$*

*(See the .OPTIONS control line for a description of GMIN, its default value results in an off-resistance of 1.0e+12 ohms.)

The use of an ideal element that is highly nonlinear such as a switch can cause large discontinuities to occur in the circuit node voltages.

A rapid change such as that associated with a switch changing state can cause numerical roundoff or tolerance problems leading to erroneous results or timestep difficulties.

The user of switches can improve the situation by taking the following steps:

First, it is wise to set ideal switch impedances just high or low enough to be negligible with respect to other circuit elements. Using switch impedances that are close to "ideal" in all cases aggravates the problem of discontinuities mentioned above. Of course, when modeling real devices such as MOS-FETS, the on resistance should be adjusted to a realistic level depending on the size of the device being modeled.

If a wide range of ON to OFF resistance must be used in the switches ($ROFF/RON$ >1e+12), then the tolerance on errors allowed during transient analysis should be decreased by using the .OPTIONS control line and specifying TRTOL to be less than the default value of 7.0. When switches are placed around capacitors, then the option CHGTOL should also be reduced. Suggested values for these two options are 1.0 and 1e-16 respectively. These changes inform SPICE3 to be more careful around the switch points so that no errors are made due to the rapid change in the circuit.

C-Switch

Spice syntax

```
WYYYYYYY N+ N- VNAM MODEL <ON><OFF>
```

Examples:

```
.MODEL switchmod1 IT=1e-3

W1 1 2 vclock switchmod1
```

The W-device is a Current controlled switch. Nodes 1 and 2 are the nodes between which the switch terminals are connected. The model name is mandatory while the initial conditions are optional. The controlling current is that through the specified voltage source. The direction of positive controlling current flow is from the positive node, through the source, to the negative node.

Switch Model (CSW)

The switch model allows an almost ideal switch to be described in SPICE.

The switch is not quite ideal, in that the resistance cannot change from 0 to infinity, but must always have a finite positive value. By proper selection of the on and off resistances, they can be effectively

zero and infinity in comparison to other circuit elements.

The parameters available are:

name	parameter	units	default
IT	threshold current	Amps	0.0
IH	hysteresis current	Amps	0.0
RON	on resistance		1.0
ROFF	off resistance		$1/GMIN$*

*(See the .OPTIONS control line for a description of GMIN, its default value results in an off-resistance of 1.0e+12 ohms.)

The use of an ideal element that is highly nonlinear such as a switch can cause large discontinuities to occur in the circuit node voltages. A rapid change such as that associated with a switch changing state can cause numerical roundoff or tolerance problems leading to erroneous results or timestep difficulties. The user of switches can improve the situation by taking the following steps:

First, it is wise to set ideal switch impedances just high or low enough to be negligible with respect to other circuit elements. Using switch impedances that are close to "ideal" in all cases aggravates the problem of discontinuities mentioned above. Of course, when modeling real devices such as MOS-FETS, the on resistance should be adjusted to a realistic level depending on the size of the device being modeled.

If a wide range of ON to OFF resistance must be used in the switches ($ROFF/RON$ >1e+12), then the tolerance on errors allowed during transient analysis should be decreased by using the .OPTIONS control line and specifying TRTOL to be less than the default value of 7.0. When switches are placed around capacitors, then the option CHGTOL should also be reduced. Suggested values for these two options are 1.0 and 1e-16 respectively. These changes inform SPICE3 to be more careful around the switch points so that no errors are made due to the rapid change in the circuit.

15.3.11   Transmission Lines

Lossless Transmission Lines

Spice syntax

```
TXXXXXXX N1 N2 N3 N4 Z0=VALUE <TD=VALUE> <F=FREQ <NL=NRMLEN>>
+ <IC=V1, I1, V2, I2>
```

Examples:

```
T1 1 0 2 0 Z0=50 TD=10NS
```

The T device is a Lossless Transmission Lines.

N1 and N2 are the nodes at port 1.

N3 and N4 are the nodes at port 2.

Z0 is the characteristic impedance.

The length of the line may be expressed in either of two forms. The transmission delay, TD, may be specified directly (as TD=10ns, for example). Alternatively, a frequency F may be given, together with NL, the normalized electrical length of the transmission line with respect to the wavelength in the line at the frequency F. If a frequency is specified but NL is omitted, 0.25 is assumed (that is, the frequency is assumed to be the quarter-wave frequency). Note that although both forms for expressing the line

length are indicated as optional, one of the two must be specified.

Note that this element models only one propagating mode. If all four nodes are distinct in the actual circuit, then two modes may be excited. To simulate such a situation, two transmission-line elements are required.

The (optional) initial condition specification consists of the voltage and current at each of the transmission line ports. Note that the initial conditions (if any) apply 'only' if the UIC option is specified on the .TRAN control line.

Note that a lossy transmission line (see below) with zero loss may be more accurate than than the lossless transmission line due to implementation details.

Lossy Transmission Lines

Spice syntax

`OXXXXXXX N1 N2 N3 N4 MNAME`

Examples:

`O23 1 0 2 0 LOSSYMOD`

`OCONNECT 10 5 20 5 INTERCONNECT`

The O device is a Lossy Transmission Lines. This is a two-port convolution model for single-conductor lossy transmission lines.

N1 and N2 are the nodes at port 1

N3 and N4 are the nodes at port 2.

Note that a lossy transmission line with zero loss may be more accurate than the lossless transmission line due to implementation details.

Lossy Transmission Line Model (LTRA)

The uniform RLC/RC/LC/RG transmission line model (referred to as the LTRA model henceforth) models a uniform constant-parameter distributed transmission line.

The RC and LC cases may also be modeled using the URC and TRA models. However, the newer LTRA model is usually faster and more accurate than the others.

The operation of the LTRA model is based on the convolution of the transmission line's impulse responses with its inputs.

The LTRA model takes a number of parameters, some of which must be given and some of which are optional.

name	parameter	units/type	default	example
R	resistance/length		0.0	0.2
L	inductance/length	henrys/unit	0.0	9.13e-9
G	conductance/length	mhos/unit	0.0	0.0
C	capacitance/length	farads/unit	0.0	3.65e-12
LEN	length of line		no default	1.0
REL	breakpoint control	arbitrary unit	1	0.5
ABS	breakpoint control		1	5
NOSTEPLIMIT	don't limit timestep to less than line delay	flag	not set	set
NOCONTROL	don't do complex timestep control	flag	not set	set
LININTERP	use linear interpolation	flag	not set	set
MIXEDINTERP	use linear when quadratic seems bad		not set	set
COMPACTREL	special reltol for history compaction	flag	RELTOL	1.0e-3
COMPACTABS	special abstol for history compaction		ABSTOL	1.0e-9
TRUNCNR	use Newton-Raphson method for timestep control	flag	not set	set
TRUNCDONTCUT	don't limit timestep to keep impulse-esponse errors low	flag	not set	set

The following types of lines have been implemented so far:

RLC (uniform transmission line with series loss only), RC (uniform RC line), LC (lossless transmission line), and RG (distributed series resistance and parallel conductance only).

Any other combination will yield erroneous results and should not be tried.

The length LEN of the line must be specified.

NOSTEPLIMIT is a flag that will remove the default restriction of limiting time-steps to less than the line delay in the RLC case.

NOCONTROL is a flag that prevents the default limiting of the time-step based on convolution error criteria in the RLC and RC cases. This speeds up simulation but may in some cases reduce the accuracy of results.

LININTERP is a flag that, when specified, will use linear interpolation instead of the default quadratic interpolation for calculating delayed signals.

MIXEDINTERP is a flag that, when specified, uses a metric for judging whether quadratic interpolation is not applicable and if so uses linear interpolation; otherwise it uses the default quadratic interpolation.

TRUNCDONTCUT is a flag that removes the default cutting of the time-step to limit errors in the actual calculation of impulse-response related quantities.

COMPACTREL and COMPACTABS are quantities that control the compaction of the past history of values stored for convolution. Larger values of these lower accuracy but usually increase simulation speed. These are to be used with the TRYTOCOMPACT option, described in the .OPTIONS section.

TRUNCNR is a flag that turns on the use of Newton-Raphson iterations to determine an appropriate timestep in the timestep control routines. The default is a trial and error procedure by cutting the previous timestep in half.

REL and ABS are quantities that control the setting of breakpoints.

The option most worth experimenting with for increasing the speed of simulation is REL. The default value of 1 is usually safe from the point of view of accuracy but occasionally increases computation time. A value greater than 2 eliminates all breakpoints and may be worth trying depending on the nature of the rest of the circuit, keeping in mind that it might not be safe from the viewpoint of accuracy. Breakpoints may usually be entirely eliminated if it is expected the circuit will not display sharp discontinuities. Values between 0 and 1 are usually not required but may be used for setting many breakpoints.

COMPACTREL may also be experimented with when the option TRYTOCOMPACT is specified in a .OPTIONS card. The legal range is between 0 and 1. Larger values usually decrease the accuracy of the simulation but in some cases improve speed. If TRYTOCOMPACT is not specified on a .OPTIONS card, history compaction is not attempted and accuracy is high.

NOCONTROL, TRUNCDONTCUT and NOSTEPLIMIT also tend to increase speed at the expense of accuracy.

15.3.12 Diode

Spice syntax

```
.MODEL MNAME DIO <Param=value>

DXXXXXXX N1 N2 MNAME <AREA>> <OFF> <IC=VD> <TEMP>
```

Graphical representation
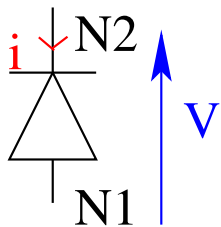
Examples:

```
DBRIDGE 2 10 DIODE1

DCLMP 3 7 DMOD 3.0 IC=0.2
```

The D device is a junction diode. N1 and N2 are the positive and negative nodes, respectively. N1 is the Anode and N2 is the Cathode.

MNAME is the model name, AREA is the area factor, and OFF indicates an (optional) starting condition on the device for dc analysis. If the area factor is omitted, a value of 1.0 is assumed.

The (optional) initial condition specification using IC=VD is intended for use with the UIC option on the .TRAN control line, when a transient analysis is desired starting from other than the quiescent operating point.

The (optional) TEMP value is the temperature at which this device is to operate, and overrides the temperature specification on the .OPTION control line.

Diode Model (D)

The DC characteristics of the diode are determined by the parameters IS and N. An ohmic resistance, RS, is included. Charge storage effects are modeled by a transit time, TT, and a nonlinear depletion layer capacitance which is determined by the parameters CJO, VJ, and M. The temperature dependence of the saturation current is defined by the parameters EG, the energy and XTI, the saturation current temperature exponent.

The nominal temperature at which these parameters were measured is TNOM, which defaults to the circuit-wide value specified on the .OPTIONS control line. Reverse breakdown is modeled by an exponential increase in the reverse diode current and is determined by the parameters BV and IBV (both of which are positive numbers).

	name	parameter	units	default	example	area
1	IS	saturation current	A	1.0e-14	1.0e-14	*
2	RS	ohmic resistance	Ω	0	10	*
3	N	emission coefficient	-	1	1.0	
4	TT	transit-time	sec	0	0.1ns	
5	CJO	zero-bias junction capacitance	F	0	2pF	*
6	VJ	junction potential	V	1	0.6	
7	M	grading coefficient	-	0.5	0.5	
8	EG	activation energy	eV	1.11	1.11 Si 0.69 Sbd 0.67Ge	
9	XTI	saturation-current temp. exp	-	3.0	3.0jn 2.0Sbd	
10	KF	flicker noise coefficient	-	0		
11	AF	flicker noise exponent	-	1		
12	FC	coefficient for forward-bias depletion capacitance formula	-	0.5		
13	BV	reverse breakdown voltage	V	infinite	40.0	
14	IBV	current at breakdown voltage	A	1.0e-3		
15	TNOM	parameter measurement temperature	℃	27	50	

15.3.13 S-Parameter blocks

Spice syntax

```
.MODEL MNAME SPARAM <PARAM=VALUE>

YXXXXXX MNAME PARAM: PARAM=VALUE PIN: PIN1 PIN2 .. PINn
```

Examples:

```
.model SPARAMM SPARAM

ysparam SPARAMM param:
+ force_passivity=1
+ sparamfile=lowpassfilter2.s2p
+ pin:
```

```
+ 2 0 3 0
```

Graphical representation
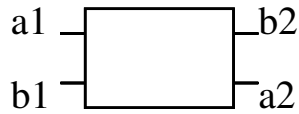
$$S_{11} = \frac{b1}{a1} \big|_{a2=0} \quad S_{12} = \frac{b1}{a2} \big|_{a1=0}$$

$$S_{21} = \frac{b2}{a1} \big|_{a2=0} \quad S_{22} = \frac{b2}{a2} \big|_{a1=0}$$

$$\begin{bmatrix} b1 \\ b2 \end{bmatrix} = \begin{bmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{bmatrix} \begin{bmatrix} a1 \\ a2 \end{bmatrix}$$

A Yelem device which is declared with a model named SPARAM is called an S-Parameter block.

Param name	iValue
IM	LC (Linear Cartesian)
	LC (Linear Cartesian)
	VF (Vector fitting)
	CS (Cubic spline)
	CSL(Cubic Spline Log)
EX	1 (Activate) or 0 (Disactivate)
SPARAMFILE	Filename of the SnP S-Parameter data
FORCE_PASSIVITY	1(Activate or 0 (Disactivate)

15.3.14 Linear Dependent Sources

Voltage-Controlled Voltage Sources (Esrc)

Spice syntax

```
EXXXXXXX N+ N- NC+ NC- VALUE
```

Examples:

```
E1 2 3 14 1 2.0
```

N+ is the positive node, and N- is the negative node. NC+ and NC- are the positive and negative controlling nodes, respectively. VALUE is the voltage gain.

Current-Controlled Current Sources (Fsrc)

Spice syntax

```
FXXXXXXX N+ N- VNAM VALUE
```

Examples:

```
F1 13 5 VSENS 5
```

N+ and N- are the positive and negative nodes, respectively. Current flow is from the positive node, through the source, to the negative node. VNAM is the name of a voltage source through which the controlling current flows. The direction of positive controlling current flow is from the positive node, through the source, to the negative node of VNAM. VALUE is the current gain.

Voltage-Controlled Current Sources (Gsrc)

Spice syntax

```
GXXXXXXX N+ N- NC+ NC- VALUE
```

Examples:

```
G1 2 0 5 0 0.1MMHO
```

N+ and N- are the positive and negative nodes, respectively. Current flow is from the positive node, through the source, to the negative node. NC+ and NC- are the positive and negative controlling nodes, respectively. VALUE is the transconductance (in mhos).

Current-Controlled Voltage Sources (Hsrc)

Spice syntax

```
HXXXXXXX N+ N- VNAM VALUE
```

Examples:

```
HX 5 17 VZ 0.5K
```

N+ and N- are the positive and negative nodes, respectively. VNAM is the name of a voltage source through which the controlling current flows. The direction of positive controlling current flow is from the positive node, through the source, to the negative node of VNAM. VALUE is the transresistance (in ohms).

15.3.15   Linear Independent Sources

Voltage Sources

Spice Syntax

```
VXXXXXXX N+ N- <DC >DC/TRAN Value> <AC <AC Mag < AC Phase>>>
```

Examples:

```
VCC 10 0 DC 6
```

```
VIN 13 2 0.001 AC 1 SIN(0 1 1MEG)
```

```
VIN 2 0 PWL 0 0 10n 0 11n 3.3 25n 3.3 26n 0 R
```

N+ and N- are the positive and negative nodes, respectively. Note that voltage sources need not be grounded.

Voltage sources, being used for circuit excitation, are the 'ammeters' for SPICE, that is, zero valued voltage sources may be inserted into the circuit for the purpose of measuring current. They of course have no effect on circuit operation since they represent short-circuits.

DC/TRAN is the dc and transient analysis value of the source. If the source value is zero both for dc and transient analyses, this value may be omitted. If the source value is time-invariant (e.g., a power supply), then the value may optionally be preceded by the letters DC.

ACMAG is the ac magnitude and ACPHASE is the ac phase. The source is set to this value in the ac

analysis. If ACMAG is omitted following the keyword AC, a value of unity is assumed. If ACPHASE is omitted, a value of zero is assumed. If the source is not an ac small-signal input, the keyword AC and the ac values are omitted.

Any independent source can be assigned a time-dependent value for transient analysis. If a source is assigned a time-dependent value, the time-zero value is used for dc analysis.

Pulse

```
PULSE(V1 V2 TD TR TF PW PER)
```
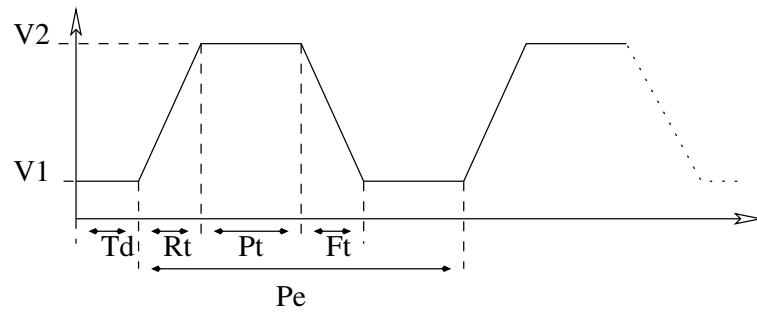

Exmples:

```
VIN 3 0 PULSE(-1 1 2NS 2NS 2NS 50NS 100NS)
```

parameter	default value	units
V1 (initial value)	-	Volts or Amps
V2 (pulsed value)	-	Volts or Amps
TD (delay time)	0.0	seconds
TR (rise time)	TSTEP	seconds
TF (fall time)	TSTEP	seconds
PW (pulse width)	TSTOP	seconds
PER(period)	TSTOP	seconds

A single pulse so specified is described by the following table:

time	value
0	V1
TD	V1
TD+TR	V2
TD+TR+PW	V2
TD+TR+PW V2	V1
TSTOP	V1

Intermediate points are determined by linear interpolation.

Sinusoidal

```
SIN(VO VA FREQ TD THETA)
```

Examples:

```
VIN 3 0 SIN(0 1 100MEG 1NS 1E10)
```

parameters	default value	units
VO (offset)		Volts or Amps
VA (amplitude)		Volts or Amps
FREQ (frequency)	1/TSTOP	Hz
TD (delay)	0.0	seconds
THETA (damping factor)	0.0	1/seconds

The shape of the waveform is described by:

time	value
0 to TD	V0
TD to TSTOP	$V0 + VA \exp^{\frac{-(t-TD)}{THETA}} \sin(2\pi * Freq(t + TD))$

Exponential

```
EXP(V1 V2 TD1 TAU1 TD2 TAU2)
```
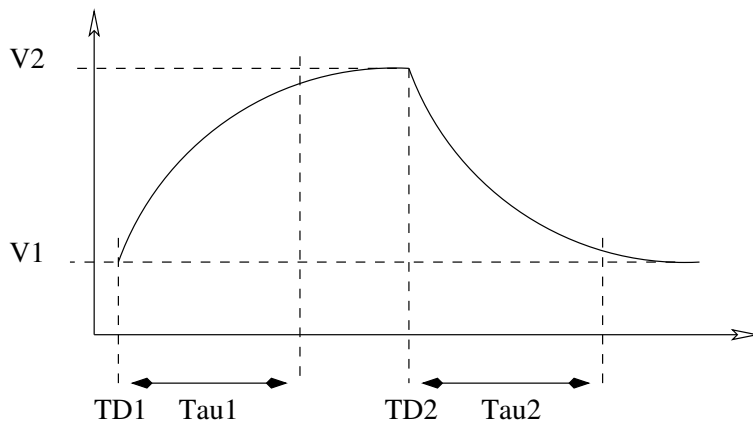

Examples:

```
VIN 3 0 EXP(-4 -1 2NS 30NS 60NS 40NS)
```

parameter	default value	units
V1 (initial value)	-	Volts or Amps
V2 (pulsed value)	-	Volts or Amps
TD1 (rise delay time)	0.0	seconds
TAU1 (rise time constant)	TSTEP	seconds
TD2 (fall delay time)	TD1+TSTEP	seconds
TAU2 (fall time	TSTEP	seconds

The shape of the waveform is described by the following table:

time	value
0 to TD1	V1
TD1 to TD2	$V1 + (V2 - V1)[1 - \exp^{\frac{-(t-TD1)}{Tau1}}]$
TD2 to Tstop	$V1 + (V2 - V1)[1 - \exp^{\frac{-(t-TD1)}{Tau1}}] + (V1 - V2)[1 - \exp^{\frac{-(t-TD2)}{Tau2}}]$

Piece-Wise Linear

```
PWL(T1 V1 <T2; V2 T3 V3 T4 V4 ...>)
```
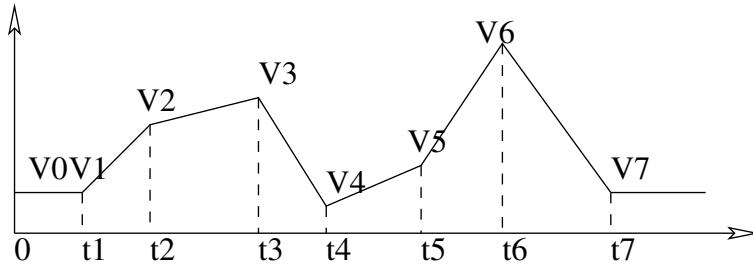
Examples:

```
VCLOCK 7 5 PWL(0 -7 10NS -7 11NS -3 17NS -3 18NS -7 50NS -7)
```

Each pair of values (Ti, Vi) specifies that the value of the source is Vi (in Volts or Amps) at time=Ti. The value of the source at intermediate values of time is determined by using linear interpolation on the input values.

Pattern Function

```
PATTERN VHI VLO < TDELAY TRISE TFALL TSAMPLE BITS R >
```

Examples:

```
VPATTERN 1 0 PATTERN 5 0 0 10p 10p 5n 0 1 1 R
```

Generating a digital like source. The VHI/VHO are used to select the voltages of the digit and the pattern is used for representing the signal.

Current Sources

Spice Syntax

```
IYYYYYYY N+ N- <<DC <DC/TRAN Value>> <AC <AC Mag <AC Phase>>>
```

Examples:

```
ISRC 1 0 1m
```

N+ and N- are the positive and negative nodes, respectively. Positive current is assumed to flow from the positive node, through the source, to the negative node. A current source of positive value forces current to flow out of the N+ node, through the source, and into the N- node.

DC/TRAN is the dc and transient analysis value of the source. If the source value is zero both for dc and transient analyses, this value may be omitted. If the source value is time-invariant (e.g., a power supply), then the value may optionally be preceded by the letters DC.

ACMAG is the ac magnitude and ACPHASE is the ac phase. The source is set to this value in the ac analysis. If ACMAG is omitted following the keyword AC, a value of unity is assumed. If ACPHASE is omitted, a value of zero is assumed. If the source is not an ac small-signal input, the keyword AC and the ac values are omitted.

15.3.16 Hierarchy declaration

Block sub-circuit declaration

In Spice a block can be defined by using the keyword .SUBCKT to define a block or a black box. The syntax of the declaration is as follows:

```
.SUBCKT BNAME PIN1 PIN2 ... PINn <Param1=value> .. <Paramn=value>
* Circuit Description of the subckt *
.ENDS
```

Between these two keywords (.SUBCKT and .ENDS), you can can define any devices or instances or redefine new blocks.

For example, an inverter can be declared as :

```
.SUBCKT INVERTER IN OUT VDD GND
.MODEL TN NMOS level=3
.MODEL TP PMOS level=3
* MOS instantiation D G S B
m1 OUT IN GND GND TN w=0.25u l=0.25u
m2 OUT IN VDD VDD TP w=0.250u l=0.25u
.ENDS
```

Optional parameters can be declared to set some parameters which are considered as new parameters for the scope of the block.

If a parameter MYPAR=1 is declared on the block definition, it means that inside the block MYPAR is equal to 1, but outside it is not defined.

Block sub-circuit instantiation

In Spice, an instance of a block can be defined by using the keyword X/x using the line syntax below:

```
Xname NODE1 NODE2 ... NODEn SUBCKTNAME <Param1=value> .. <Paramn=value>
```

Optional parameters can be used to overwrite the parameters that have been defined for the "SUBCK-TNAME" block definition.

15.3.17   Parameters

It is possible to define parameters through a command line as follow:

```
.PARAM PARAMNAME=VALUE
```

This syntax can be used as a define statement. It is used when a generic value has to be defined several times.

15.3.18   S-Parameter extraction

Altair Spice solver offers the possibility to extract the S-Parameter data of a circuit. S-Parameters are considered to be the ratio measured between the inflective and the reflective waveforms on each ports of the inputs of the S-Parameter block. The scheme is the following one:
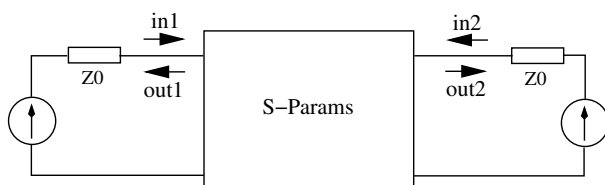
The first step of the S-Parameter extraction is to select the ports inside the circuit that we will be taken into account to make the extraction.

To select a port, you have to add a voltage port declaration. This is needed to clearly select the nodes for the S-Parameter block input/output. For example, you could use following syntax:

```
V<n> PIN_NAME 0 iport=<NUM PORT> rport=<Z0>
```

You have to include a Voltage port declaration for each port number of the S-Parameter block. The 'n' number is just an increasing number. The PIN_NAME value is the circuit node on which a measure will be made and will be considered as a port of the S-Parameter block.

The "iport" is the index port number of the port which will be seen in S-Parameter results.

Rport parameter is the parameter used to specify the impedance that is used to make the measures. By default this number has to be equal to 50 ohms.

These Voltage ports are needed to specify the port input/outputs. You also have to specify the frequency list that will be used to make the simulation. For this, you have to set the AC simulation and provide the frequency range of the simulation using the command:

```
.AC DEC <Nb frequency divider> <Starting frequency> <Stop frequency>
```

You have to add the following command to produce the results as outputs of the several simulations

```
.SEXTRACT <output filename>
```

The command will inform the simulator that results of S-Parameter extraction will have to be put on the filename called "output filename". The file will contain data as for an example of 2x2 ports S-Parameter block:

```
\# Hz S MA R 50 ! S(1,1) S(2,1) S(1,2) S(2,2)
1.0000000000000000E+00 3.3336841518187355E-01 5.8793299979049041E-01 ...
1.2589254117941673E+00 3.3338893175136475E-01 7.4011447062501234E-01 ...
...
```

15.3.19 Options

Setting DC algorithm

```
.option DCALGO=<GMIN | GRAMP | VRAMP>
```

Force the DC solver to use a specific algorithm for operating point computation.

GMIN is a method to add a impedance on nodes in order to facilitate convergence

GRAMP and VRAMP are methods to facilitate convergence on increasing voltage on sources

Force an Initial condition value

The command .IC can be used to force an initial condition on a node.

```
.IC V(node)=0.5
```

For example on the line below, the command forces the simulator to use an initial value of 0.5 volt for the node "node".

```
.NODESET V(node)=<value>
```

Performs a 2-pass DC analysis :

- First pass ac .IC (force initial conditions)
- Second pass starts with first pass solution but stops enforcing initial conditions.

`.GUESS V(node)=<value>`

Performs first DC iteration with specified initial conditions, but subsequent iterations are made without .GUESS initial conditions.

Fixing step (STEP)

`.option STEP=value`

This option can be used to force the simulator to make fixed step for the transient simulation. It should be used on certain conditions as it is changing the accuracy of the results. By default the fixed step is disabled.

Setting a max step (HMAX)

`.option HMAX=value`

By default the simulator is using a variable time step which is internally determined according to the step computation. In some conditions the user can force the simulation to take a maximum value for a step computation.

Setting a relative tolerance (RELTOL)

`.option RELTOL=value`

This option can be used to change convergence tolerances for all devices. The default value is 1e-3

Setting Simulation Temperature

`.option TEMP=value`

This option can be used to change the temperature of the simulation. By default the temperature is set to 300.15 K.

.option VNTOL

`.option VNTOL=value`

This option can be used to change the convergence tolerance for node voltages only. The default value is set to 1e-6V

.option ITOL

`.option ITOL=value`

This option can be used to change the convergence tolerance for node voltages only. The default value is set to 1e-5A

.option BE

```
.option BE
```

This option can be used to change the integration scheme method. By default Altair Spice solver is managing the integration method internally according to the solver results. It is possible to fix it to a more simple scheme like BE, to facilitate convergence on some circuits.

15.4 Known Issues

```
Error: Warning: No ground node is defined
```

This message means that ground node (implicit node '0') isn't connected to any device, resulting in an unsolvable equation set. At least one device should be connected to ground node. The user should then declare at least the following command:

```
Vgnd gnd 0 0
```

15.5 SpiceCustomBlock

In **Twin Activate** the user can enter and simulate Spice netlists using the `SpiceCustomBlock` block 15.1 located in `CustomBlocks` palette.

Figure 15.1: SpiceCustomBlock available in CustomBlocks palette.

In the block GUI the netlist, as well as input and output nodes of the netlist, can be defined. See Fig. 15.2.
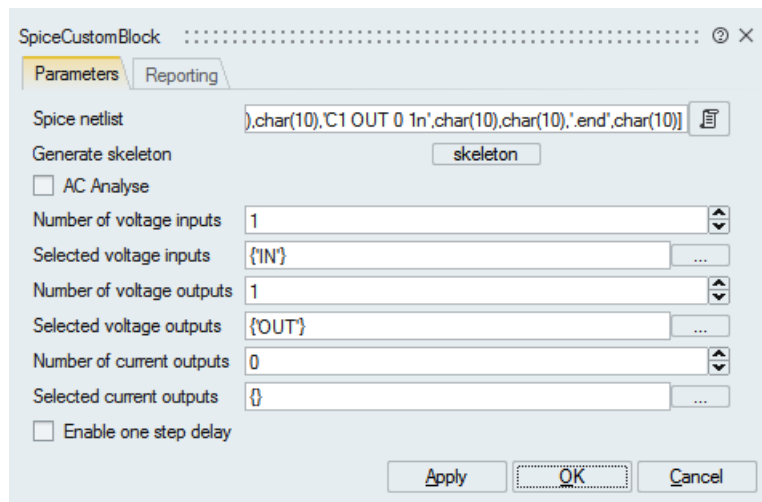
Figure 15.2: GUI of the SpiceCustomBlock block.

Netlist editor

In SpiceCustomBlock, the user can edit the Spice netlist. In order to open the editor, click on the icon ☑ . The netlist editor (15.3) opens and the user can edit the Spice netlist.
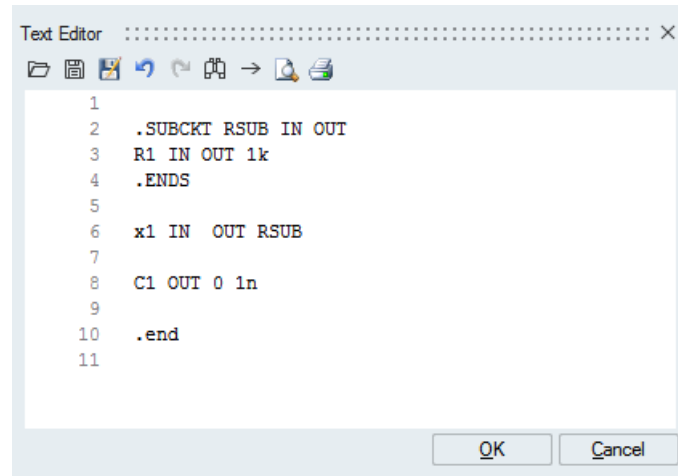
Figure 15.3: Netlist editor.

As an example, suppose we need to simulate a simple RC circuit, as shown in Fig.15.4.
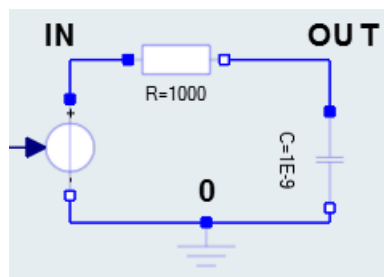
Figure 15.4: The simple RC circuit to be simulated using SpiceCustomBlock.

The corresponding netlist of this circuit is shown in Fig.15.3. This circuit is composed of two components **R** and **C** and three nodes **IN, OUT, 0**. The input signal voltage is supposed to be defined by standard **Twin Activate** blocks. In order to allow communication between Spice netlist and other standard **Twin Activate** blocks, input and output nodes of the netlist should be selected. click on the icon ▭▭ , to display the port selection GUI, as shown in Fig.15.5.
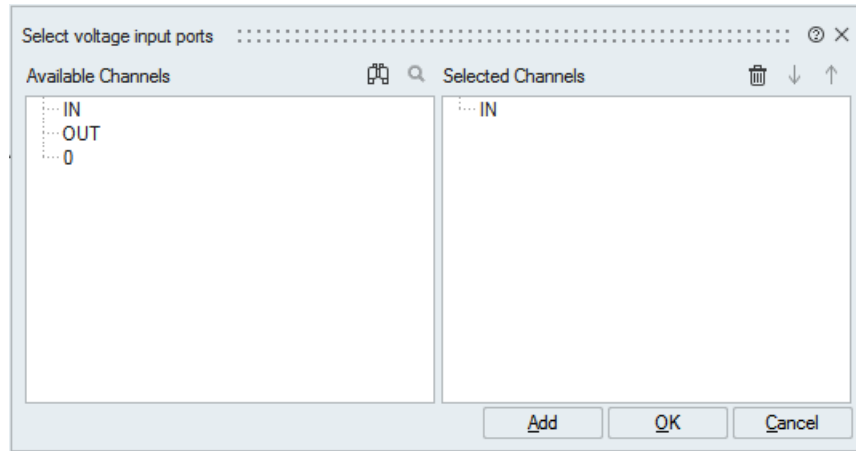
Figure 15.5: input/output port selection interface.
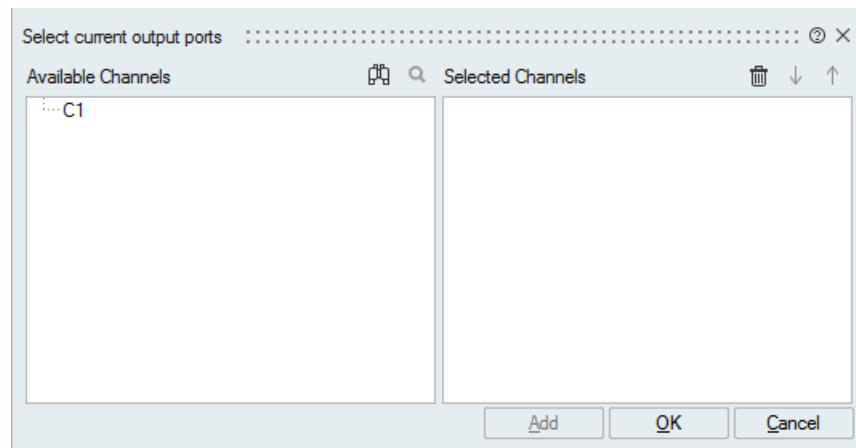
Figure 15.6: Selection of components for current output port.

Input/output port selection

SpiceCustomBlock allows for choosing three types of input and output ports.

1. input voltage port: If a node is chosen as input, the voltage of that node is provided by the corresponding input port of the SpiceCustomBlock. In the RC example, the port **IN** is an input, see Fig.15.5.

2. output voltage port: If a node is chosen as output, the voltage signal of that node is available at the corresponding output port of the SpiceCustomBlock. This signal is then available inside **Twin Activate** for example to be displayed by Scopes. In the RC example, the port **OUT** can be chosen as an output port to be displayed.

3. output current port: The block can also provide the current passing through components. In order to choose current signal, the corresponding component should be chosen, see Fig.15.6. For example in order to choose the current passing through a resistor, that resistor should be selected in the GUI.

Once input and output voltage nodes are selected, click on the OK button and the corresponding input

and output ports will be added to the block.

A complete model for the RC can be made by defining the input signal, *i.e.,* the voltage signal at node **IN**, and connecting the output port of the SpiceCustomBlock to a Scope block. In order to compare the results, the circuit corresponding to the RC circuit is made with Modelica blocks in **Twin Activate** and is displayed in the Scope, as shown in Fig.15.7. The circuit is excited by a sinusoid signal. The output voltage is shown in Fig.15.8. Note that output voltages of the Spice netlist and the Modelica RC circuit are identical and superposed in Fig.15.8.
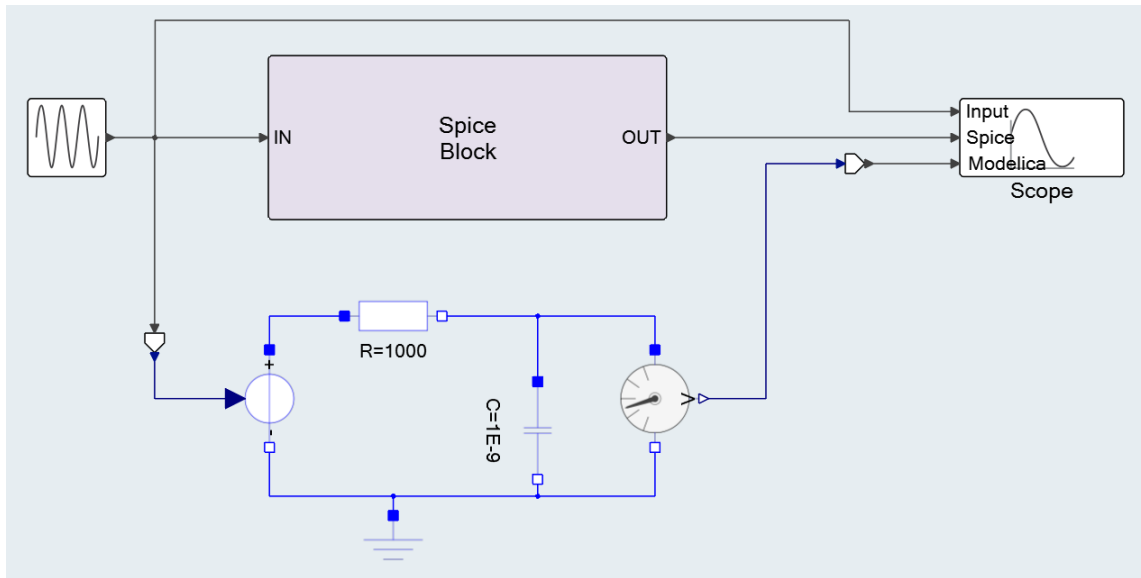
Figure 15.7: The complete Spice model and circuit counterpart in Modelica.
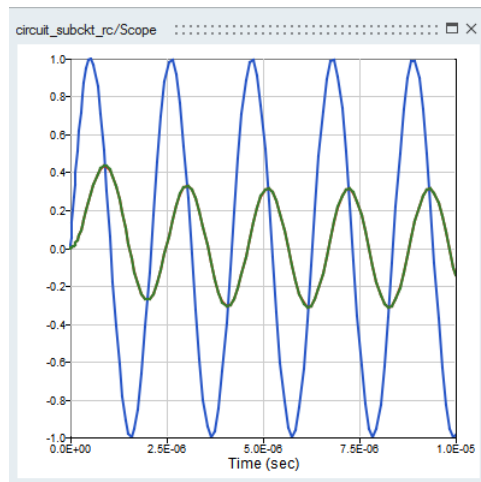
Figure 15.8: Simulation result of 15.7 in **Twin Activate**.

AC analysis

SpiceCustomBlock has an important feature to perform the frequency analysis of a Spice netlist. In order to obtain the frequency response of a Spice netlist, check on the **AC Analysis** checkbox in the

block GUI, see Fig.15.2. If this checkbox is selected, an input Frequency port is added to the block. This input defines the frequencies at which the Spice circuit should be excited. For each frequency the output frequency response is computed. Note that in the AC analysis the input and output ports are complex numbers.

Consider this netlist representing an low-pass filter circuit.

```
*LPFILTER.CIR - SIMPLE RC LOW-PASS FILTER
*
VS 1 0 AC 1 SIN(1  1    2KHZ)
*
R1 1 2 1K
C1 2 0 0.032UF
*
* ANALYSIS
.AC  DEC  5 10 10MEG
.TRAN  5US  500US
*
* VIEW RESULTS
.PRINT AC  VM(2) VP(2)
.PRINT TRAN  V(1) V(2)
*
.PROBE
.END
```

In order to obtain the frequency response of this circuit, a **Twin Activate** model is built, as shown in Fig.15.9.
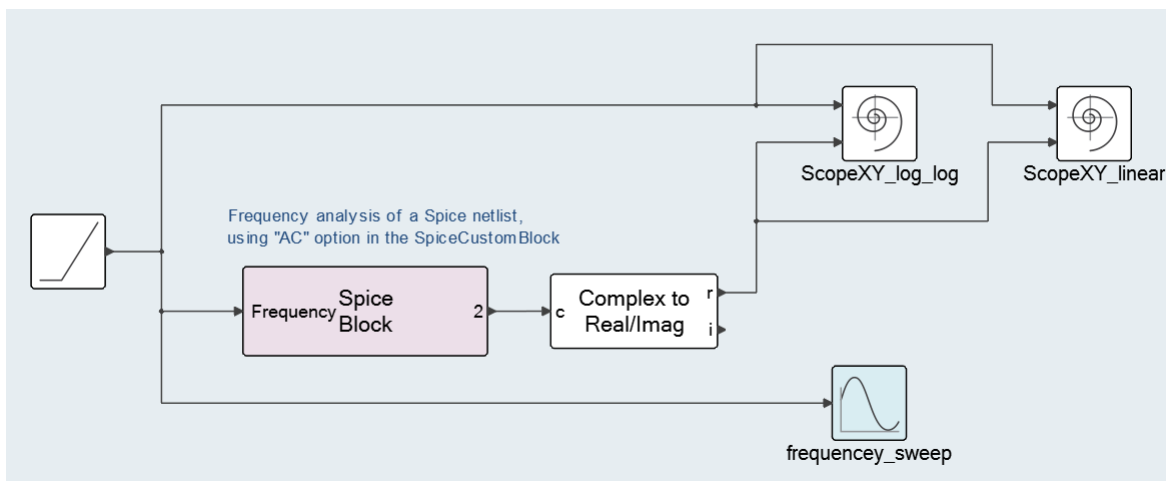
Figure 15.9: A **Twin Activate** model to compute the frequency response of a Spice netlist.

The SpiceCustomBlock block GUI is shown in Fig.15.10. The block has one only output port (*i.e.,* node 2). The block has no input port.
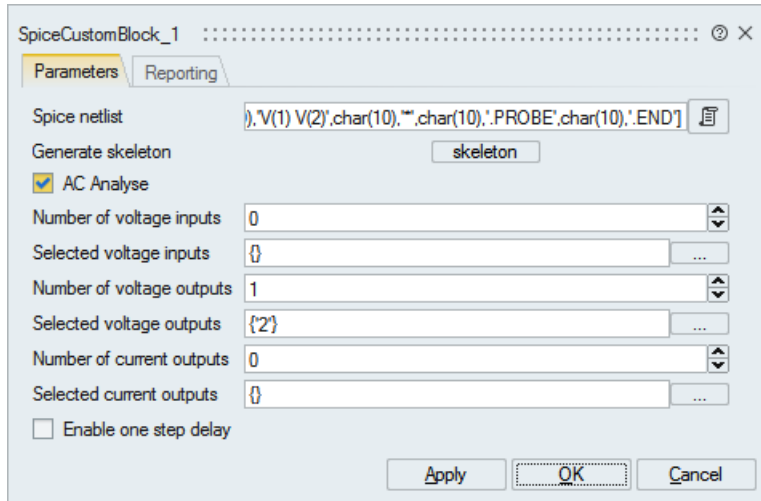
Figure 15.10: A **Twin Activate** model to compute the frequency response of a Spice netlist.

A **Ramp** block has been used to sweep the frequency. The frequency is started from 1Hz and is increased with a rate of 1KHz/Sec. The output of the ramp block is connected to the Frequency input of the SpiceCustomBlock block. The filter output (*i.e.,* node 2) is complex, so a **Complex to Real/Imag** block is used to separate the complex and imaginary parts of the frequency response. The real part of output frequency response with respect to the input frequency is shown in Fig.15.11. Note that frequency axis is display on a logarithmic scale.
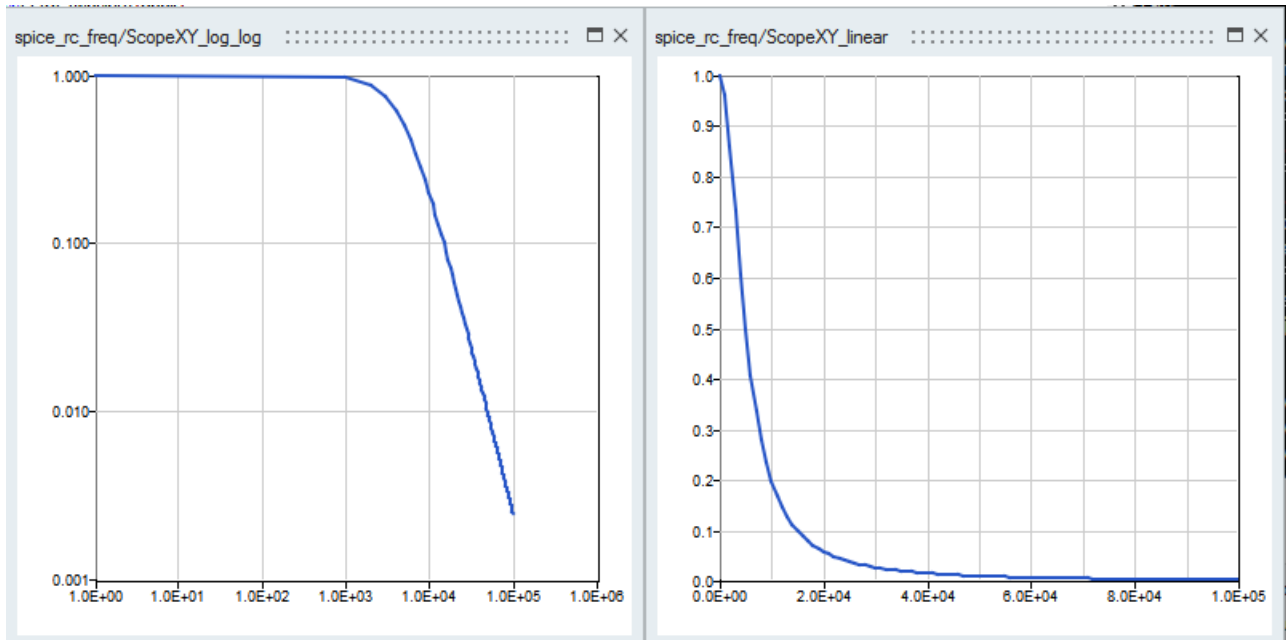
Figure 15.11: Frequency response of the low-pass filter for the model in Fig.15.9. Left plot is in Log-Log scale and the right plot is Linear-Log scale

Reporting

When a simulation of a Spice netlist fails, it is helpful to generate a logfile to find out the reason of the failure. In order to create a logfile, click on the **Reporting** tab in the block GUI as shown in Fig.15.12.
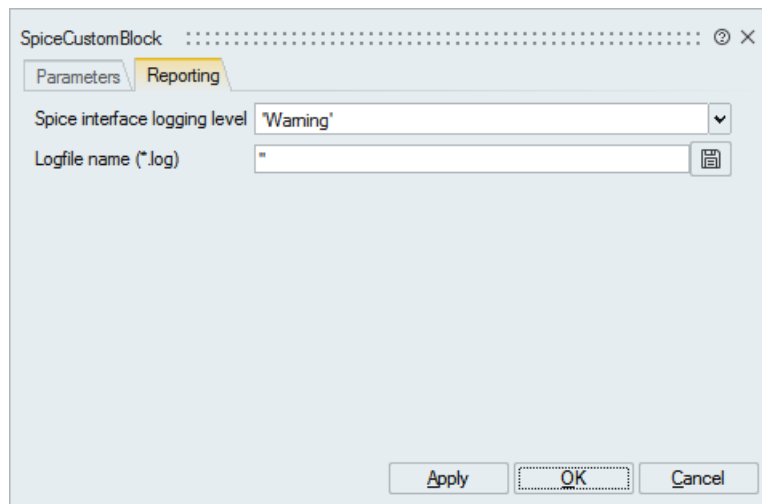
Figure 15.12: GUI of the SpiceCustomBlock block for the report generator.

In the Spice interface logging level, the user can choose one of `Nothing, Error, Warning, Info`.

1. `Nothing`: No message is logged at all. This turns off the logger.

2. `Error`: Only error messages are logged.

3. `Warning`: Only Warning and error messages are logged.

4. `Info`: Every message emitted by the Spice solver is logged.

The log filename should be specified. If a relative path is used, the file is created next to the model.

Chapter 16

Co-Simulation with Altair PSIM

16.1   Introduction

The **Twin Activate**-**PSIM** interface block provides the co-simulation interface between **PSIM** and **Twin Activate**. The objective of this section is to show the way the **PSIM** block is set up in **PSIM** and **Twin Activate**. A simple buck converter with current feedback control, as shown in 16.1 below in the **PSIM** environment, will be used as the example. In this circuit, the inductor current is measured and compared with a reference. The error signal is sent to a PI controller, and the PI output is compared with a carrier waveform to generate the gating signal for the switch in the power circuit. In this section we will illustrate the way this circuit should be splitted to two parts: the control circuit (as shown below inside the dotted rectangle) to be simulated in **Twin Activate** and the rest of the circuit to be simulated in **PSIM**.
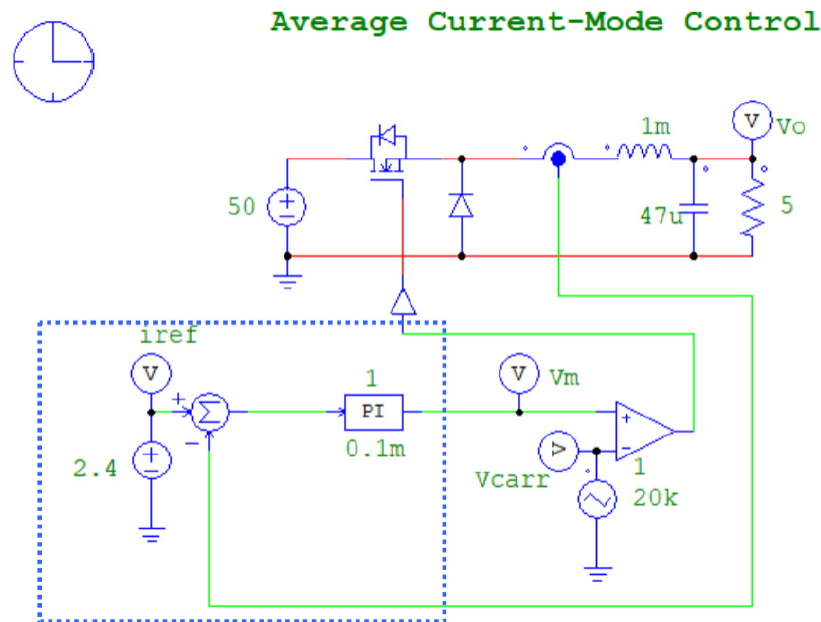
Figure 16.1: Average current-mode control simulated entirely inside **PSIM**.

16.2 Setting up the model in PSIM for Co-Simulation

Below are the steps to set up the co-simulation in **PSIM**:

1. The Cosimulation with **Twin Activate** uses the latest **PSIM** installation build available in the Windows registry.

2. Launch **PSIM**, and open the file "chop1q_ifb.psimsch". The file can be found in the sub-directory `examples\SimCoupler` in the **PSIM** directory.

3. Save the file to a different name, such as "chop1q_ifb_psim.psimsch", in the directory `c:\test`.

4. Modify the circuit by deleting the reference source, the summer, and the PI controller. After the modification, the circuit looks as follows in 16.2:
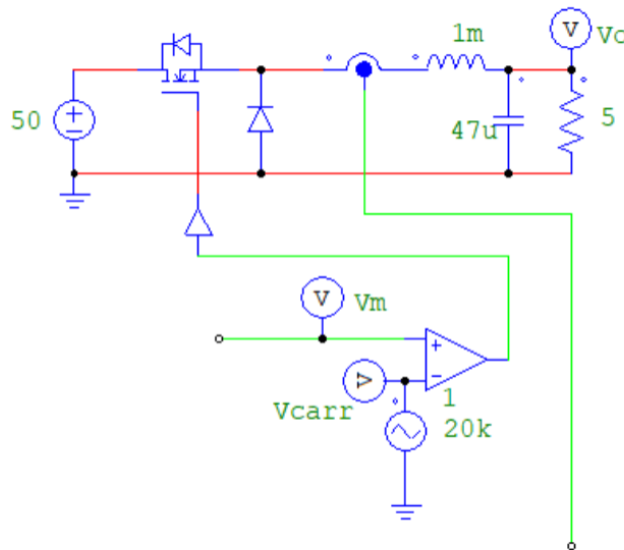
Figure 16.2: The eletrical circuit part to be simulated inside **PSIM**.

5. In order to add the communication port between **Twin Activate** and **PSIM**, **In Link Node** and **Out Link Node** are needed. These block can be found in `Elements » Control » SimCoupler Module` pull down menu in **PSIM**. Click on these block and choose the name for input and output signals, as shown in 16.3.
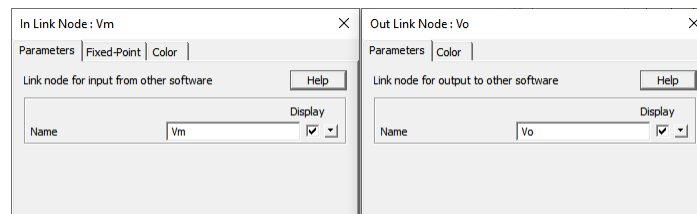
Figure 16.3: **In Link Node** and **Out Link Node** in **PSIM**.

6. Select the **Out Link Node** and connect it to the current sensor output, and rename it to **iL**.

7. Similarly, select the **In Link Node** and connect it to the comparator input, and rename it to **Vm**.

8. The PSIM block in **Twin Activate** uses the Link nodes to establish interface between **PSIM** and **Twin Activate**. In Link Nodes receive values from **Twin Activate**, and Out Link Nodes send values to **Twin Activate**. Multiple In/Out Link Nodes can be used in a circuit to exchange values between **PSIM** and **Twin Activate**. In this case, for example, we are going to measure and send the load voltage to **Twin Activate** by connecting a voltage sensor across the load resistor and placing an Out Link Node at the voltage sensor output. The Out Link Node will be renamed as **Vo**. After this, the circuit will look as below in 16.4.
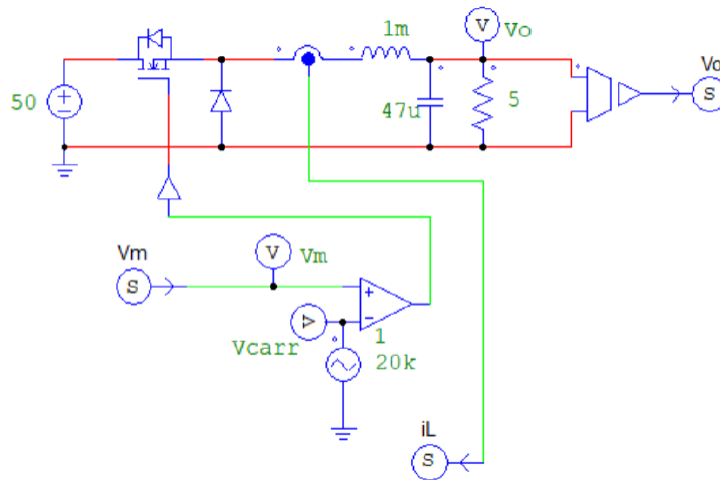
Figure 16.4: The eletrical circuit augmented with interface In/Out link blocks inside **PSIM**.

9. If there are more than one In Link Node or Out Link Node (such as in this case), you may wish to arrange the order of the link nodes. Go to `Simulate » Arrange SLINK Nodes`, and a dialog window will appear as shown below.
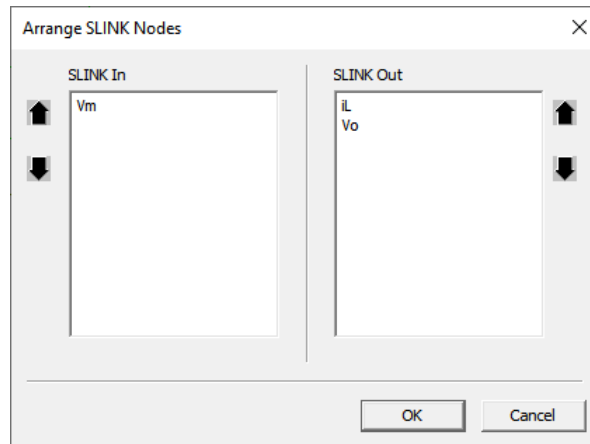
Figure 16.5: Changing the order of inputs and outputs.

Arrange the order of the In nodes and Out nodes to be the same as how the input/output ports would appear in the PSIM block in **Twin Activate**. The order of the ports is from the top to the bottom. In this case, the output port corresponding to **iL** will be on the top, and the output port

corresponding to **Vo** will be on the bottom. To re-arrange the node sequence, highlight the name of the node, and click on the up or down arrow.

10. Save the schematic file in **PSIM**. In this example, the file will be saved to

    `c:\test\chop1q_ifb_psim.psimsch`. At this point, the setup in **PSIM** is complete, and we will start the setup in **Twin Activate**.

11. Launch **Twin Activate** and open a new model

12. Drag and drop a PSIM block in the CoSimulation palette, as shown in 16.6.
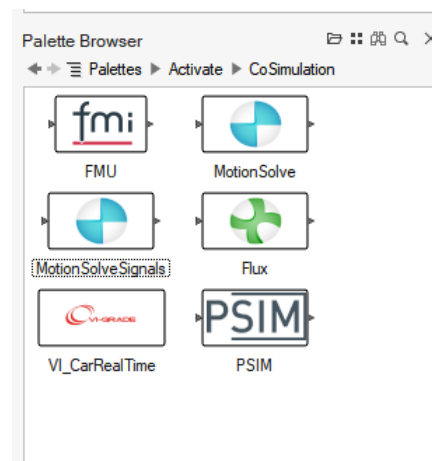
Figure 16.6: Cosimulation palette of **Twin Activate**

13. Click on the **PSIM** block and choose the `chop1q_ifb_psim.psimsch` file, as shown in 16.7. Once the file is chosen, the input and output names, as well as proposed fixed-step time and final times fields are populated automatically. If you change the **PSIM** model, you need to press the `reload` button to repopulate the fields. The model can be directly accessed and modified by clicking on the PSIM buttton in the GUI.
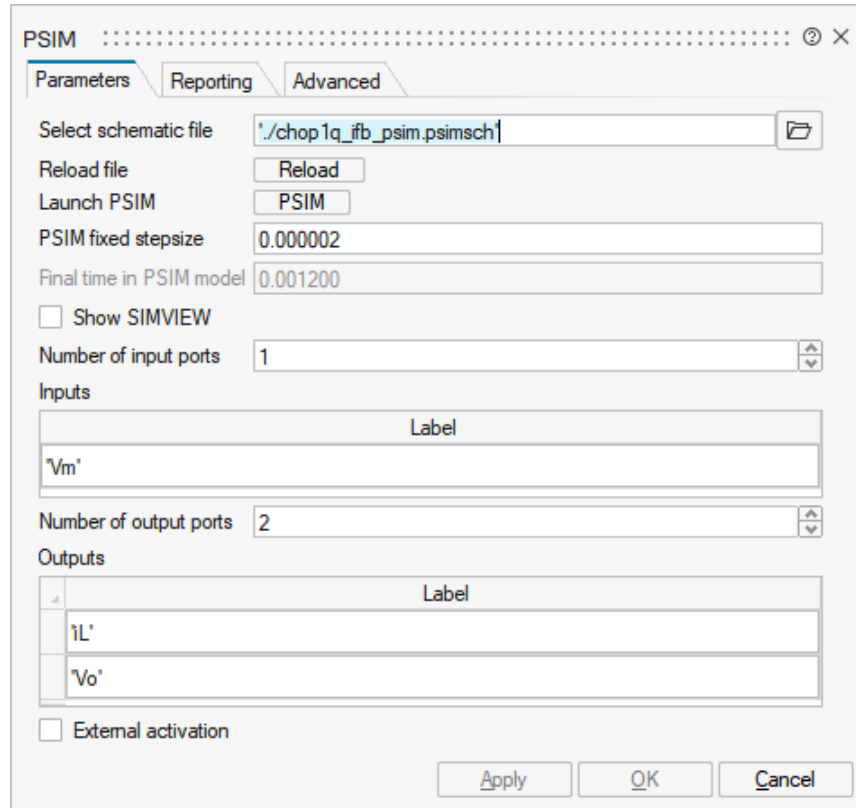
Figure 16.7: The PSIM block GUI in **Twin Activate**

14. Once the PSIM model is selected and the fields are populated, clock on OK to close the GUI. Now the PSIM block should look like the figure in 16.8 and should have the same number of input and outputs in the **PSIM** environment.

Figure 16.8: The PSIM block in **Twin Activate**

15. Now a control model corresponding to the control circuit that was deleted from the **PSIM** circuit in Step 4, should be built in **Twin Activate**, as shown in 16.9.
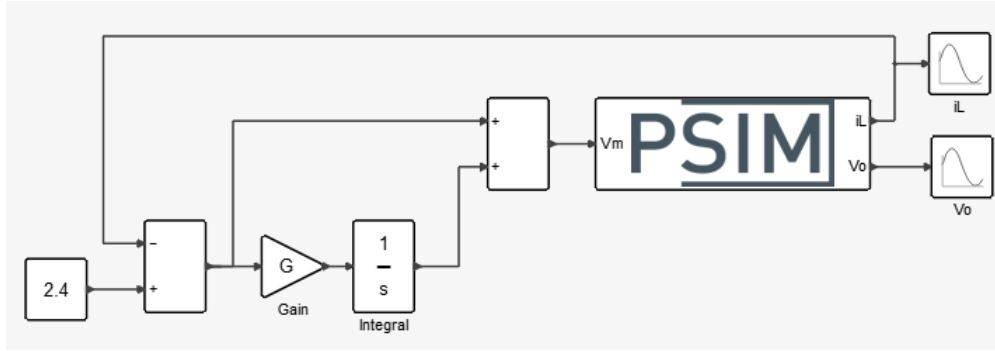
Figure 16.9: The complete PSIM model in **Twin Activate**

16. Once the model in **Twin Activate** is finished, change the **Twin Activate** simulation setup and choose the appropriate simulation time and step-size. Both fixed-step and variable step size solver can be used. Now cosimulation can be started. The simulation result are shown in Scopes as shown in 16.10.
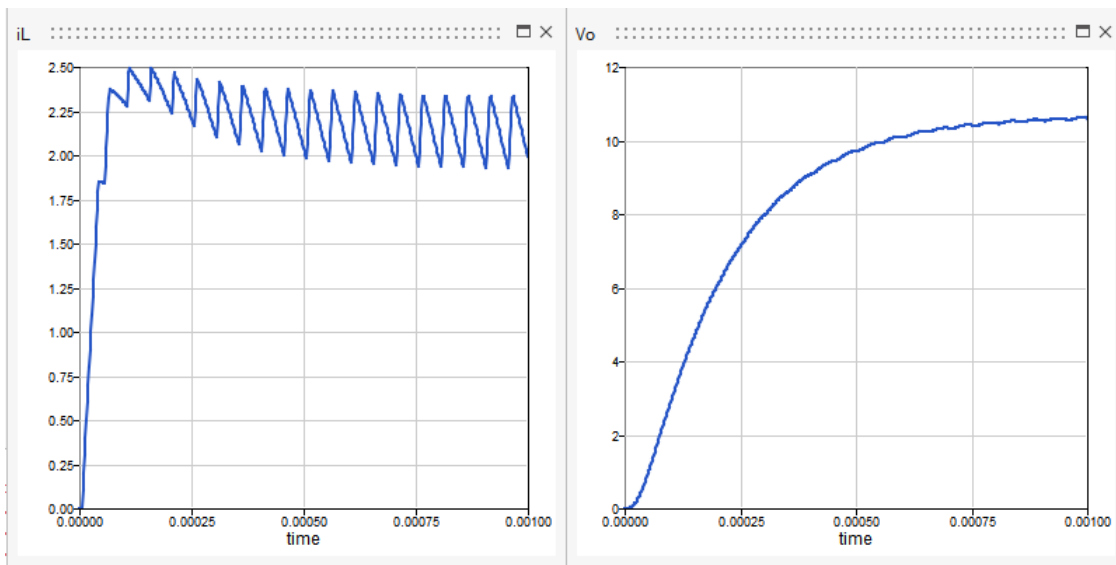
Figure 16.10: The co-simulation result in **Twin Activate**

16.3   Passing parameters from Twin Activate to PSIM

Sometimes, it is usefull to use parameters in **PSIM** instead of numerical values. The value of these parameters can be defined and chnaged in **Twin Activate**. In order to set some parameter values in **Twin Activate** and pass them to **PSIM**, in the PSIM schematic file, change the desired value to a variable name. For example, to set the inductance of the inductor `L1` in **Twin Activate**, you can also change it to *e.g.*, `varL1`, as shown in 16.11.

Choose the number of variables that should be passed to **PSIM** and provide their name and their values in the table.
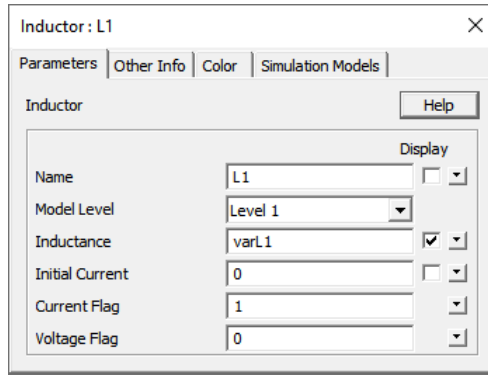
Figure 16.11: Define parameters **PSIM**.

In **Twin Activate**, double click on the PSIM block to open the GUI, and click on the `Advanced` tab, as shown in 16.12. Choose the number of variables that should be passed to **PSIM** and provide their name in **PSIM** and their values in the table.
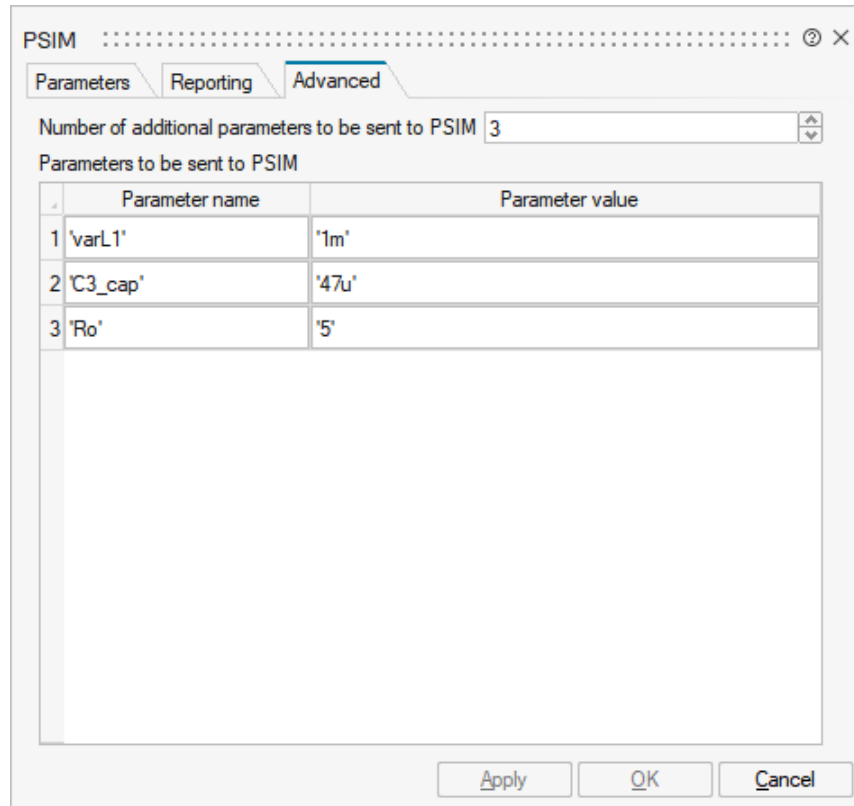
Figure 16.12: Define parameters in **Twin Activate** and pass their values them to **PSIM**.

The value of parameters can be any string variable readable by **PSIM**. The values may also be variable in **Twin Activate**.

Demo models `chop1q_ifb_Buck.scm` and `chop1q_ifb_psim.psimsch` in **Twin Activate** Cosimulation/PSIM demonstrate this feature.

16.4 Reporting

The PSIM block can generate a logfile and all inputs and output values communicated between **Twin Activate** and **PSIM** will be logged. This is very useful in case of trouble, but it slowsdown the co-simulation.

The logfile can also be `'stdout'` or `'stderr'`

Bibliography

Altair Twin Activate 2023   Extended Definitions 332

Proprietary Information of Altair Engineering Inc.

Chapter 17

Hybrid Automata in Twin Activate

17.1   Abstract

Hybrid automaton is a standard model for describing a hybrid system. A hybrid automaton is a state machine augmented with differential equations and is generally represented by a graph composed of vertices and edges where vertices represent continuous activities and edges represent discrete transitions. Modeling a hybrid automaton with large number of vertices may be difficult, time-consuming and error prone using standard modules in modeling and simulation environments such as **Twin Activate**. In this section, we present the `Automaton` block used for modeling and simulation of hybrid automata.

17.2   Introduction

Hybrid systems define a very wide class of dynamical systems that involve the interaction of heterogeneous data types and dynamics especially the interaction of continuous-time dynamics described by differential equations, with discrete dynamics described by a finite state under automata or other models of computation. Examples include mechanical systems with collisions, circuits with ideal diodes and switches, chemical processes controlled by valves or pumps, and most importantly, embedded computation systems where digital devices interact with an analog environment.

Early works on formal models for hybrid systems includes phase transition systems and hybrid automata. A hybrid automaton is a state machine augmented with differential equations. It is a standard model for describing a hybrid system. Hybrid automata come in several forms: The Alur-Henzinger hybrid automaton is a popular model that was developed primarily for algorithmic analysis of hybrid systems model checking [1, 2, 3]. In this document we will explain the way **Twin Activate** environment can be used to model and simulate hybrid systems, and in particular, hybrid automata.

Hybrid automata have been introduced by Xavier Nicollin *et al.* in [5] and an analysis of linear hybrid automata are given in [2, 4]. They modeled a hybrid system as a finite automaton that is expanded with a set of real valued variables. These variables can be tested and modified at transitions. At an automaton mode the value of variables change continuously with time according to evolution laws which are associated with the mode. The transition relations are specified by guarded commands; the activities by differential equations; and the invariants by logical formulas. We now present the relevant definition of a hybrid automaton from Henzinger's theory [4] that will be used in the construction of our formal semantics for **Twin Activate** automaton block.

H is a tuple which is defined as:
$H = (V, E, X, F, Invariant, Initial, Jump)$, where

- V is the set of control modes $\{v_1, \ldots, v_M\}$ where M is the number of control modes;

- E is the set of control switches which identify the source mode and a destination mode during a transition.

- X is a set of real-valued variables $\{x_1, \ldots, x_N\}$ where N is the dimension of H. $\dot{X}$ is first time derivative of X;

- F is a predicate over $\{X, \dot{X}\}$ assigned to each control mode; $Invariant$ defines the admissible range for $\{X, \dot{X}\}$ in each mode;

- $Initial$ defines initial values of $\{X, \dot{X}\}$ in each mode.

- $Jump$ is an edge labeling function which assigns a predicate over $\{X, \dot{X}\}$ to each edge.

As an example consider the model of the bouncing ball in (17.1).

$$\begin{cases} \dot{x} & = & v \\ \dot{v} & = & -g \end{cases} \tag{17.1}$$

The model of this bouncing ball has been implemented in **Twin Activate** as shown in Fig. 17.1.
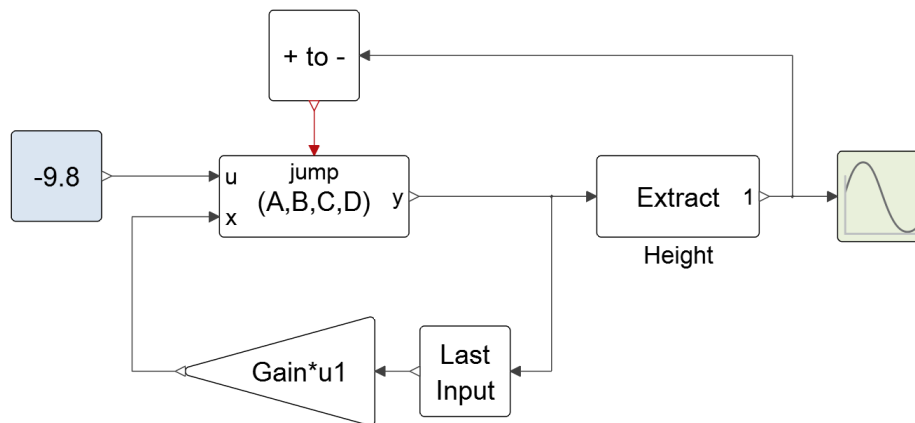
Figure 17.1: The model of a control system in **Twin Activate**

The output of the block `JumpStateSpace` is a vector of size two consisting of $[x, v]^t$. Whenever the condition $(x < 0)$ is satisfied, an event is generated by the `Zero-crossing` (**+ to -**) block. This event makes the `Jump` block do a reinitialization, *i.e.,* $v := -0.9v$ and $x := x$. In Fig. 17.1, the `Clock` block generates periodic events (activation signals) to activate the `Scope` block which displays x and v.

This hybrid system can also be modeled as an automaton with a single control mode. The graphical representation of this automaton is depicted in Fig. 17.2. In this model, there are two variables x and v whose initial values are given at $t = 0$. There are also two event sources; a zero-crossing event and a periodic time event. Only the zero-crossing event causes discontinuity in variables.

Although it is often possible to model hybrid automata with generic **Twin Activate** blocks, it is not an easy and efficient way to develop a hybrid automaton especially when the number of control modes is high. In a hybrid automaton composed of several control modes, at any time instant, only one mode or subsystem is active and the others should stay inactive. In **Twin Activate**, even though it is possible

$$x(0)$$
$$v(0)$$

$$v := -0.9v$$

$$\begin{cases} \dot{x} = v \\ \dot{v} = -g \end{cases}$$

e_2 : Clock e_1:if $(x < 0)$
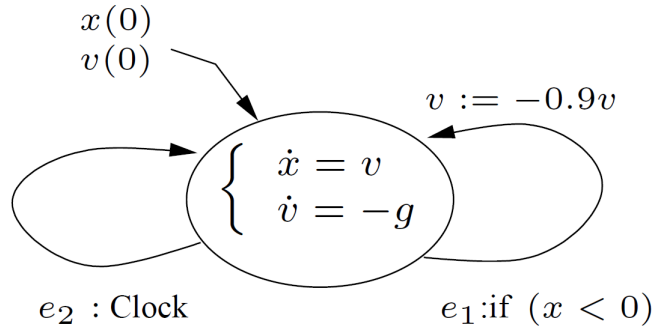
Figure 17.2: Automaton for the bouncing ball

to generate an activation signal to activate conditionally a subsystem via `If-Then-Else` blocks, the variables are still present in the state vector of the model and reduce the performance of the numerical solver. Another difficulty is in detecting the jumps; in each control mode of the hybrid automaton, zero-crossing functions should be evaluated for monitoring the jump conditions. If we develop a hybrid automaton with several subsystems activated by `If-Then-Else` blocks, all zero-crossing functions are active and it will be difficult to distinguish the zero-crossing functions that have to be monitored. Using switches and selector blocks, to activate or deactivate the zero-crossings, increases the model size and reduces the simulation speed.

Twin Activate is a hybrid system simulator which means that **Twin Activate** can simulate mixed continuous-time and discrete-time subsystems. Developing hybrid automata containing large number of modes and continuous-time states [using generic **Twin Activate** blocks] is a cumbersome, time-consuming, and difficult task. In this section, we present the `Automaton` block used for modeling and simulation of hybrid automata. This block provides a modular way to model a general hybrid automaton in **Twin Activate**. This block provides an interface to model and encode the graphical representation of an automaton to be used by the simulator and consequently by the numerical solver in a completely transparent way.

17.3 Automaton Block

The automaton block provides a switching mechanism between subsystems corresponding to control modes of the automaton. Subsystems are constructed in such a way that they receive the state vector as input from the automaton block and compute the flow and jump functions. The state variables are defined only in the automaton block and not in the subsystems.

Suppose that a hybrid automaton consists of M control modes. The continuous-time dynamics in mode 'i' is defined with DAE $(0 = F_i(\dot{x}, x, u))$ where $i \in \{1 \ldots M\}$ and the dimension of x is N $(N \geq 0)$ for any $i \in \{1 \ldots M\}$, as shown in Fig. 17.3. Suppose that in control mode 'i', there are Z_i jump conditions indicating jumps toward other modes. The jump conditions are defined by $Jump_{ij}(\dot{x}, x, u)$ functions where $j \in \{1 \ldots Z_i\}$. When a jump function becomes positive a mode transition will happen. When a transition to mode 'k' happens state vector x is reset to $Reset_k(\dot{x}, x, u)$, *i.e.,* $x_l = Reset_{kl}(\dot{x}, x, u)$ for $l \in \{1 \ldots N\}$. Note that the reset function is identical for all incoming edges of mode k. In order to develop an automaton containing a mode with multiple reset functions, the value of the current and previous active modes should be used. These values are available at the first output port of the block.

For such a hybrid automaton, we can use the `Automaton` block of **Twin Activate**. This block has the
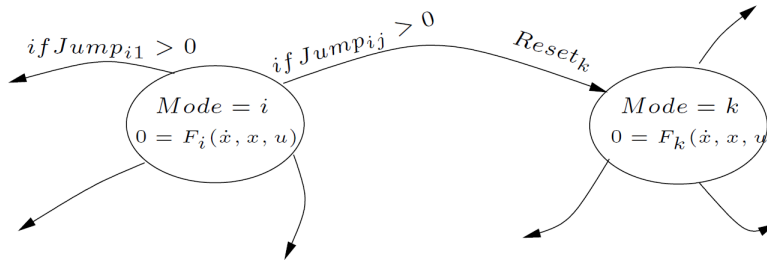
Figure 17.3: A graphical representation of an automaton.

following input/output ports, see Fig. 17.4.

- **Output 1:** The first output port is a vector of size two consisting of the current and the previous active control modes.

- **Output 2:** The second output port is a vector of size $2N$ providing the state vector and its first time derivative, *i.e.,* $[x, \dot{x}]^t$.

- **Inputs:** The automaton block has M vector input ports corresponding to M modes or subsystems of the automaton. The input port 'i' which is the output of the ith subsystem of the automaton defines the dynamic behavior in the control mode 'i'. This input port is a vector of size $2N + Z_i$, as indicated in (17.2). The diagram in Fig.17.4 shows the way subsystems are connected to the automaton block: Using an automaton block defined with M modes and N continuous-time states. The automaton block has therefore M input ports, two output ports of size 2 and $2N$, and an event output port.
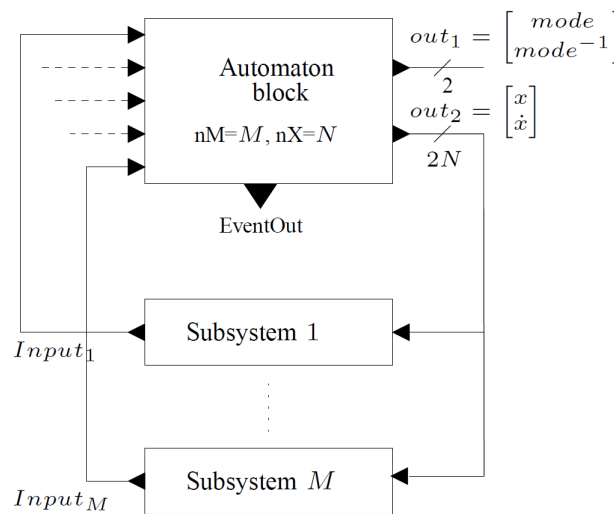
Figure 17.4: Automaton with M modes and N ports

$$Input_i = \begin{bmatrix} F_i(\dot{x}, x, u) \\ Reset_i(\dot{x}, x, u) \\ Jump_i(\dot{x}, x, u) \end{bmatrix} \qquad (17.2)$$

- The first N elements of the $Input_i$ are the continuous-time dynamics. The dynamics of the system in the control mode 'i' is described by a smooth index-1 DAE (*i.e.,* $0 = F_i(\dot{x}, x, u, t)$).

- The next N elements of $Input_i$ are the values used to reset the continuous-time states when a transition <u>to</u> control mode 'i' is activated.

- The next Z_i elements of $Input_i$ are the jump or zero-crossing functions. If the jth zero-crossing function of mode 'i' crosses zero with negative to positive direction, a transition to 'j^{th}' destination mode happens.

- **Event Output:** This is an event output port, which is activated whenever a mode transition happens. This event is useful when an event is needed to activate or initialize a part of the subsystem not included in the internal dynamics of the automaton block.

In order to parametrize the Automaton block, the user should double click on the block. Then the block dialog opens and the user can enter the block parameters, as shown in Fig. 17.5.
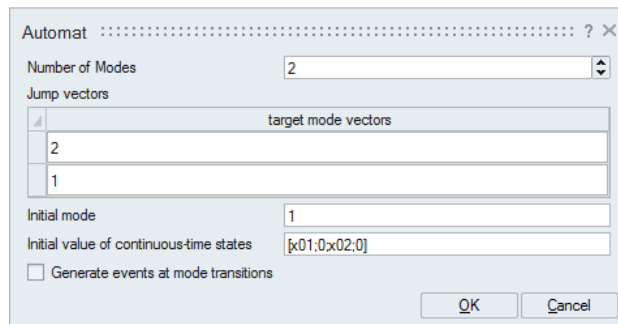
Figure 17.5: Block dialog of the `Automaton` block

In the block dialog, the number of control modes, the initial control mode and the initial value of continuous-time states at the beginning of the simulation should be given. *Xproperties* vector, coded in an $M \times N$ matrix, indicates whether a continuous-time state is algebraic or differential in each control mode. If a state is differential, its Xproperty should be set to "+1", otherwise it should be "-1". Xproperty can be given as a $1 \times N$ vector if Xproperties of the states in all modes are identical.

The *Jump* fields express the mode transition information. Suppose that all control modes are labeled from 1 to M. Then, in the field corresponding to control mode 'i', destination modes of mode 'i' are defined in a vector. j^{th} element of this vector gives the destination mode when j^{th} jump function ($Jump_{ij}(\dot{x}, x, u)$) becomes positive. For example, if in the field of the mode '2', the user defines $[1; 3; 4]$, it means that in mode '2', there are three active jump functions. When, for example, the third jump function becomes positive, a mode transition to mode '4' will be activated.

In order to illustrate the utility of this block for simplifying the development of hybrid automata models, the rest of this section is devoted to the implementation of two automata examples in **Twin Activate**.

17.3.1 Example 1: Pendulum Swing Up

The inverted pendulum is a popular system for illustrating control problems. In [6], the hybrid control strategy for swinging up a pendulum from a downward to an upright position has been presented. Here we will explain the way the pendulum and the control strategy can be modeled (and thus simulated) in **Twin Activate**, using a hybrid automaton block.

Let x_1 be the angle between the vertical axis and the pendulum, x_2 the angular velocity, and u the applied acceleration, which we consider as the control signal. Then, the dynamics of the system is given by

$$0 = f(\dot{x}, x, u) = \begin{cases} -\dot{x}_1 + x_2 \\ -\dot{x}_2 + ml(g\sin(x_1) - u\cos(x_1))/J \end{cases} \tag{17.3}$$

where g, m, l, J are the acceleration of gravity, the mass, the length, and the moment of inertia of the pendulum. It can easily be shown that the input control force u defined with (17.4)

$$u = \frac{\sin(x_1) + x_1 + x_2}{\cos(x_1)}g \tag{17.4}$$

brings the pendulum to the upward position with a single swing. Since the force of the pivot (u) is limited to $[-u_m, u_m]$, the control force defined with (17.4) cannot be used. As a result, in order to bring the pendulum up, more than one swing may be needed.

A swing-up control strategy suggested by Astrom and Furuta is based on a bang-bang control strategy controlling the energy of the pendulum [6]. Assuming zero potential energy in the upward position, the total energy of the pendulum E is given by:

$$E = J\frac{x_2^2}{2} + mgl(\cos(x_1) - 1) \tag{17.5}$$

Starting from a negative energy in down position, the control strategy should bring up the pendulum to zero energy. To change the energy as fast as possible, the magnitude of the control signal should be as large as possible. The pendulum energy will become zero, if

$$\frac{dE^2}{dt} = -mluEx_2\cos(x_1) < 0 \tag{17.6}$$

This condition is fulfilled with the control law (17.7), which drives $\|E\|$ to zero.

$$\begin{aligned} u &= u_m\text{sign}(\beta) \\ \beta(x_1, x_2) &= Ex_2\cos(x_1) \end{aligned} \tag{17.7}$$

The control law (17.7) results in chattering. Chattering can be avoided by using a *saturation* function instead of the *sign* function, *i.e.,* the control law (17.8).

$$u = u_m\text{sat}(k\beta) = \begin{cases} -u_m & k\beta < -1 \\ ku_m\beta & -1 < k\beta < 1 \\ u_m & 1 < k\beta \end{cases} \tag{17.8}$$

The parameter k determines the region where the strategy is linear. In practice, k is determined by the noise levels on the measured signals. When the pendulum arrives near the upward position, *i.e.,* $\cos(x_1) > c_1$, the nonlinear controller (17.4) is applied to take over the control of the pendulum. This pendulum system can be modeled with four control modes as shown in Fig. 17.6. Since following each jump the states are re-initialized to last value of states in the previous mode, the initialization equations are not shown.

The block dialog use to parametrize the Automaton block for this automaton is given in Fig. 17.5. The **Twin Activate** model of this system is given in Fig. 17.7. There are four subsystems handled by this automaton block. As an example, the superblock SB2 corresponding to the control mode-2 is given in Fig. 17.8.

The simulation result for a pendulum with $mgl/J = 1$, $u_m = g/4$, and $x(0) = [\pi, 0]^t$ is shown in Fig. 17.9. The evolution of control modes and the pendulum angle and angular velocity (x_1 and x_2) are given in the first and the second subplots, respectively.
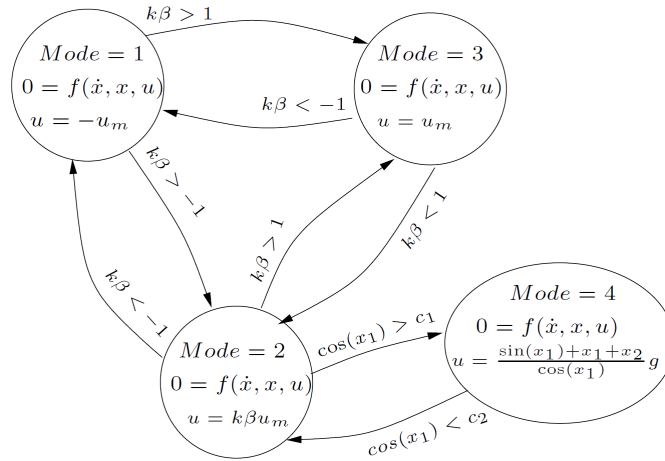
Figure 17.6: Hybrid control strategy to swing up and stabilize an inverted pendulum on a cart
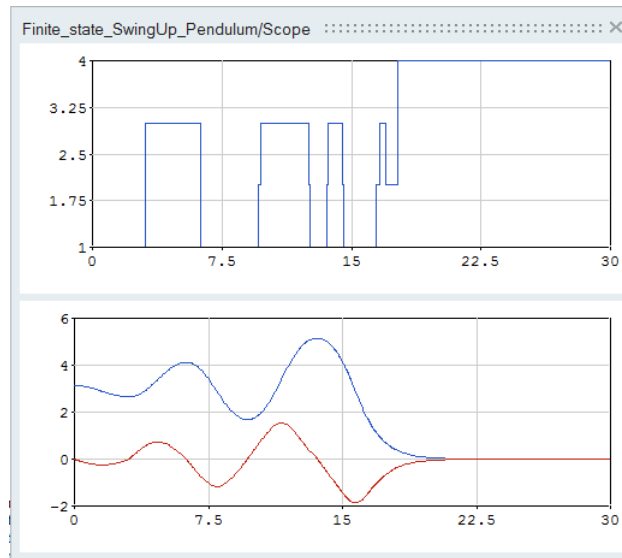
Figure 17.9: Simulation result for the model of Fig. 17.7

17.3.2 Example 2: DC/DC Buck Converter

Another hybrid system that shows an interesting dynamical behavior is the DC/DC buck converter whose schematic circuit diagram is shown in Fig. 17.10. A buck converter is a step-down DC to DC power supply composed of two electronic switches (a transistor and a diode), an inductor, and a capacitor [7].
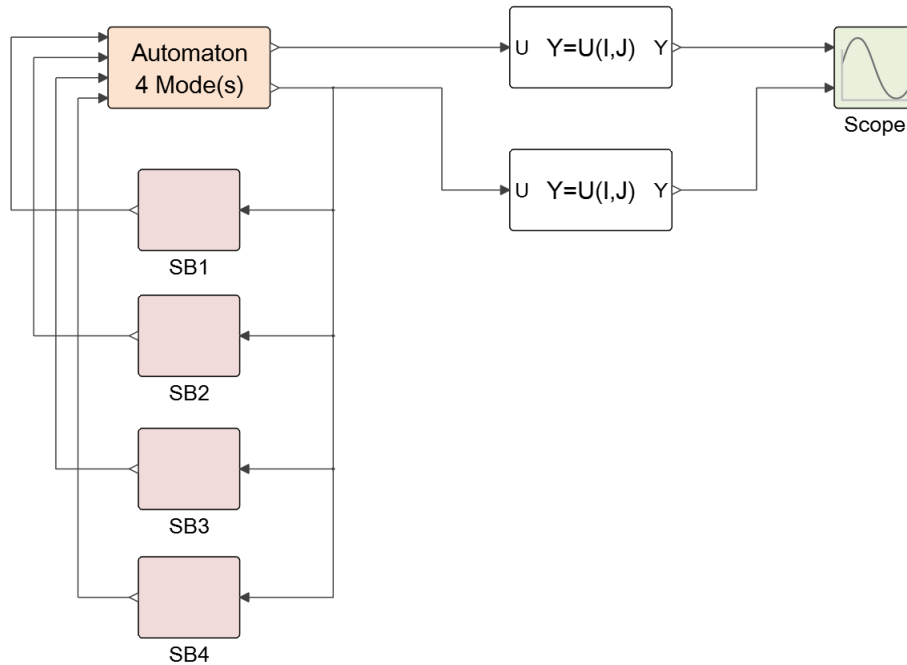
Figure 17.7: A model for swing up a pendulum (Finite_state_SwingUp_Pendulum.scm)
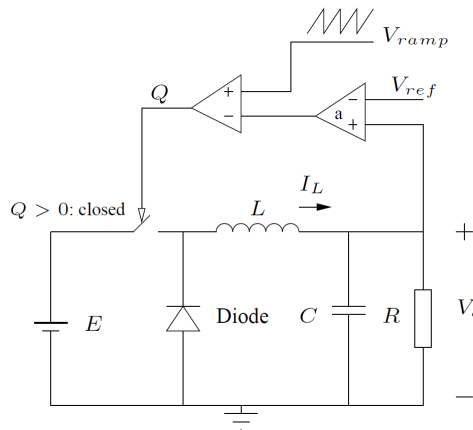
Figure 17.10: Voltage-mode controlled Buck DC/DC converter

The operation of the buck converter is fairly simple. It alternates between connecting the inductor to source voltage to store energy in the inductor and discharging the inductor into the load. Depending on the switches' status, the circuit has three configurations, thus three modes of operations. The first mode is active when the switching condition $Q > 0$ is satisfied (see (17.9d) or Fig. 17.10). In this case, the switch is closed and the diode is conducting. The second case is when the switch is open and the current in the inductor is positive. The positive current allows the diode to conduct. The last case is when in the second mode, the current through the inductor falls to zero. In this case, the inductor is completely discharged and the diode does not conduct. In order to implement this system with the automaton block, an automaton graph with three control modes is drawn; see Fig. 17.11. In each mode a different linear continuous-time dynamics defines the behavior of the circuit. Taking the state vector
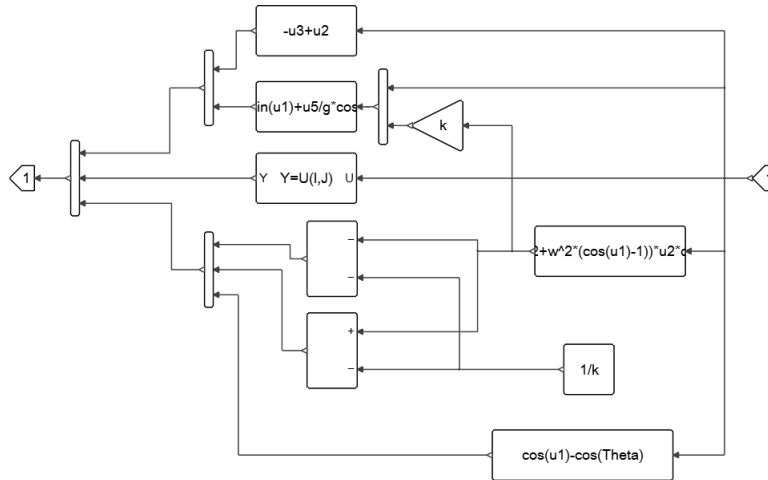
Figure 17.8: Superblock SB2 in the model of Fig. 17.7

as $x = [V_o, I_L]^t$, we get (17.9a-17.9c). Note that the Xproperties of the states evolve: x_2 is a differential variable in mode 1 and mode 2, and an algebraic variable in mode 3.
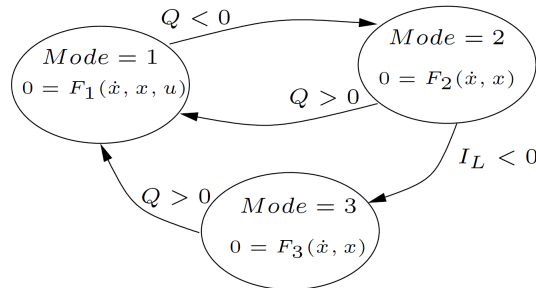
Figure 17.11: Hybrid automaton for a DC/DC Buck converter

$$0 = F_1(\dot{x}, x, u) = \begin{cases} RC\dot{x}_1 + x_1 - Rx_2 \\ L\dot{x}_2 + x_1 - E \end{cases} \tag{17.9a}$$

$$0 = F_2(\dot{x}, x) = \begin{cases} RC\dot{x}_1 + x_1 - Rx_2 \\ L\dot{x}_2 + x_1 \end{cases} \tag{17.9b}$$

$$0 = F_3(\dot{x}, x) = \begin{cases} RC\dot{x}_1 + x_1 \\ x_2 \end{cases} \tag{17.9c}$$

$$Q = V_{ramp} - (x_1 - V_{ref})a \tag{17.9d}$$

The hybrid automaton of Fig. 17.11 has been implemented in **Twin Activate** as given in Fig. 17.12.

Altair Twin Activate 2023
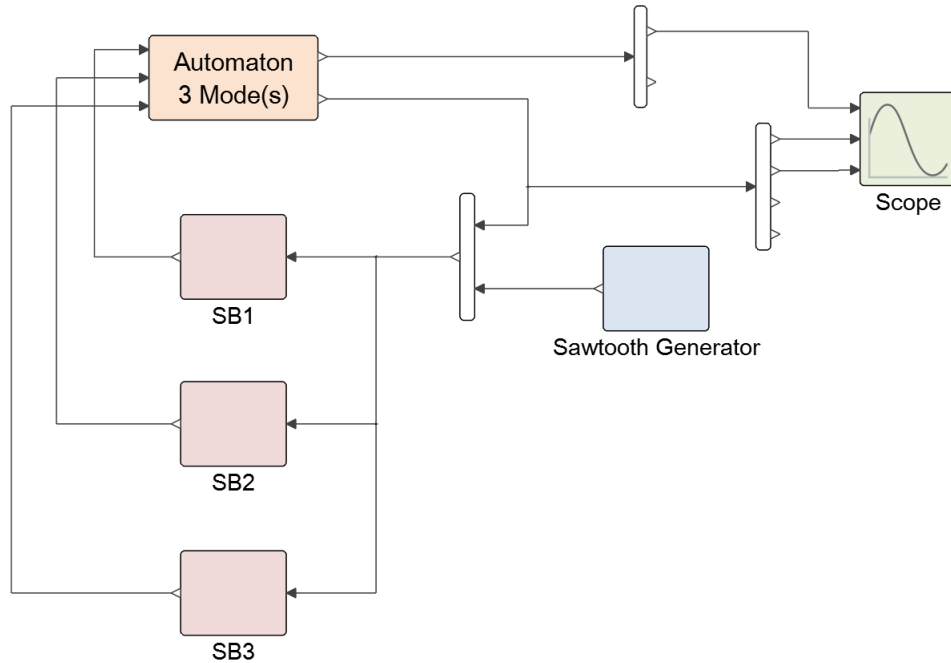
Figure 17.12: Model (Finite_state_Buck.scm) for automaton given in Fig. 17.11

For the circuit, we have used the parameter values proposed in [8].

$$
\begin{array}{lll}
L = 20mH, & C = 47\mu F, & R = 110\Omega, \\
a = 8.4, & V_U = 8.2v, & V_L = 3.8v, \\
V_{ref} = 11.3v, & E = 20v, & T = 400\mu S
\end{array}
\qquad (17.10)
$$

Depending on the value of R, the circuit operates in two or three operating modes. The simulation result is given in Fig. 17.13. The first subplot is the evolution of the control mode during the simulation, the second subplot shows the output voltage, and the third subplot is the current through the inductor. It is interesting to note that the circuit exhibits a chaotic behavior as explained in [8, 9].
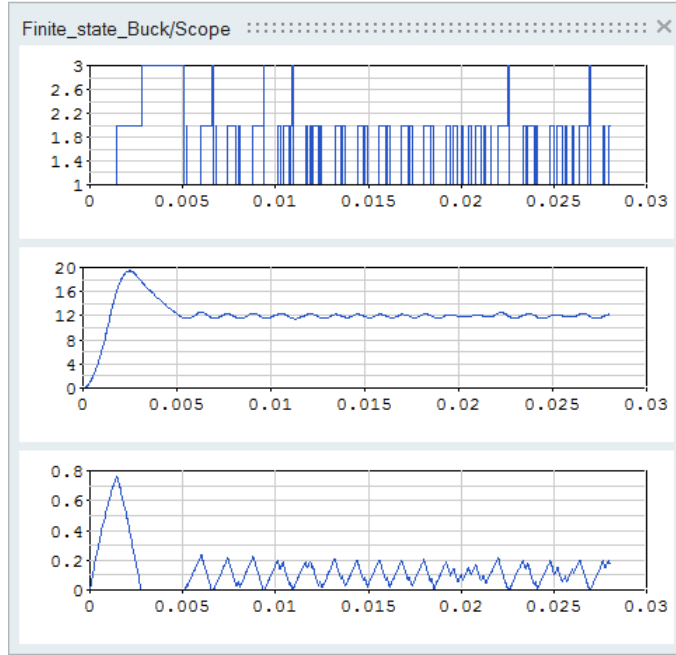
Figure 17.13: Simulation result for the model of Fig. 17.12

17.3.3 Example 3: Sticky balls

In this example ([10]), we consider two (sticky) masses on a flat frictionless table. As shown in Fig.17.14. Each of the balls is attached to a spring with distinct spring coefficient. If both balls are free, they will swing back and forth. However, if the two balls hit each other, they stick together. The stickiness is assumed to be exponentially decaying after they hit, and the decay speed is controlled by the parameter "Stickiness Decay" So after a while, they will separate.
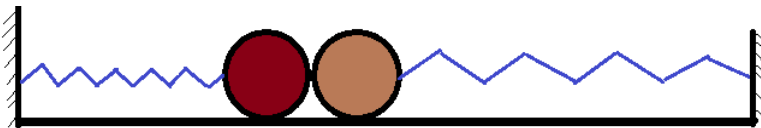
Figure 17.14: Schematic of two sticky balls

The dynamical equation describing the movement of two masses are given in 17.11 and 17.12.

$$\begin{cases} \dot{x}_1 & = v_1 \\ m_1 \dot{v}_1 & = -k_1(x_1 - a_1) \end{cases} \tag{17.11}$$

$$\begin{cases} \dot{x}_2 & = v_2 \\ m_2 \dot{v}_2 & = -k_2(x_2 - a_2) \end{cases} \tag{17.12}$$

where x_1, x_2, v_1, and v_2 are position and velocity of masses. Whenever the condition $(x_1 >= x_2)$ becomes true two balls stick together. The dynamical equation of the stuck masses is given in 17.13.

$$\begin{cases} \dot{x}_3 = v_3 \\ m_3 \dot{v}_3 = -k_3(x_3 - a_3) \end{cases} \tag{17.13}$$

where x_3 and v_3 are position and velocity of the stuck balls and

$$\begin{cases} m_3 = m_1 + m_2 \\ k_3 = k_1 + k_2 \\ a_3 = \frac{k_1*a_1+k_2*a_2}{a_1+a_2} \end{cases} \qquad (17.14)$$

The stickiness force is defined with the equation

$$\dot{s} = -Cs$$

where C is the decay constant. At the sticking instant (t_e), x_3, v_3, and s are reinitialized to

$$\begin{cases} x_3(t_e) &= x_1 \\ v_3(t_e) &= \frac{m_1 v_1+m_2 v_2}{m_1+m_2} \\ s(t_e) &= s_0 \end{cases} \qquad (17.15)$$

where s_0 is a constant. The stuck balls is subjected to two forces, the stickiness s and the separation force which equal to

$$F = k_1(x_1 - a_1) - k_2(x_2 - a_2)$$

Once $s < F$ becomes true, two balls separate. At the separation instant, states are reinitialized as follows.

$$\begin{cases} x_1(t_e) &= x_2(t_e) &= x_3(t_e) \\ v_1(t_e) &= v_2(t_e) &= v_3(t_e) \end{cases} \qquad (17.16)$$

The sticking balls can be modeled as a finite state system or an automation. There are two modes, separate and stuck as shown in Fig.17.15.
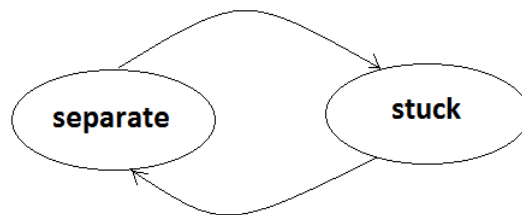
Figure 17.15: Hybrid automaton for sticky balls

In order to model this system in **Twin Activate** with an automaton block, the automaton block should be parametrized as shown in Fig.17.16.
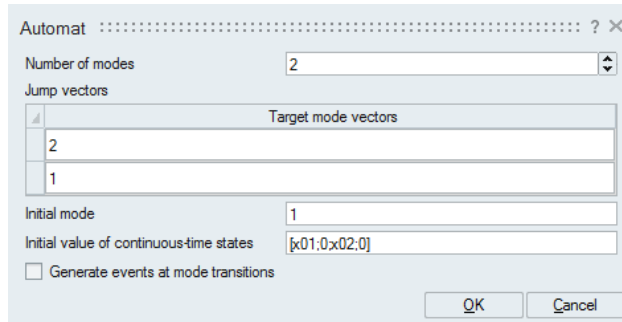
Figure 17.16: Parameterizing the Automaton block for the sticky masses model.

There are two modes and the continuous-time state size is four. Note that in this model number of states changes in each mode.

In the "separate" mode, the state vector X is composed of $[x_1, v_1, x_2, v_2]$. The residual vector delivered to the first port of the Automaton block is 17.17.

$$
0 = \begin{cases}
-\dot{x}_1 + v_1 \\
m_1 \dot{v}_1 + k_1(x_1 - a_1) \\
-\dot{x}_2 + v_2 \\
m_2 \dot{v}_2 + k_2(x_2 - a_2)
\end{cases}
\tag{17.17}
$$

When a transition from "stuck" mode to "separate" model happens, the state vector X is reinitialized to $[x_3, v_3, x_3, v_3]$. The zero crossing function in the separate mode is:

$$
g_1 = x_1 - x_2
\tag{17.18}
$$

In the "stuck" mode, the state vector X is composed of $[x_3, v_3, s, s]$. The residual vector delivered to the second port of the Automaton block is 17.19. The number of states of the block cannot change, as a result, in mode "stuck" we need a dummy state (q).

$$
0 = \begin{cases}
\dot{x}_3 = v_3 \\
m_3 \dot{v}_3 = -k_3(x_3 - a_3) \\
\dot{s} = -Cs \\
\dot{q} = -q
\end{cases}
\tag{17.19}
$$

When a transition from "separate" mode to "stuck" model happens, the state vector X is reinitialized to $[x_1, \frac{m_1 v_1 + m_2 v_2}{m_1 + m_2}, s_0, 1]$. The zero crossing function in the separate mode is :

$$
g_2 = k_1(x_1 - a_1) - k_2(x_2 - a_2) - s
\tag{17.20}
$$

The model is built in **Twin Activate** using the an automaton block, the block diagram of the system is shown in Fig.17.17. The simulation result illustrating the position of masses (x_1 and x_2) is given in Fig.17.18.
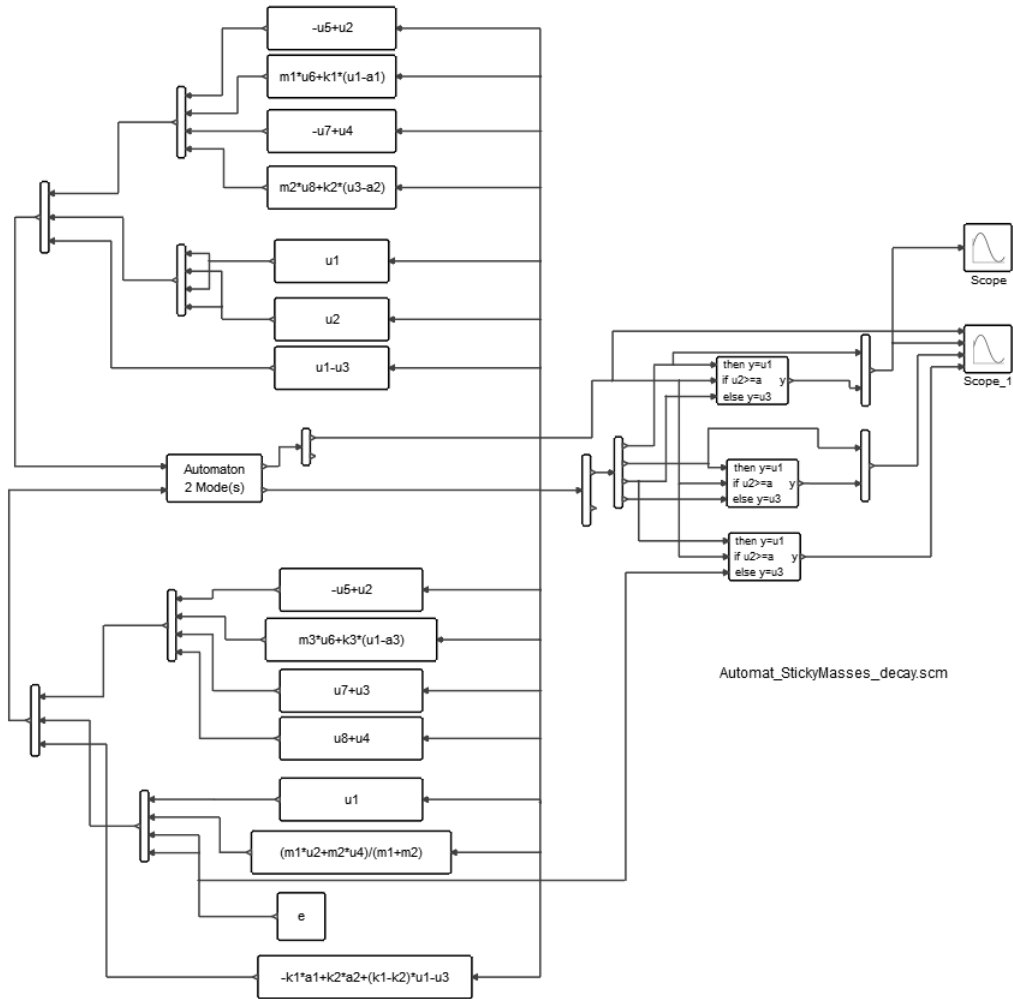
Figure 17.17: Complete model of sticky masses (Automat_StickyMasses_decay.scm)
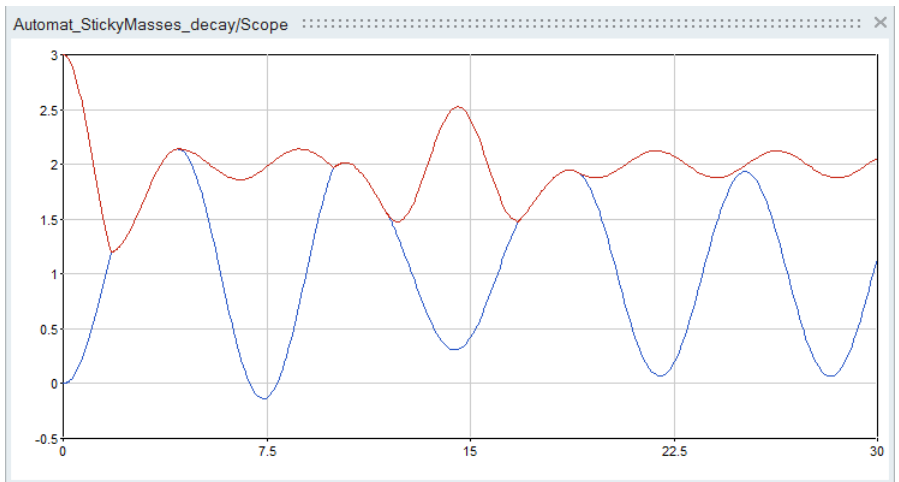
Figure 17.18: Position of masses.

17.4 conclusion

Hybrid automata are used to express hybrid dynamical systems. Although **Twin Activate** is a general purpose hybrid system simulator, developing a hybrid automaton with several subsystems is difficult and time-consuming. That is why the `Automaton` block is useful to allow straightforward construction of hybrid automata. This block centralizes the control of the hybrid automaton and can simulate complex multi-mode DAEs based on hybrid automata. This block can model different continuous-time dynamics in each automaton mode.

Bibliography

[1] L. P. Carloni, R. Passerone, A. Pinto, and A. L. Angiovanni-Vincentelli, "Languages and tools for hybrid systems design,"Theoretical Computer Science, vol. 1, pp. 1-193, 2006.

[2] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho, "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems," in Hybrid Systems, 1992, pp. 209-229. [Online]. Available: citeseer.ist.psu.edu/alur92hybrid.html

[3] N. Lynch, R. Segala, F. Vaandrager, and H. Weinberg, "Languages and tools for hybrid systems design in hybrid systems iii," Lecture Notes in Computer Science„ vol. 1066, pp. 496-510, 1996.

[4] T. A. Henzinger, "The theory of hybrid automata," Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS), pp. 278-292, 1996.

[5] X. Nicollin, J. Sifakis, and S. Yovine, "From ATP to timed graphs and hybrid systems," Acta Informatica, vol. 30, no. 2, pp. 181-202, 1993. [Online]. Available: citeseer.ist.psu.edu/article/nicollin93from.html

[6] K. J. Astrom and K. Furuta, "Swinging up a pendulum by energy control," Automatica, vol. 36, pp. 278-285, 2000.

[7] M. di Bernardo, F. Garofalo, L. Glielmo, and F. Vasca, "Switching, bifurcations, and chaos in DC/DC converters," IEEE Transactions on Circuits and Systems-I: Fundamental Theory and Applications, vol. 45, no. 2, pp. 133-141, Feb. 1998.

[8] E. Fossas and G. Olivar, "Study of chaos in the buck converter," IEEE Trans. on Circuits and Systems I, vol. 43, pp. 13-25, 1996.

[9] F. E. guezar, ascal Acco, H. Bouzahir, and D. Fournier-Prunaret, "Chaotic behavior in a hybrid dynamical system that arises from electronics," AIMS' Sixth International Conference on Dyn. Systems, Diff. Equations and Applications, Poitiers, France, 2006.

[10] http://sec.eecs.berkeley.edu/demo/StickyBall/StickyBall.htm

Chapter 18

Interpolation and extrapolation methods in Twin Activate

18.1 Introduction

Several blocks in **Twin Activate** such as `Lookup table`, `Signal generator`, `FromHDF`, `FromText`, `FromMat`, `FromCSV`, and `Signal-in` generate continuous-time signals from discrete-time signals. The discrete-time signal variable provides the values of a signal Y_i, at different time instants X_i. The value of the signal at times other than elements of X_i are computed via interpolation or extrapolation. Several interpolation and extrapolation methods are available inside **Twin Activate** blocks that will be explained in the following sections. In order to illustrate each method, the method is applied on this data-set whose x-y diagram is shown in Fig.18.1.

$$\begin{aligned} X &= [0, 1, 2, 4] \\ Y &= [4, 7, 6, 5] \end{aligned}$$
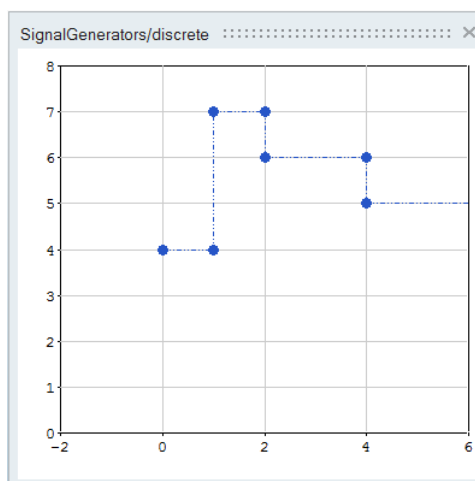
Figure 18.1: x-y diagram for test data set.

18.2 Interpolation methods

The values of the signal at time instants inside the range $[X_1, X_N]$ are computed via interpolation. Several interpolation methods are available in **Twin Activate** blocks:

- **ZeroOrder**, **ZeroOrder(floor, Point_just_below)**: The index of element in X which is equal or below x is chosen to compute the output, i.e.,

$$\texttt{for } X_i \leq x < X_{i+1}$$

$$y = Y_i$$

The interpolation result is shown in Fig.18.2.
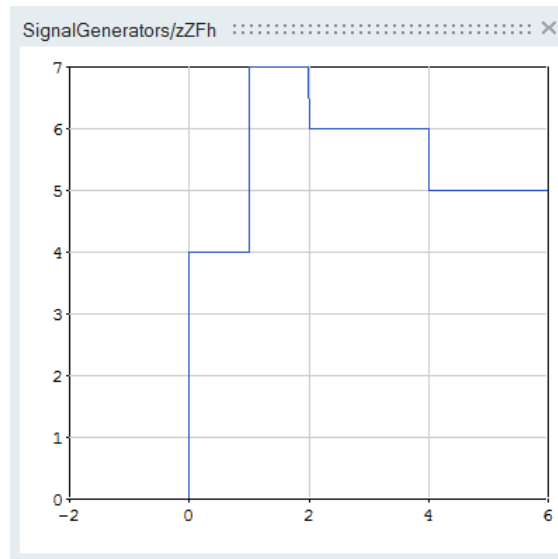
Figure 18.2: x-y diagram **ZeroOrder**, **ZeroOrder(floor)** methods.

- **ZeroOrder(ceil, Point_just_above)**: The index of element in X which is equal or above x is chosen to compute the output, i.e.,

$$\texttt{for } X_i < x \leq X_{i+1}$$

$$y = Y_{i+1}$$

The interpolation result is shown in Fig.18.3.

Altair Twin Activate 2023   Extended Definitions 352
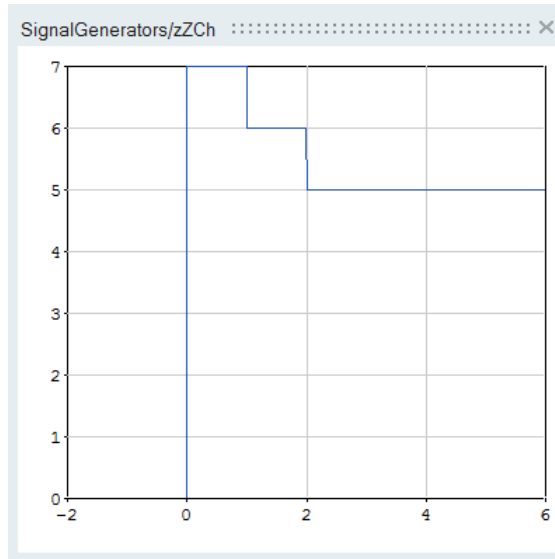
Figure 18.3: x-y diagram **ZeroOrder(ceil)** methods.

- **ZeroOrder(nearest, nearest_poin)**: The index of element in X which is nearest to x is chosen to compute the output, i.e.,

$$\texttt{for} \quad \frac{X_{i-1} + X_i}{2} \leq x < \frac{X_i + X_{i+1}}{2}$$

$$y = Y_i$$

The interpolation result is shown in Fig.18.4.
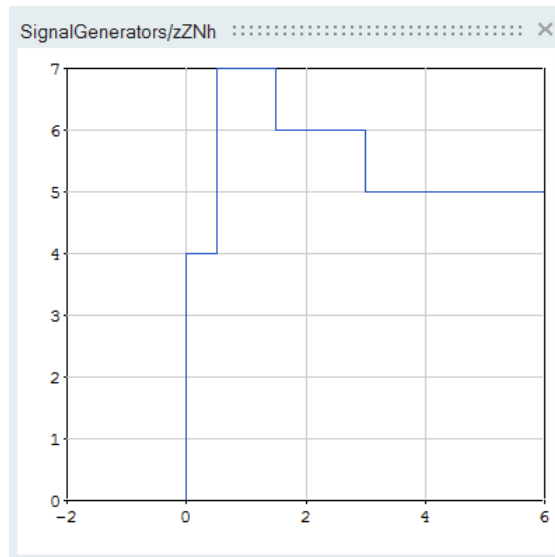
Figure 18.4: x-y diagram **ZeroOrder(nearest)** methods.

- **Linear**. This is the default method and performs a linear interpolation. If x matches one of X elements, the output is the corresponding element in the Y matrix. If no value matches, then

a linear interpolation is performed between two nearest elements of X to determine the output value, i.e.,

$$\text{for } X_i \le x < X_{i+1}$$

$$y = Y_i + (x - X_i) \frac{Y_{i+1} - Y_i}{X_{i+1} - X_i}$$

The interpolation result is shown in Fig.18.5.
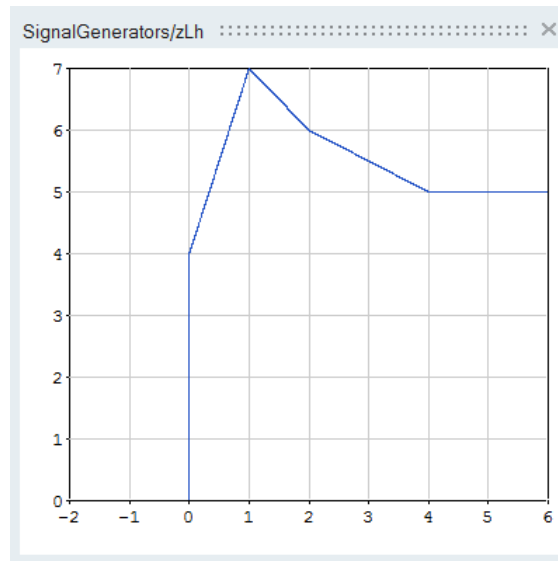
Figure 18.5: x-y diagram **Linear** methods.

- **Bilinear interpolation**: The bilinear interpolation is an extension of linear interpolation for interpolating functions of two variables on a rectilinear 2D grid. In this method, first an interpolation in one direction is performed, and then again in the other direction. Although each step is linear in the sampled values and in the position, the interpolation as a whole is not linear but rather quadratic in the sample location.

- **Trilinear interpolation**: The trilinear interpolation method is an interpolation on a 3-dimensional regular grid which approximates the value of an intermediate point within the local axial rectangular prism linearly, using data on the lattice points. The trilinear interpolation is an extension of linear interpolation for interpolating functions of three variables on a 3D grid. In this method, first an interpolation in one direction is performed, and then again in the other directions. Although each step is linear in the sampled values and in the position, the interpolation as a whole is not linear but rather cubic in the sample location.

- **Not_a_knot**: A cubic spline is computed by imposing following conditions (considering N points $X_1,...,X_N$).

$$\begin{cases} S^{(3)}(X_2^-) & = S^{(3)}(X_2^+) \\ S^{(3)}(X_{N-1}^-) & = S^{(3)}(X_{N-1}^+). \end{cases}$$

where $S^{(n)}$ is n^{th} derivative of the interpolated curve.

The interpolation result is shown in Fig.18.6.
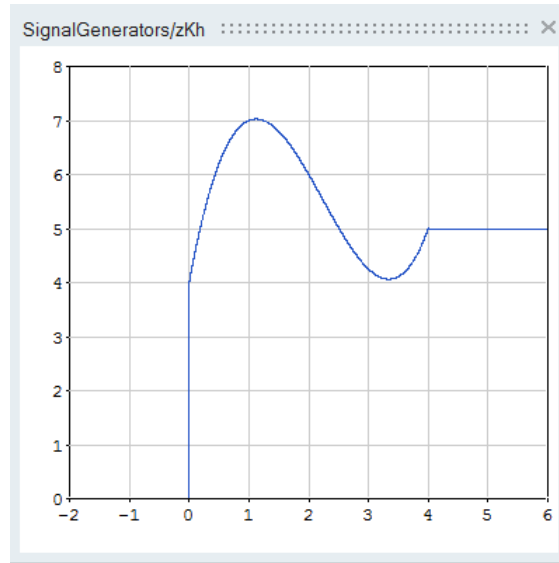
Figure 18.6: x-y diagram **Not_a_knot** methods.

• **Natural**: A cubic spline is computed by imposing following conditions:

$$
\begin{cases}
S^{(2)}(X_1) & = 0 \\
S^{(2)}(X_N) & = 0.
\end{cases}
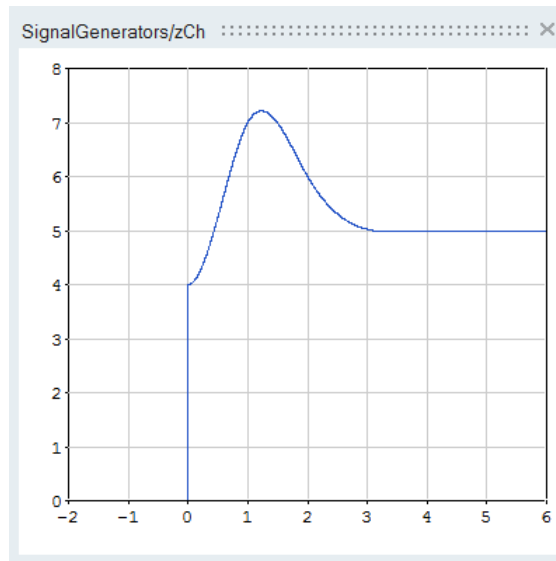$$

The interpolation result is shown in Fig.18.7.
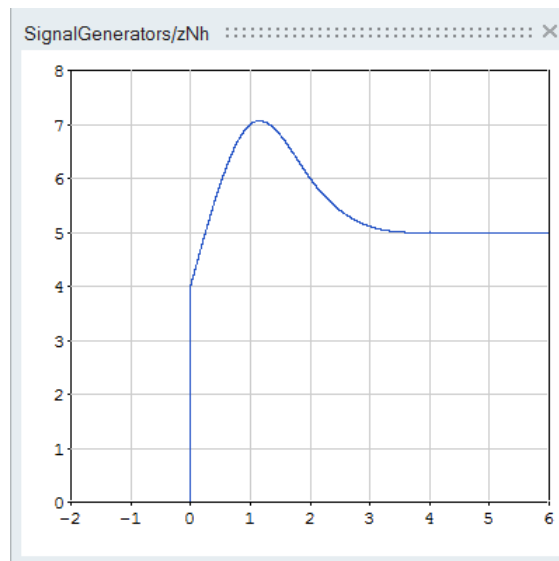
Figure 18.7: x-y diagram **Natural** methods.

• **Clamped_To_Zero**: A cubic spline is computed by imposing following conditions:

$$\begin{cases} S^{(1)}(X_1) & = 0, \\ S^{(1)}(X_N) & = 0. \end{cases}$$

The interpolation result is shown in Fig.18.8.

Figure 18.8: x-y diagram **Clamped_To_Zero** methods.

- **Akima interpolation method**: The Akima interpolation method [1] is a continuously differentiable sub-spline interpolation method built from piecewise third order polynomials. This method provides less ringing and overshoot than natural and not-a-knot spline algorithms. In this method only data from the next neighbor points is used to determine the coefficients of the interpolation polynomial. Since no functional form for the whole curve is assumed and only a small number of points is taken into account this method does not lead to unnatural values in the resulting curve.

- **Fritsch-Butland interpolation method**:

  The Fritsch-Butland interpolation method [2] is a continuously differentiable cubic interpolation method built from piecewise third order polynomials. In this method local first derivative approximations is used and guaranties to generate locally monotone interpolation curves.

- **Steffen interpolation method**:

  The Steffen interpolation method [3] guarantees the monotonicity of the interpolating function between the given data points. The minima and maxima can only occur exactly at the data points, and there can never be spurious oscillations between data points. The resulting curve is piecewise cubic on each interval with the slope at each grid point chosen to ensure monotonicity and prevent undesired oscillations. The interpolated function is piecewise cubic in each interval. The first-order derivative is guaranteed to be continuous everywhere, but the second derivative may

---

[1]Hiroshi Akima, 'A New Method of Interpolation and Smooth Curve Fitting Based on Local Procedures,' Journal of the ACM, Vol. 17, No. 4, pp. 589-602, 1970

[2]F. N. Fritsch and J. Butland. 'A method for constructing local monotone piecewise cubic interpolants'. SIAM J. Sci. Statist. Comput., 5 (1984), pp. 300-304

[3]M.Steffen, "A simple method for monotonic interpolation in one dimension", Astron. Astrophys. 239, 443-450 (1990).

be discontinuous.

18.3   Extrapolation methods

The values of the signal at time instants outside of the range $[X_1, X_N]$ are computed by extrapolation. Several extrapolation methods are available in **Twin Activate** blocks. Extrapolation can be applied either at the left-side, i.e., for $x < X_1$ or at the right-side, i.e., for $x > X_N$.

- **Zero**:This is the default method. In this case, the output is zero for time instants out of signal's range, i.e.,

$$\begin{cases} \texttt{for} \ \ x < X_1, & y = 0.0 \\ \texttt{for} \ \ x > X_N, & y = 0.0 \end{cases}$$

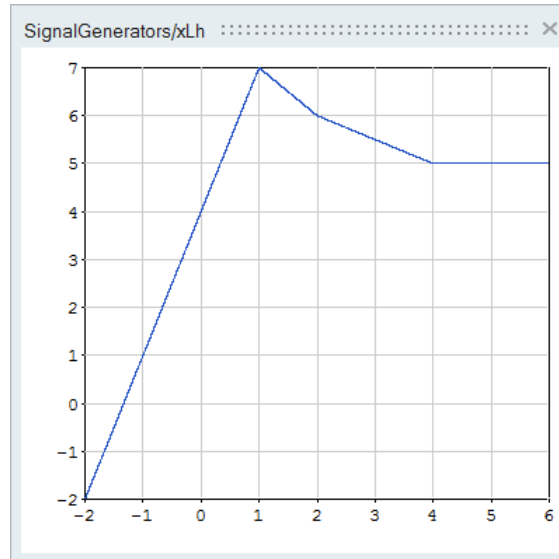The left-side **Zero** extrapolation result is shown in Fig.18.9.
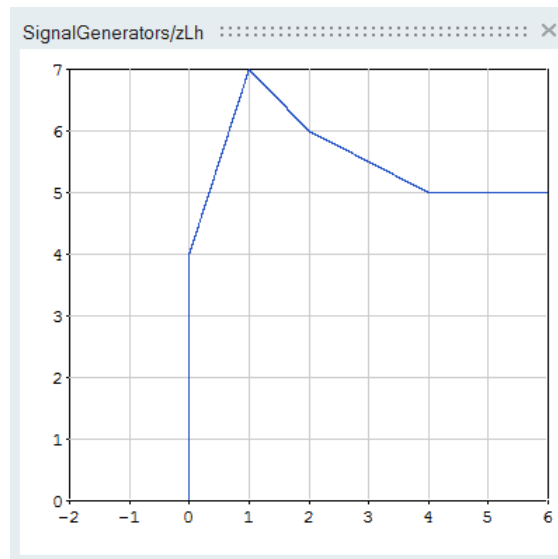
Figure 18.9: x-y diagram for left-side **Zero** extrapolation methods.

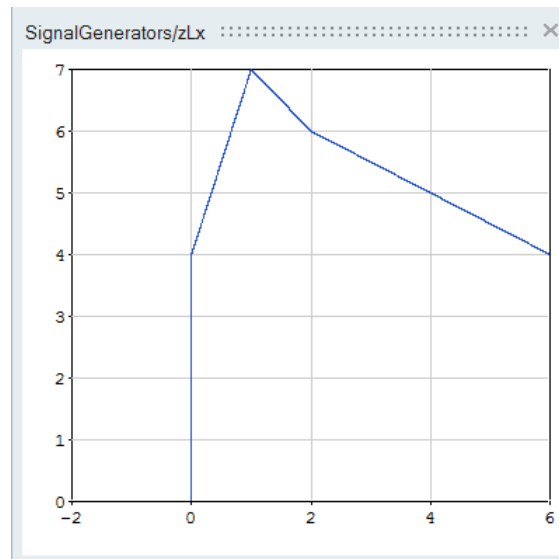The left-side and right-side **Zero** extrapolation result is shown in Fig.18.10.
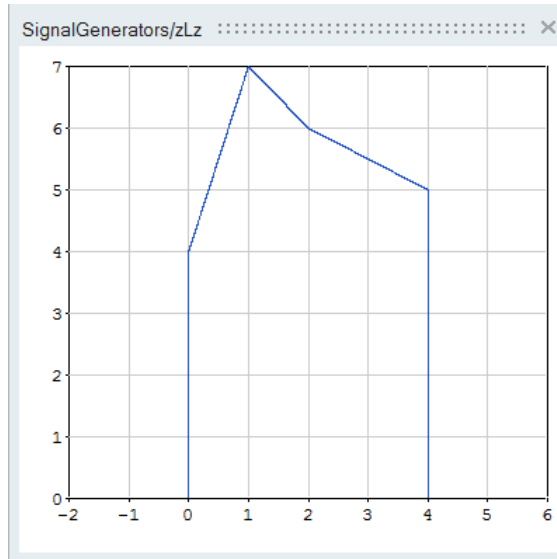
Figure 18.10: x-y diagram for left-side and right-side **Zero** extrapolation methods.

- **Hold**: The output is equal to the last element of the Y vector, i.e.,

$$\begin{cases} \texttt{for } x < X_1, & y = Y_1 \\ \texttt{for } x > X_N, & y = Y_N \end{cases}$$

The left-side and right-side **Hold** extrapolation result is shown in Fig.18.11.
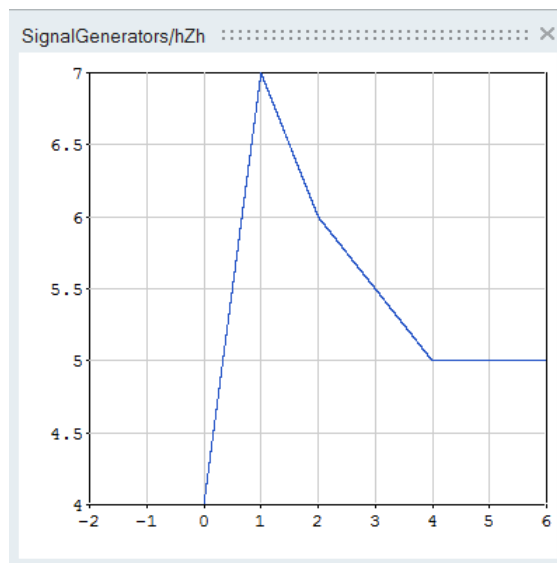
Figure 18.11: x-y diagram for left-side and right-side **Hold** extrapolation methods.

- **Extrapolation**. A linear extrapolation is performed using two left-most or right-most elements of

the signal, i.e.,

$$\begin{cases} \text{for } x < X_1, \quad y = \ Y_1 + (x - X_1)\frac{Y_2 - Y_1}{X_2 - X_1} \\ \text{for } x > X_N, \quad y = \ Y_{N-1} + (x - X_{N-1})\frac{Y_N - Y_{N-1}}{X_N - X_{N-1}}, \end{cases}$$

The left-side **Extrapolation** extrapolation result is shown in Fig.18.12.

Figure 18.12: x-y diagram for left-side **Extrapolation** extrapolation methods.

The right-side **Extrapolation** extrapolation result is shown in Fig.18.13.

Figure 18.13: x-y diagram for right-side **Extrapolation** extrapolation methods.

- **Repeat**. The signal is repeated over the whole range of X. For $x < X_1$ or $x > X_N$

$$y = y(x - k(X_N - X_1))$$

where k is an integer number to satisfy the following condition.

$$X_1 \leq x - k(X_N - X_1) < X_N$$

The left-side **Repeat** extrapolation result is shown in Fig.18.14.
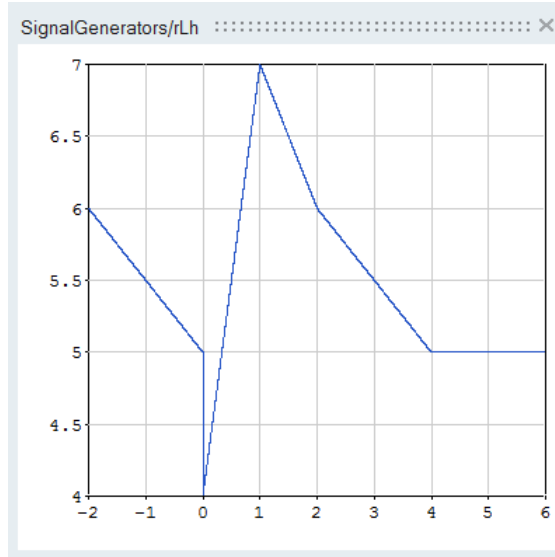
Figure 18.14: x-y diagram for left-side **Repeat** extrapolation methods.

The right-side **Repeat** extrapolation result is shown in Fig.18.15.
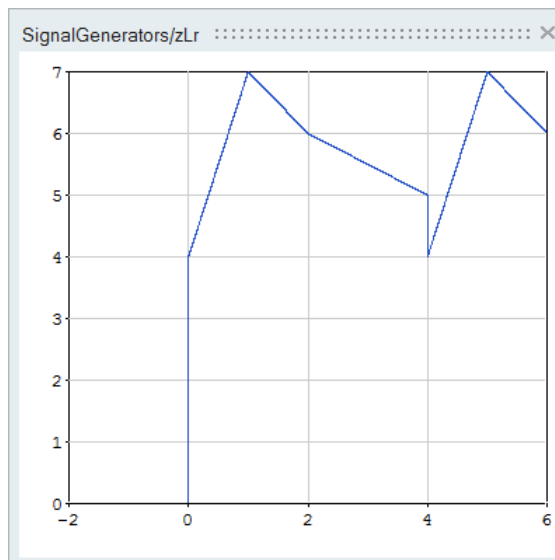
Figure 18.15: x-y diagram for right-side **Repeat** extrapolation methods.

18.3.1 Application - Behavior in blocks

The interpolation and extrapolation methods described above are available inside **Twin Activate** blocks and are used to compute the value of the signal for inputs that are either before or after the data span.

In the following tables, all options and behaviors are described. The options can be set in the Advanced tabs of the blocks

Case **with** External Activation - Zero Order		
Selected Action	Left time span	Right time span
Zero	Zero	Zero
Hold	Hold	Hold
Extrapolation	Hold	Hold
Repeat	Repeat	Repeat

Case **without** External Activation - Zero Order		
Selected Action	Left time span	Right time span
Zero	Zero	Hold
Hold	Hold	Hold
Extrapolation	Extrapolation	Extrapolation
Repeat	Repeat	Repeat

Case **without** External Activation - Linear		
Selected Action	Left time span	Right time span
Zero	Zero	Zero
Hold	Hold	Hold
Extrapolation	Extrapolation	Extrapolation
Repeat	Repeat	Repeat

Chapter 19

SignalOut and SignalIn blocks in Twin Activate

19.1 Introduction

A signal is an object used to define time dependent function over a given time span. Signals are often defined and used in **Twin Activate** but they are also accessible in the environment (**OML** workspace). Signals store time functions, mainly **Twin Activate** simulation results, and also can provide input to **Twin Activate** models. Signals also can be used for comparison and validation of simulation results, in particular in the QA process. Signals can also be used as reference inputs in **Twin Activate** models.

Signals can either be created and queried in **OML** using table commands, or be created and accessed in the **Twin Activate** environment using SignalIn and SignalOut blocks, respectively. A Signal is a table in the **OML** environment composed of the time (`time`) and data (`ch`) fields. Following commands are used to create a signal with name `mysg`.

- **mysg={}** : This command is used to create the empty signal `mysg` in **OML**.
- **mysg.time** : The `time` field is a 3*N matrix where N is the number of time samples stored in the signals structure.
    - **time vector** : The first row of `mysg.time` matrix is the vector of the sampling time instants.
    - **event coding** : The second row of `mysg.time` matrix is the event-coding (synchronization information) of the corresponding sample. This coding will be explained in 19.2.
    - **event kind** : The third row of `mysg.time` matrix indicates the kind of the event that has triggered the sampling. The event kind will be explained in 19.2
- **mysg.ch={}** : `mysg.ch` is a sub-table used for storing the set of data. Once `mysg.ch` created, data can be stored in the several channels. Each channel is a table composed of `type` and `data` fields.
- **mysg.chi.type** : Indicates the data-type of the ith channel, i.e., `double`, `complex`, `int32`, `int32`, `int16`, `int8`, `uint32`, `uint16`, `uint8`.
- **mysg.chi.data**: Represents the data stored in the ith channel and can be a matrix of size [Nrow, Ncol*N]. In fact, the data matrices of samples of size [Nrow*Ncol] are concatenated.

Example: Here is an example of a signals structure with two data sets.

```
B={};
B.time=[1 2 4];
B.ch={};
B.ch{1}={};
B.ch{1}.type='double';
B.ch{1}.data=[1 2 3];
B.ch{2}={};
B.ch{2}.type='int32';
B.ch{2}.data=[1 2 3 4 5 6;7 8 9 10 11 12];
```

Now you can use the "B" signal in a SignalIn block as we will see in following sections.

19.2   Activation information inside a signal object

Signal objects in **Twin Activate**, contain specific information about how the SignalOut block has been activated. An activation signal specifies the time instants where the signal may potentially change value. For example a discrete-time signal, which in **Twin Activate** is simply a piece-wise constant function, has an activation signal present at the points of potential discontinuities.

In general, an activation signal is a series of isolated points, called events, and intervals over time. If the activation signal contains only isolated points, then we say that the corresponding signal is discrete-time. A special case of a discrete-time signal is the periodic signal where the activation points are equally spaced. But the latter property does not imply that a discrete-time signal is periodic, a periodic signal must be declared as such. Activation intervals correspond to periods over which the signal evolves as a continuous-time function. The always-active signal is a special case of activation signal in which there exists a single activation interval covering the entire time period. A **Twin Activate** signal in general is made of multiple activation intervals. Over each such period, the corresponding time function is supposed to be at least continuous and differentiable. Any point of discontinuity, in the value or the derivative of the function, corresponds to a specific event. Two events may have the same time, therefore a **Twin Activate** signal may take on successive values at the same time instant. The time event that is not being used as an indexing mechanism can mean that based on the time values of the two events associated with two different signals, nothing can be concluded as to their synchronization. Different methods can be used for coding signals over continuous-time activation periods. The method used here samples the function and stores a list of time-value pairs, just as with event activations. An indicator however distinguishes between the two types of "events".

Signals are typically created in **Twin Activate** by the SignalOut blocks and read by the SignalIn blocks. **Twin Activate** signals are always considered asynchronous because each has its own `activation signal`. In order to preserve synchronization information between several data series, multiple data can be placed together into one signal. In other words, a signal may contain several data series and a single time vector. A Signal contains a time vector. For each time, the following information is associated:

Event-coding or synchronization information (accessed by `GetNevIn(block)` API) indicates the port[s] that has[ve] been activated at the sampling time instants. remember binary coding of numbers, for example, if an event activates the SignalOut via the input port 1, its coding 1. If an event activates the SignalOut via the input port 2, its coding is 2. if an event activates the SignalOut via the input port 2 and 1, its coding is 3.

Event-kind or simulation phase (accessed by `GetSimulationPhase(block)` API) provides infor-

mation about the kind of activation. In the following table, different kinds of activation are listed.

Phase	Definition
PHASE_MESHPOINT_LEFTL (numerical value=-1)	A Left-Limit of Discrete event: sampling that is triggered just before a discrete event happens. The sampling of this kind of point is not active by default in the SignalOut block.
PHASE_MESHPOINT (numerical value=0)	Mesh-point (continuous): A pure continuous sampling point, also known as a solver's mesh-point. The sampling of this kind of point is active by default in the SignalOut block.
PHASE_DISCRETE (numerical value=1)	discrete event: A discrete event has triggered the sampling, or this is the very first non-try call of the block. The sampling of this kind of point is active by default in the SignalOut block.
PHASE_TRY_MFX (numerical value=2), PHASE_TRY_MRLX (numerical value=3)	Try calls: Indicates the try points that a simulator or a solver may take. The sampling of this kind of point is not active by default in the SignalOut block.
PHASE_ZC_LEFTL (numerical value=4)	Left-limit of Zero-crossing: A sample just before the zero-crossing event takes place. This is useful to register the states or outputs just before a zero-crossing event that may cause discontinuities. The sampling of this kind of point is not active by default in the SignalOut block.
PHASE_ZCROSS (numerical value=5)	Zero-crossing: A zero-crossing event triggered the sampling. The sampling of this kind of point is active by default in the SignalOut block.
PHASE_EXIT_INITIALIZATION (numerical value=6)	First-call: The block is being called for the first time after the initialization has finished. After this call the simulation is started.

19.3 SignalOut block

The SignalOut block[1] allows for creating a Signal in the **Twin Activate** environment. The created signals then can be retrieved in the **OML** workspace using ordinary table operations. The Parameters Dialog of the SignalOut block is shown in Fig.19.1 and 19.2.

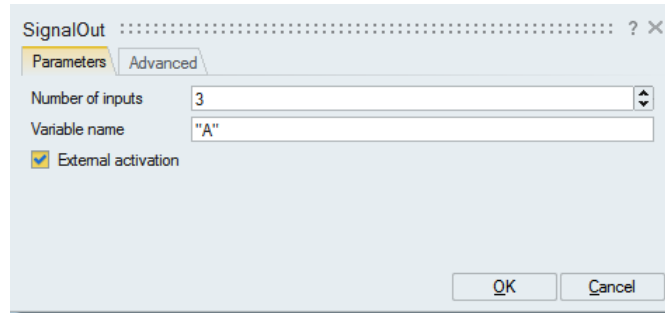[1]Available in `SignalExporters` palette

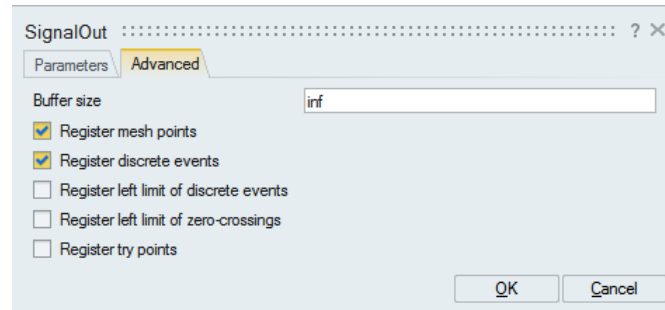Figure 19.1: Parameters Dialog of the SignalOut block.

Figure 19.2: Parameters Dialog of the SignalOut block (advanced tab).

The block has several parameters:

- **Number of inputs**: Used for specifying the number of data sets to be stored. This represents the number of elements in the `ch` field that will be available in the generated Signals. The data-type of each channel can be different and is inherited from the source block.

- **Variable name**: Indicates the name of the signal generated in the **OML** environment.

- **External activation**: Once checked, the block samples its inputs only when the block is activated by the incoming events.

- **Buffer-size**: The buffer indicates the number of samples to be stored. If number of samples exceeds the size of the buffer, the earliest sample is replaced by the new one. If the user needs to keep all samples, the buffer-size should set to `inf` which means unlimited.

- **Sampling filter**: In many occasions, the user needs to register only specified kind of samples, e.g., only continuous-time samples or only discrete ones. These check boxes allow the user filtering out unnecessary samplings points. Note that the event kind `try` would not generate samples with increasing time samples. Note that in order to regenerate the signal by the SignalIn block, the time vector of the signal needs to be increasing.

19.4 SignalIn block

The SignalIn block[2] reproduces the data series stored in the importing signal in the **Twin Activate** environment. The GUI of the SignalOut block is shown in Fig.19.3 and 19.4.

---

[2]Available in `SignalImporters` palette
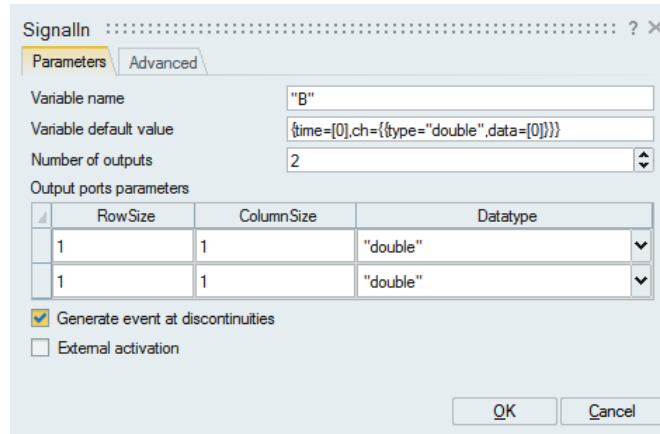
Figure 19.3: GUI of the signalIn block.
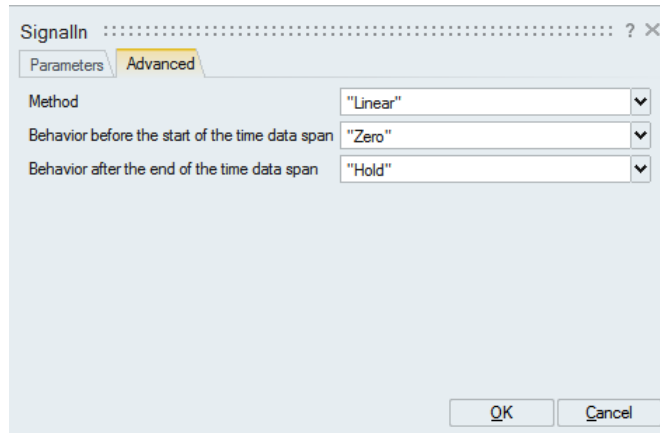
Figure 19.4: GUI of the signalIn block (advanced tab).

The block has several parameters:

- **Variable name**: The name of the signal to be retrieved from **OML** environment.

- **Variable default value**: If the requested signal name is not present in the **OML** environment, this filed allows providing a default value for the signal.

- **Number of outputs**: The SignalIn block needs the number of `ch` channels in the signal variable. Furthermore, for each channel, the data-size and data-type should be known, before reading the signal variable. Otherwise an error signal will be emitted indicating the difference between the read signal and the given parameter in the GUI.

- **Activated by the external events (ignore time)**: Once checked, all time information in the signal variable is ignored and external events trigger the output of the block. At each event trigger, a new sample is read.

- **Generate event at discontinuities**: This option allows generation events at sample times stored in the signal variable.

- **Method**: Indicates the interpolation method used for regeneration of the signal data series in the **Twin Activate** environment. The acceptable values are:

- **–** Zero-order (order 0)

- **–** Linear (order 1)

- **–** Not-a-knot (order 3)

- **–** Natural (order 3)

- **–** Clamped to zero (order 3)

These methods have been explained in the Interpolation methods chapter(18). When one of the data types stored in the signal variable is not of type double, the interpolation method should be zero-order.

- **Behavior before the start of the time data span**: If the output value of a channel in a signal value is requested for a time instant before the start of the time span of the signal, this extrapolation method is used. The accepted values are:

  - **–** Zero: Output is zero.

  - **–** Extrapolation (Linear): The output is computed based on a linear extrapolation between first and second data samples (if they exist).

  - **–** Hold: The output will be the value of the first data sample.

  - **–** Repeat: The output is computed to produce a periodic signal, repeating the whole time span of the signal variable.

These methods has been explained in the Interpolation methods chapter(18).

- **Behavior after the start of the time data span**: If the output value of a channel in a signal value is requested for a time instant after the start of the time span of the signal, this extrapolation method is used. The accepted values are

  - **–** Zero: Output is zero.

  - **–** Extrapolation (Linear): The output is computed based on a linear extrapolation between two end data samples (if they exist).

  - **–** Hold: The output will be the value of the final data sample.

  - **–** Repeat: The output is computed to produce a periodic signal, repeating the whole time span of the signal variable.

These methods have been explained in the Interpolation Methods chapter (18).

Chapter 20

Animation

Currently **Twin Activate** does not provide means to create sophisticated animations. The only block creating animations is the **Anim2D** block, which can be used to animate simple 2D graphical objects.

The block is based on **OML** functions to create figures and graphical objects, and define and modify their parameters. These functions can also be used directly by the user in an **OML** based block (for example the **OmlCustomBlock**) to create custom animations. In most cases however the **Anim2D** block is easier to use. Some behaviors, for example graphical objects changing color during simulation, are not supported by the block and custom animation must be used. The **OML** functions also give access to more properties such as the edge colors and thickness of some of the objects. The list of these **OML** functions can be found in **OML** documentations.

20.1   Anim2D block

The **Anim2D** block provided in the `SignalViewers` palette can be used to create 2D animations. Such animations are useful in particular to illustrate the behavior of simple mechanical and hydraulics systems.

The **Anim2D** block can be used to animate different types of graphical objects in a Figure (on a canvas). The block has two input activation ports and a variable number of regular input ports.

Block regular inputs can be associated with different object types:

1. Type 1 is a closed polyline and the input signal is an $N \times 2$ matrix where each row represents the x, y coordinates of a point of the polyline.

2. Type 2 is similar to type 1 but the polyline is not closed.

3. Type 3 corresponds to circles and the input is an $M \times 3$ matrix. Each row corresponds to a circle and contains the x, y coordinates of the center of the circle and its radius.

4. Type 4 corresponds to rectangles and the input is an $M \times 4$ matrix. Each row corresponds to a rectangle and contains the x, y coordinates of the lower left corner of the rectangle, and its width and height.

5. Type 5 corresponds to line segments and the input is an $M \times 4$ matrix. Each row corresponds to a line segment and contains the x, y coordinates of the two end points of the segment $(x1, y1, x2, y2)$..

Types 103, 104 and 105 are respectively the variants of types 3, 4, and 5, where the color of the object is defined by the input signal and not a block parameter, and can even change during simulation. For these types, the inputs have three additional columns defining their colors (standard RGB coding).

The block parameters are:

- Number of input ports (nin), i.e., the number of input signals.
- Object types: vector of size nin. If all types are identical, a scalar may be used.
- Canvas geometry: vector of size 4.
- Color table: a matrix with three columns representing the RGB values of each color (valued between 0 and 1). A row including -1's is considered transparent. The number of rows need not match the number of objects being animated. If the table is too small, colors are reused from the top of the table.
- Zoom factor setting: correspondence between Figure coordinate values and the number of points on screen.

When the block is activated through its first input activation port, the animation advances by moving all the objects to the positions specified by the current input signals. When it is activated through its second activation port, the objects are duplicated and a copy is frozen in place on the Figure in order to create snapshots visible at the end of the simulation (useful for example for exporting the Figure as an image).

Object types can be given negative values (-1 instead of 1, -2 instead of 2, etc.) if they should be excluded from snapshots. This is particularly useful for non-moving objects (such as background decorations) that need not be included in the snapshots multiple times.

20.2 Examples

20.2.1 Bouncing balls

The **Twin Activate** bouncing balls demo contains the animation of an arbitrary number of balls bouncing (in 2D) against each other and against the walls, floor and ceiling. The information available for animation is the positions and the radii of the balls, and the positions of the walls, floor and ceiling. The coordinates x and y of the center of balls are given by signals, the rest are constant parameters.

The top diagram of the bouncing balls demo is shown in Fig. 20.1. The content of the Animation Super Block is illustrated in Fig. 20.2. The **Anim2D** block is used with two inputs. The first is of type rectangle used to display the walls, floor and ceiling. The second input is of type circle and animate the balls. See Fig. 20.3. The **Anim2D** block parameters are shown in Fig. 20.4.
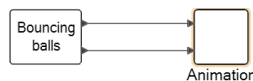

Figure 20.1: The Bouncing balls block output vectors provide the x and y coordinates of the balls.
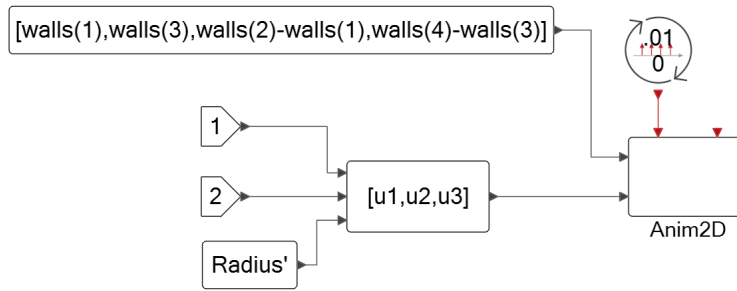
Figure 20.2: The content of the Animation Super Block.
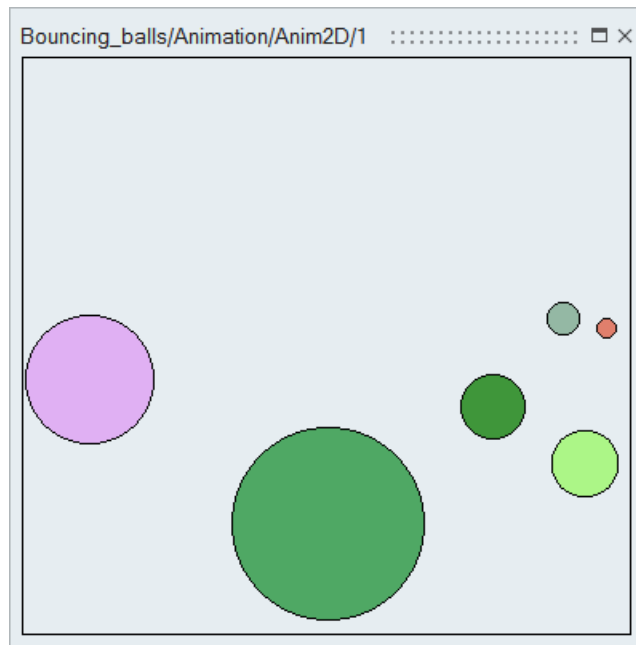

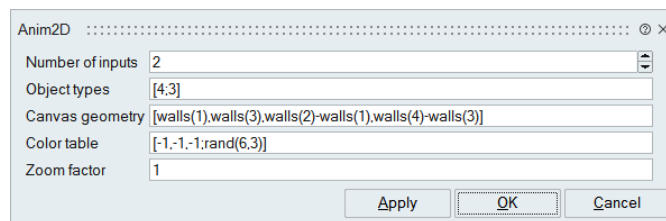Figure 20.3: The animation Figure.

Figure 20.4: The parameters of the **Anim2D** block. The rectangle is not filled (its content is transparent) and the ball colors are chosen randomly.

20.2.2 Control of tank water levels

The model considered here, `lvel.scm`, is made up of two connected water containers. The first container is supplied with water by a constant flow pump which can be turned on and off. This container is connected via a pipe to a second container. The second container also has an exit pipe, which can be opened and closed by a valve. It is opened at random times and closed after a constant delay.

A simple controller is used to keep the level of water in the second container close to a given value while avoiding the water level in the fist container to go over a maximum value.

Denoting the water levels in the two tanks respectively h_1 and h_2, the flow f_1 from container 1 to 2 can be expressed as follows

$$f_1 = \alpha\sqrt{2g(h_1 - h_2)}$$

and the flow out of the second container is

$$f_2 = \begin{cases} \beta\sqrt{2gh_2} & \text{if outflow valve is open} \\ 0 & \text{otherwise} \end{cases}$$

where the parameters α and β depend on the pipes geometry. The water levels in the containers depend on the flows as follows

$$\begin{aligned} \dot{h}_1 &= \gamma(f_{\text{in}} - f_1) \\ \dot{h}_2 &= \delta(f_1 - f_2) \end{aligned}$$

where f_{in} is a constant f_0 if the pump is on and 0 if it is off. The parameters γ and δ depend on container surface areas.

The water levels are measured by two sensors. The sensor output y_i depends on h_i, $i = 1, 2$:

$$\dot{y}_i = \tau(h_i - y_i), i = 1, 2.$$

The parameter τ corresponds to the time response of the sensors.

The control is of type hysteresis where the pump is turned on when y_1 is below the maximum allowed and y_2 below h_{down}. It is turned off if y_1 goes above the maximum allowed or y_2 above h_{up}.

The full model is illustrated in Fig. 20.5.
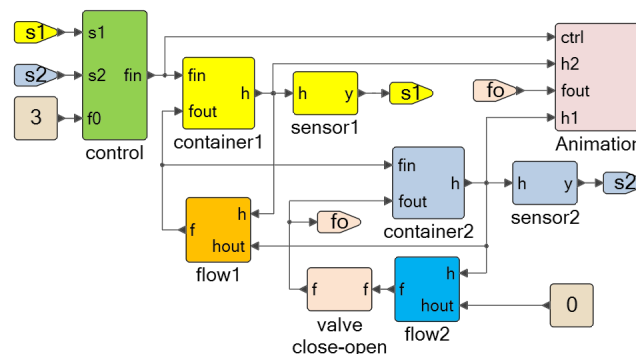
Figure 20.5: The two water container level control model.

The container and flow diagrams are given in Figs. 20.6 and 20.7. The Animation block content is illustrated in Fig. 20.8.
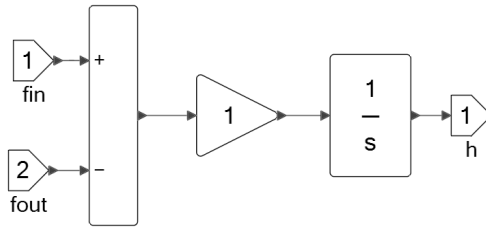
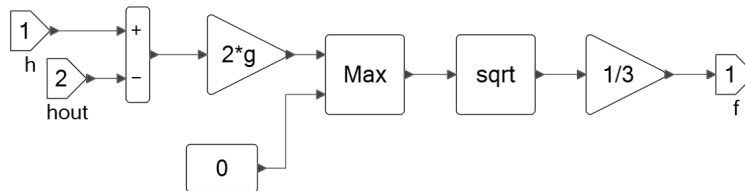Figure 20.6: The diagram used to model the containers.

Figure 20.7: The diagram used to model flows.

The **Anim2D** block is used here to draw polylines and rectangles to illustrate not only the water levels in the containers but also the opening and closing of the exit valve and the state of the pump. A snapshot is given in Fig. 20.9.

20.2.3 Rolling disk on the ground

The demo `wheelsim.scm` contains an animation showing a wheel (disk) rolling on the ground. The 3D animation is obtained by projecting the 3D geometry into 2D using a projection matrix. The model top diagram is shown in Fig. 20.10 and the modified Animation block in Fig. 20.11. This modification is made to create snapshots of the wheel at regular times to obtain the picture shown in Fig. 20.12. The snapshot times correspond to events received by **Anim2D** block on its second input activation port.

One other change is required to make the snapshots visible: the type of the filled polyline representing the floor must be set to -1. Otherwise the floor is drawn at every step snapshot overshadowing all the previous snapshots of the wheel. This change is shown in Fig. 20.13.
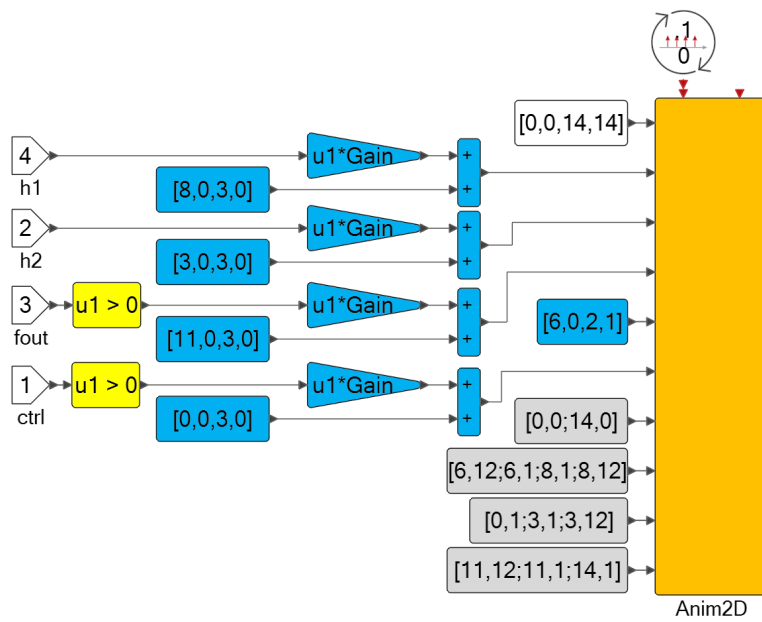
Figure 20.8: The content of the Animation block.
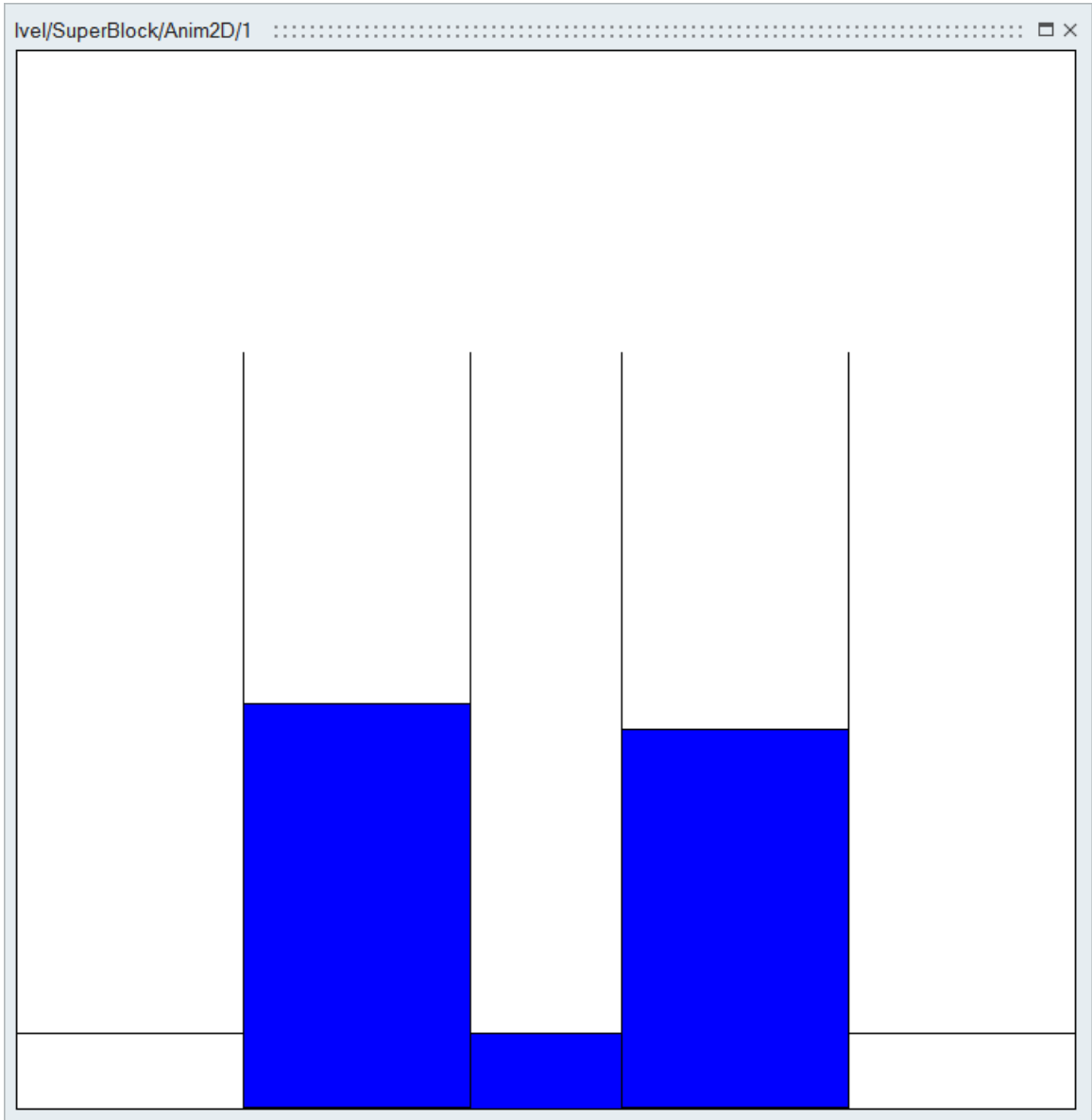
Figure 20.9: A snapshot of the animation. In this case the left and right rectangles are white because the flow in (controlled by pump) and flow out (controlled by the exit valve), are zero.
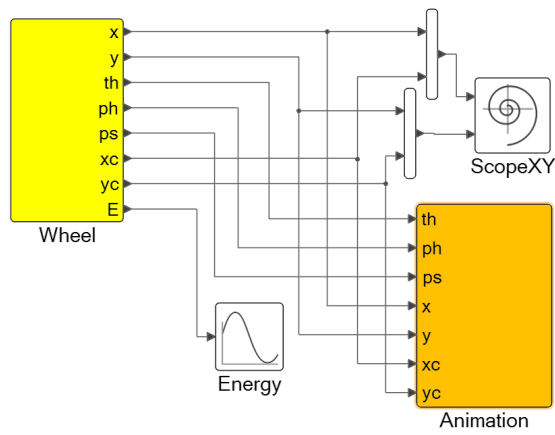
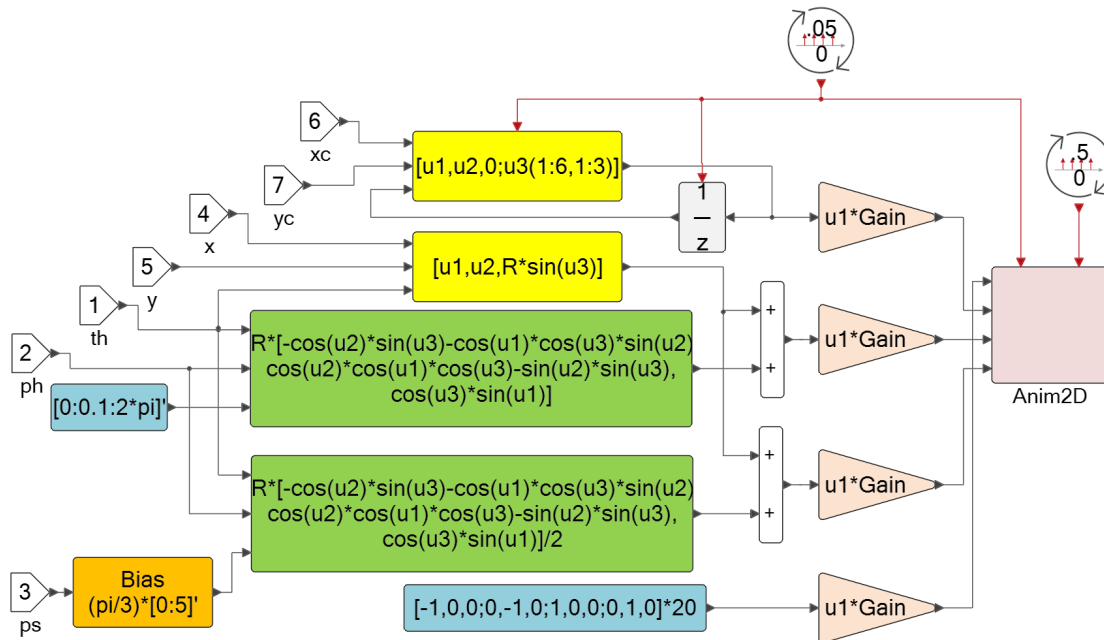Figure 20.10: The top diagram of the wheelsim demo.
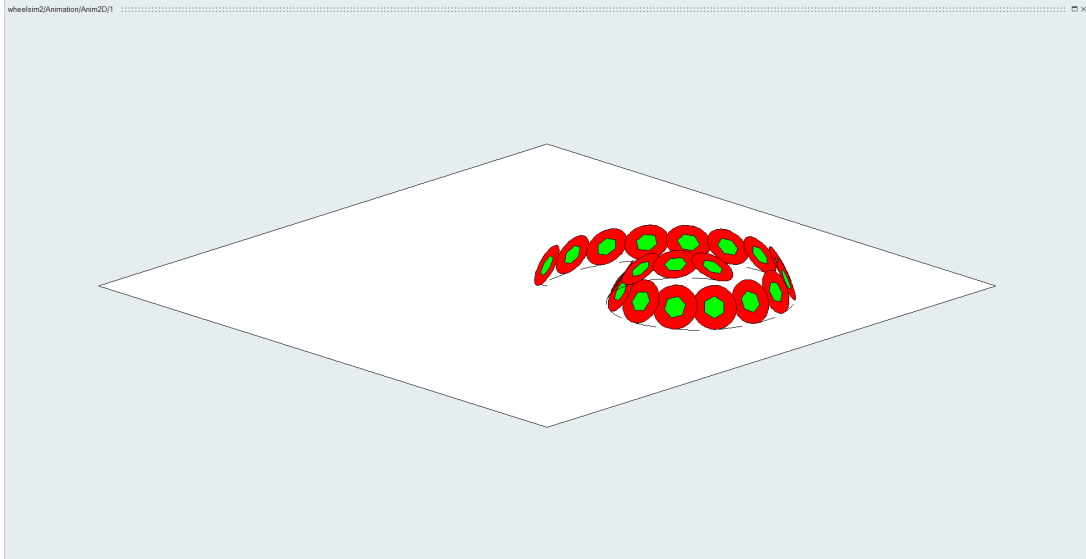

Figure 20.11: The content of the Animation block.

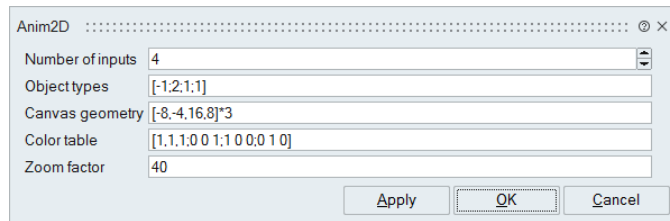Figure 20.12: The final image of the animation.

Figure 20.13: The modified dialog box of the **Anim2D** block.

Chapter 21

Blocks to execute OML and Python scripts interactively

The blocks **ExecOMLScript** and **ExecPythonScript** can be used in Activate models to run scripts, respectively in OML and in Python. The script associated with the block is ran by the user in the block diagram edtior interactively through the block GUI. Unlike other **Twin Activate** blocks, these blocks do nothing during simulation.

The main usage is to allow users to initialize the environment, for example by running a script to linearize a plant and construct a controller before running a simulation to validate the controller.

Unlike the Initialization script, which is executed at every simulation, the **OML** script in **ExecOMLScript** is executed on demand. Such scripts can also be executed directly in **OML** prior to running simulations. The advantage of using a **ExecOMLScript** block is that the model is self contained (the script is included in the model).

Two demo models are provided using the **ExecOMLScript** block, one for linearization and the other for optimization. A demo is also provided to show the usage of the **ExecPythonScript** block.

21.1 ExecOMLScript blocks: examples

In most cases, the script associated with the **ExecOMLScript** block in the model must be executed before running a simulation. The execution of the script creates **OML** variables that are needed by the simulator.

21.1.1 Controller design based on linearization

The `inv_pendulum_lin_OML_exec.scm` model contains the classical inverted pendulum on a cart system with a linear controller modeled by a state-space block. The controller (At, Bt, Ct, Dt) matrices are defined by the structure `_syslin`, if defined in the base environment. Otherwise they are defined so that the associated transfer function is zero (so no control). This is done in the top diagram Context script as follows:

```
_syslin=GetFromBase('_syslin',[]);
if isempty(_syslin)
  At=[];
```

```
  Bt=zeros(0,2);
  Ct=zeros(1,0);
  Dt=zeros(1,2);
else
  At=_syslin.a;
  Bt=_syslin.b;
  Ct=_syslin.c;
  Dt=_syslin.d;
end
```

The construction of the `_syslin` structure is done by linearizing the Super Block containing the model of the cart-pendulum system and using the resulting linear system to construct an observer-based controller based on pole placement. This is done by the **OML** script associated with the **ExecOMLScript** block. By executing this script, the controller is computed, placed in the `_syslin` structure and saved in the base environment (the **OML** script runs in the base environment):

```
clear('_syslin');
% vector of input port indices considered for linearization
inps = 1;
% vector of output port indices considered for linearization
outs = [1,2];
% Modified context
ctx=struct; ctx.z0=0; ctx.th0=0; ctx.phi=0;

model=bdeGetCurrentModel;
% Selected Super block to linearize
superblock = 'pendulum';

[A,B,C,D]=vssLinearizeSuperBlock(model,superblock,inps,outs,0,ctx);
n=size(A,1);
Kc=-place(A,B,-1*ones(1,n));
Kf=-place(A',C',-2*ones(1,n)); Kf=Kf';

At=A+B*Kc+Kf*C+Kf*D*Kc; Bt=-Kf; Ct=Kc;

_syslin.a=At; _syslin.b=Bt; _syslin.c=Ct; _syslin.d=[];
```

If the model is simulated without running the script, the cart pendulum system is simulated without control and as expected the pendulum is not stabilized. On the other hand, if the script is run before, the pendulum is stabilized.

21.1.2 Controller design based on optimization

The `car_min_time_opt_OML_exec.scm` contains the model of a vehicle. The objective is to find the best gear ratios so that the vehicle can travel 1 Km in least amount of time. The optimization is performed over the gear rations `gear` and the gear changing times `T`.

This model has been used and explained earlier in the Optimization chapter. Here the script performing the optimization is placed inside an **ExecOMLScript** block. The script finds the optimal values of `gear` and `T`. They are used during simulation is already computed, otherwise default values are used. This

is done in the Initialization script of the model as follows:

```
T=GetFromBase('T',4*ones(4,1));
gear=GetFromBase('gear',[3.2;1.8;1.3;1;.8]);
```

21.2   ExecPythonScript blocks: example

The demo model `reinforcement_learning_python_exec.scm` considers again the cart pendulum system but here the controller is designed using reinforcement learning.

The reinforcement learning process used for the controller design of this inverted pendulum on a cart system is based on the cart-pole system implemented by Rich Sutton et al. The code is inspired by the 'gym' environment CartPole and the corresponding example in `stable-baselines3` package.[1] The Python code installs the required Python packages if needed.

The learning code is run by executing the Python script inside the **ExecPythonScript** block (this is done double-clicking on the block and 'Execute'-ing the Python script). The execution of the script takes a few minutes; at the end the result is saved in the model's temporary folder. At this point the model can be simulated. The controller obtained from the learning process is loaded by the **PyCustomBlock** and used during simulation to implement the feedback controller.

This implementation of the reinforcement learning requires that the model of the plant (cart-pendulum in this case) be expressed in Python. So the plant model is actually created twice (once in **Twin Activate** inside the cart pendulum Super Block, and once in Python). So there is a risk of discrepancy between the models.

An alternative to the above approach is to use the **Twin Activate** model to create the code used by reinforcement learning. This can be done by using the code generation capabilities of **Twin Activate** to create a C code for the corresponding Super Block, and interface it in Python. In this approach not only the plant model is created only once (so simplicity and no risk of discrepancy) but the performance is also improved since the reinforcement learning program runs simulations in C code instead of Python code.

The **Twin Activate** model `reinforcement_learning_python_Pcode.scm` contains an implementation of this approach. The OML script in the ExecOmlScript block must be executed first to generate the C code for the cart pendulum Super Block. This code is compiled and linked with Python. Then the Python Script in the Reinforcement learning block must be executed. This runs the reinforcement learning algorithm that creates the controller. Finally running the simulation of the model shows the result of the application of the controller.

---

[1] https://pypi.org/project/stable-baselines3/

Chapter 22

Code generation and export

22.1 Introduction

Code generation is used to produce C code from a **Twin Activate** Super Block capturing its dynamic behavior. The generated code can be used for various applications:

- Creating new blocks: the generated code can be used for the simulation function of a **CCustomBlock** or a new basic **Twin Activate** block. These blocks can then replace the original Super Block with better performance and with the possibility to hide its content; further IP protection can be provided by not providing the source code of the block.

- Exporting to other simulation environments: the Super Block can be exported as a block or a functional unit to other simulation environments such as **Altair Embed** and **PSIM**, and a large number of other simulation environments natively or through FMUs based on the FMI standard.

- Generating standalone applications: for Windows and Linux operating systems, and for embedded usages for specific targets such as Arduino.

Two independent code generation technologies are provided in **Twin Activate**: one closely tied to the **Twin Activate** simulator, mimicking its behavior by relying on the libraries used by the simulator, in particular the libraries of the simulation functions of the blocks. The other, generates specific, compact, and highly efficient code with virtually no dependence on external libraries.

22.1.1 Code generation based on block simulation functions

For this code generator, also referred to as the *standard code generator*, the generated code is a hard-coded version of the actions taken by the simulator. The generated code uses the same functions used by the simulator from the core **Twin Activate** libraries. Its performance is not significantly higher than the simulation (the orders of block executions are already precomputed for simulation: no online scheduling is performed). It is primarily used for the implementation of Atomic Super Blocks[1] within the **Twin Activate** environment and code hiding for IP protection of new blocks and exported units.

The generated code, which is not intended to be inspected or even made available, has dependencies on **Twin Activate** libraries. These libraries must be distributed when the code is exported. This is the reason why the exported FMUs using this code generation technology are packed with many shared

---

[1]When a Super Block is declared Atomic, its content is no longer expanded in the model for compilation; it is compiled separately into a basic block with certain assumptions about its activation.

Targets	Standard CG	Inlined CG	Limitations
Twin Activate block	YES	YES	Close to full support for classical generator P has Limited support for blocks significantly extended with use of embedded solver
FMU	YES	YES	Limitations due to FMU constraints
Host Standalone	NO	YES	At most one discrete activation supported
Python	NO	YES	At most one discrete activation supported
Embed block	NO	YES	Continuous-time activation only supported by using the embedded solver

Table 22.1: Code generator (CG) supports for different targets.

libraries.

Since the generated code operates very similarly to the way the simulator operates, and uses the same block simulation functions, its behavior in most cases is identical to that of the original Super Block during simulation and supports all existing blocks, including FMU and Modelica blocks. Some blocks however are not supported when the generated code is destined to be used outside the **Twin Activate** environment, for example blocks relying on the graphical functionalities such as **Scope** blocks, or the **OML** or **Python** custom blocks requiring interpreters at run time.

22.1.2 Inlined Code generation (P code generator)

Unlike the standard code generator, the inlined (also called P) code generator does not rely on **Twin Activate** library of block simulation functions. Instead of using these generic block simulation functions, it produces custom code depending on block parameters and signal types and sizes, for the blocks present in the Super Block. In most cases the corresponding codes are inlined (no function calls). Consequently, the generated code is more efficient and simpler. Moreover, all memory used by the code is statically allocated.

The P code generator can be used for all code generation applications, but not all **Twin Activate** model constructions are supported by P. Originally developed for generating embedded code for discrete controllers runnig on single clocks, the P code generator has been extended to support continuous-time dynamics and to some extent multiple activation clocks. There are however some limitations on the class of Super Blocks for which code can be generated. These limitations will be discussed later.

The P code generator relies on a code generator for the **OML** language. The simulation functions of the **Twin Activate** blocks are not used by P; instead the dynamics of the blocks are expressed in **OML**, and the **OML** codes of the blocks present in the Super Block are used for generating code.

22.1.3 Support for user defined blocks

In some applications, for the construction of the model there is need for blocks that are not available in **Twin Activate** libraries. For that, custom blocks can be used to define new blocks where their simulation functions (so their behavior during simulation) can be coded in C, **OML**, **Modelica**, or **Python**.

Super Blocks containing custom blocks however are not all supported by code generators. The standard code generator supports all blocks as long the generated code is used in the **Twin Activate** environment. But only **CCustomBlock** can be used if the generated code is exported, for example as an FMU.

The P code generator does not support **CCustomBlock**. Custom blocks however can be constructed using a special custom block: **PCustomBlock**. In this block the simulation behavior of the block is expressed in **OML**. The **OML** code is used for model simulation and for code generation.[2] The usage of this block is explained in Section 22.6.3.

22.2 Code generation and export using the graphical user interface

Both code generators operate on Super Blocks. So, if it is not already the case, the part of the model for which code is to be generated must be placed inside a Super Block. Code generation can then be applied to the Super Block.

Various code generation operations are available through **OML** APIs but they can also be realized through a unique GUI; this simplifies considerably the code generation operation. This GUI, shown in Fig. 22.1, is used to set the target and its corresponding options for code generation operations. The GUI can be invoked for the selected Super Block using the contextual menu.
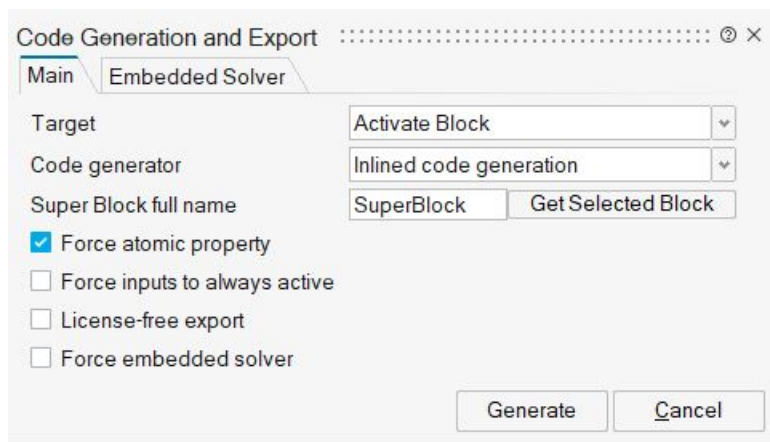
Figure 22.1: Code generation and export GUI. The list of options depend on the choice of the target.

Different targets and their options are presented later in this document. Table 22.1 provides a summary of what targets are supported by each code generator.

Two specific code generation options: "Force atomic property" and "Force inputs always-active" are used to override properties usually defined through the "Atomic" property of the Super Block and the "time dependency" parameter of its **Input** blocks. These can be used more easily than editing the model for setting the properties for code generation purposes.

The parameters of the **Input** blocks may still need to be used, for example if some of the inputs should be considered always-active but not all. They should also be used in case the sizes and/or the types of the inputs cannot be determined by the compiler.[3] But these cases are exceptional.

The Atomic property of the Super Block and the "time dependency" parameters of the **Input** blocks can have significant consequences on the resulting generated code for the Super Block. The activation

---

[2]Similarly to **OmlCustomBlock**, the **PCustomBlock** block's actions during simulation are realized by interpreting **OML** code, but its performance is in general even lower than **OmlCustomBlock**. However, unlike the latter, the **PCustomBlock** is supported by the P code generator, providing highly efficient code.

[3]The size and type information is determined automatically in most cases by the code generator. This is done by performing first a compilation of the full model.

inheritance rules in particular are different for Atomic and non-Atomic Super Blocks. When the Super Block is Atomic, and already explicitly activated through an input activation port, then the activations associated with the input signals are ignored (no activation inheritance). And in case the Super Block has no explicit activation input, then the inherited activation reduces to a single activation resulting from the union of the activations of the input signals.

Since the code generator generates a code for each activation combination separately, in case of an Atomic Super Block, the complexity of the generated code is independent of the number of Super Block inputs. Even if in very special cases, treating a non-Atomic Super Block as Atomic, may change the behavior of the block, the Atomic property is the proper choice in most situations. Otherwise, the complexity of the generated code exponentially grows with the number of inputs.[4]

The "time dependency" parameters of an **Input** block is another way of controlling the treatment of activations inside the Super Block. The code generator cannot determine if an input signal is an always active signal or not. Even if it could, based on the environment in which the Super Block is used, the result may not be satisfactory. For example an input may expected to be an always active signal but, it had been used with a constant signal in the current model (for example for testing). Only users can provide reliably the setting of this porperty for each block.

Code generation can have different objectives. It can be used to create a new **Twin Activate** block, export an FMU or an **Altair Embed** block, or create a standalone application. Even though the same core C code (called body) is generated for all these targets, each target requires specific interface routines. For example, in case of a **Twin Activate** block target, the body is used in the creation of a **CCustomBlock**; in the case of **Python** target, a **Python** package is constructed providing **Python** functions interfacing generated C functions. Specific information regarding different targets are provided in the following sections.

An important code generation option available for most targets is that of using an embedded solver. The embedded solver converts the super block with continuous-time dynamics into a purely discrete-time system with periodic activation by integrating the solver algorithm inside the generated C code. This option is not discussed here. See Section 22.6.4 for details.

22.2.1 Twin Activate block target

The most common target of the code generator is "**Twin Activate** block". The result, which is a new Super Block including the **CCustomBlock** containing the generated C code, replaces automatically the original Super Block, resulting in a ready to be used software in the loop (SIL) model. This moel can immediately be tested by running simulations. This makes this target useful as a first step for validating the generated body even when the actual target is not "**Twin Activate** block".

Note that this operation modifies the current model, so if the new model needs to be saved, it should be saved under a different name to avoid writing over the original model file. Of course, the original model can be recovered after this operation using the undo operation.

Both code generators can be used for this target, and the Super Block may be Atomic or not. The time dependency parameters of the **Input** blocks can also be arbitrarily set. These latter properties can also be overridden using the corresponding checkboxes in the GUI to force them to specific choices.

In case of the P code generator, there are restrictions on the type of blocks that can be present in Super Block, in particular **FMU** and Modelica blocks are not supported.

---

[4]The code generator must consider the possibility that the eventsactivations associated with the inputs are both potentially synchronous and asynchrnous. So, for n inputs, $2^n - 1$ activation scenarios will be considered.
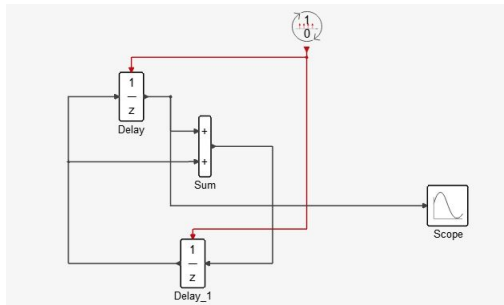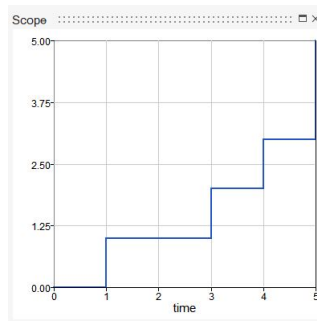
Figure 22.2: A generator of the Fibonacci sequence.

Figure 22.3: The resulting Fibonacci sequence.

Example   In this example, a simple discrete-time model generating the Fibonacci sequence[5] is converted to a block using the P code generator. The model is shown in Fig. 22.2; the delay states are intialized to 0 and 1. The simuation result is shown in Fig. 22.3.
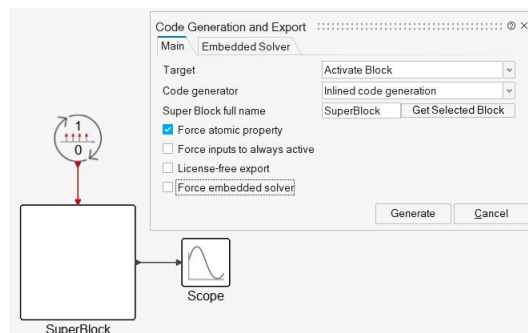

Figure 22.4: Code generator applied to the generator of the Fibonacci sequence model to create a block.

To create a Fibonacci-sequence-generator block, the part of the model creating the sequence is isolated and placed in a Super Block. The code generator GUI can then be opened using the contextual menu of the Super Block, and parameterized as shown in Fig. 22.4. In this case the Super Block has no input and only one activation input port, so the Atomic and time-dependency options do not matter.

---

[5]A sequence of numbers where each number is the sum of the two preceding ones, startig with 0 and 1.

22.2.2 FMU target

The FMU export currently only supports FMU 2.0 standard for 64 bit architecture. The code however can be generated with both code generators. If the inlined (P) code generator is used, the exported FMU requires and thus will contain very few shared libraries (if any). Code generator by the classical generator will have many dependencies resulting in a bigger FMU file. The FMU can be exported for Model-Exchange, Co-Simulation or both.

The exported FMU can be re-imported in **Twin Activate**, in particular to replace the original model for validation by simulation. This can be done using the **FMU** block.

The exported FMU for co-simulation uses the same numerical solver which is selected in the model. If the export is done with the P code generator, and the selected numerical solver is not an explicit Euler, expicit Trapezoidal, or, a 4th or 5th order Runge-Kutta (so in particular for all variable-step solvers), explicit Euler solver is implemented for co-simulation instead.

Note that the P code generator does not support all **Twin Activate** blocks, in particular **FMU** and Modelica blocks are not supported.
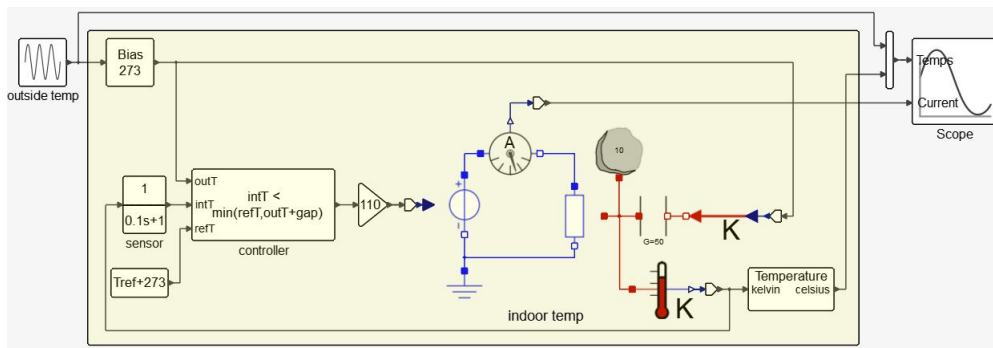
Figure 22.5: A temperature controller applied to a plant modeled using Modelica blocks.

Example   The model shown in Fig. 22.5 contains a temperature controller implemented in **Twin Activate** blocks acting on a plant modeled using Modelica components providing the indoor temperature provided the outside temperature and the electrical current applided to the heating resistance. The control has two parameters: a reference temperature `Tref` and a temperature gap `gap`. The heater is turned on if the current temperature is below both the reference temperature and the outside temperature plus the temperature gap.

The part for which **FMU** should be exported is highlighted. it has one input and two outputs. The input is the outside temperature and the outputs are the indoor temperature and the electrical current used by the heater.
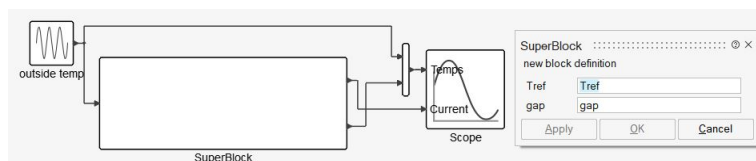

Figure 22.6: The part of the model to be exported is placed in a Super Block and masked.

For exporting the **FMU**, the hightlighted part is placed in a Super Block. Two of the model parameters,

`Tref` and `gap` are defined outside the Super Block, so by masking the Super Block, these parameters become mask parameters, as shown in Fig. 22.6. The variables `Tref` and `gap` are used in the definitions of the mask parameters, so they are *exposed* (become exposable parameters) when code generation is applied. As such, these variables become parameters of the generated code. In particular for **FMU** export, this means that the exported **FMU** will have them as **FMU** parameters. For information on exposable parameters, see Section 22.5.
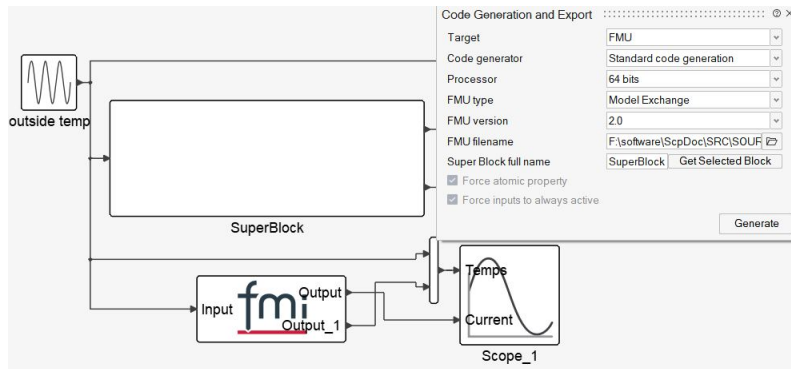
Figure 22.7: **FMU** is exported and then used in the same model to compare simulation results.

The exported **FMU** can be re-imported and tested in **Twin Activate** as shown in Fig. 22.7. The simulation results for the original Super Block and the imported **FMU** are seen to be identical, shown in Fig. 22.8.
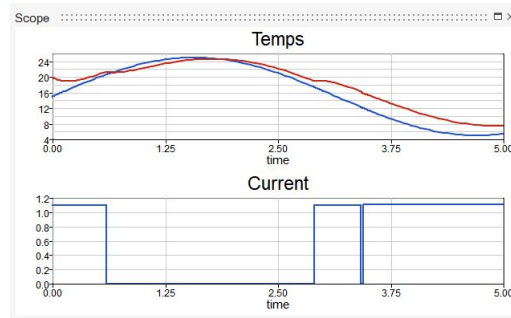
Figure 22.8: simulation result for both the original Super Block and the generated **FMU**.

22.2.3 Host Standalone target

Code for standalone applications can only be generated with the P code generator and currently only for a 64 bit architecture. The Generated files are: *body.c*, *body.h*, *makefile* and *main.c*. The body, which exports the functions initialize, updateoutput, updatederivative and updatestate, and possibly finish,[6] is the file which is generated and used for all targets.

Currently, this target does not support exposable parameters.[7] Moreover, the content of the Super Block can only be activated by one activation (other than initialization event); either it should be purely

---

[6]The code for finish is only generated if one or more blocks perform specific actions at the end of the simulation. Most blocks don't.

[7]The exposable parameters are explained in Section 22.5.

discrete-time with one activation input port (explicitly or by inheritance, and not necessarily periodic), or contain continuous-time dynamics with all input block ports having time-dependency properties, and no activation input ports.

The file *main.c* contains an "example" main function which illustrates the usage of the functions exported by body. This proptotype, specifically generated for the model, can be compiled with the provided makefile producing an executable file *main.exe*. The exe file reads inputs from the keyboard and writes the outputs to the stdout. The simulation is performed with a simple Euler solver included in *main.c*.

For standalone applications, user will develop its own code by taking avdantage of functions exported in body. The generated *main.c* should be considered a prototype showing how to use the functions provided by body.

For more information on the files generated for this target see Section 22.3.2.

Example   As an example, consider the moon landing game **Twin Activate** demo. The dynamics of the moon lander is considered for the creation of a standalone application for the host, for example for an interactive game.

The first step as in any code generation application consists of isolating the part of the system for which code is to be produced. This is done as shown in Fig. 22.9. The input signal to this part corresponds to the rate of fuel used to slow down the lander over the next time step. There are four outputs: the second, third and forth provide the altitude, speed and the remaining fuel; the first output is an indicator taking values 0, 1 and 2. The value 0 means the lander has not reached the surface, 1 means the lander has landed safely, and 2 indactes a crash.
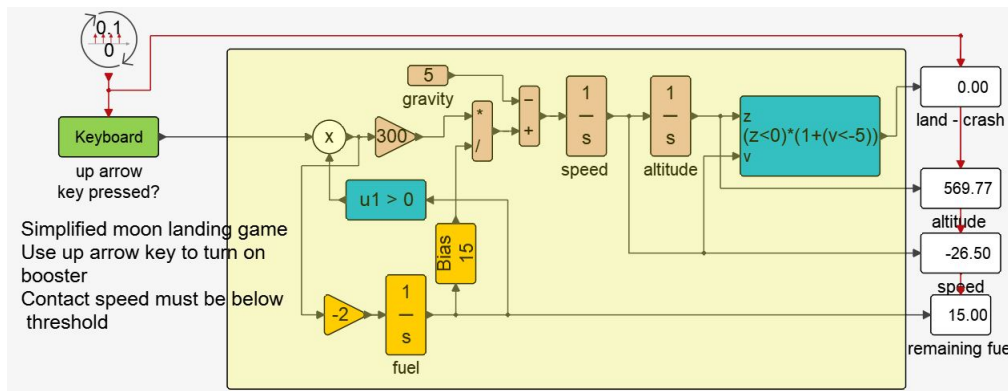

Figure 22.9: Moon landing game reorganized to highlight the part used for code generation.

The dynamics of the lander is placed in a Super Block as shown in Fig. 22.10. Code can then be generated for the Host Standalone target by setting the forced Atomic and input time-dependency options. The generated code is then compiled with the provided makefile, assuming the host is equipped with proper C compiler.[8].

The main code is generated as a prototype, to be customized for the specific usage (for example a game). But the provided default main, which reads data from the keyboard and writes to the standard output, can already be used as an interactive game played in the console, as shwn below

---

[8]Unlike for other targets, the generated code is not compiled by **Twin Activate**; in all cases, the main function should be adapted to the application
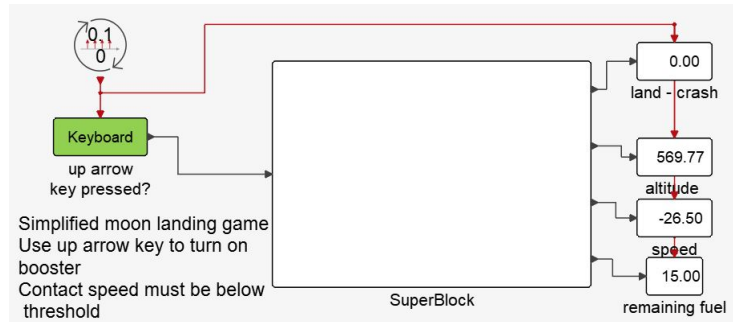
Figure 22.10: The dynamics of the moon lander placed in a Super Block to be used for code generation.

22.2.4 Python target

Despite what the target name may suggest, the generated code which contains the behavior of the Super Block is not in **Python** language. It is C (based on `body` used in all P targets), but it is interfaced with **Python**.

The **Python** code is for specific usages, in particular to be used for optimization and learning applications using specialized **Python** libraries. The Super Blocks supported by this target can have continuous-time dynamics and with exactly one activation input port. In most applications, the input activation is used to set or reset the continuous-time states of the Super Block. But the Super Block may also have discrete-time states and contain discrete-time dynamics.

The **Python** file, defining four functions `init`, `reset` and `step`, and, `finish`, exposing the `body` functions for co-simulation usage, is available in the selected folder after code generation. A **Python** test file is also generated showing the usage of these functions. The folder also contains the shared library file interfaced (transparently) with **Python** and a number of other files used to create the shared library. The latter files can be removed if needed.

See Section 22.3.2 for more details.

Examples   The moon landing example used for illustrating the usage of the standalone target is used for the **Python** target. In this case the `reset` function is not used, and the simple game can be implemented as follows:

and used as follows

```
Enter control value : 0.4
time =  1
Altitude =  579.5629558289684
Speed =  -20.951173656529186
Fuel =  14.2

Enter control value : 0.8
time =  2
Altitude =  560.1158292334636
Speed =  -17.52201210385048
Fuel =  12.599999999999998

Enter control value : |
```

A more advanced usage of the **Python** target is presented in Section 22.3.2.

22.2.5   Altair Embed block target

The **Altair Embed** target can be used to export **Twin Activate** Super Blocks as **Altair Embed** blocks, which can be used both for simulation and code generation in the **Altair Embed** software. The underlying code is the body generated by the P code generator. Currently, systems with continuous-time dynamics (so in particular system with continuous-time states) are not supported The Super Block should also have at most one activation port, so that the dynamics of the system is synhronous.

Altair Embed blocks can be exported both for 32 and 64 bit **Altair Embed** environments. The exported block is automaticaly compiled, resulting in the corresponding dll file,[9] ready for use in **Altair Embed**. This process requires the full path of the **Altair Embed** installation to be provided.

Example   The Fibonacci example is used again. The **Altair Embed** export is realized as shown in Fig. 22.11. The generated code can be compiled using the provided make shell script. The resulting file `SuperBlock.dll` can then be used as an **Altair Embed** block and used in **Altair Embed** models, for example as shown in Fig. 22.12.
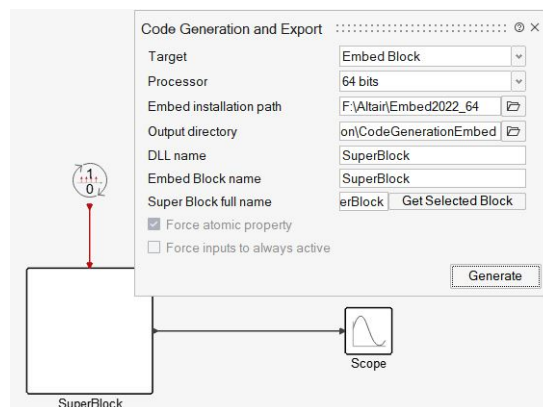

Figure 22.11: The Fibonacci model used to create an **Altair Embed** block.

22.3   Code generation through OML APIs

Code generation functionalities, both standard and P, are available also via **OML** functions, which can be used interactively from the **OML** command window or programmed in **OML** scripts. The main argument

---

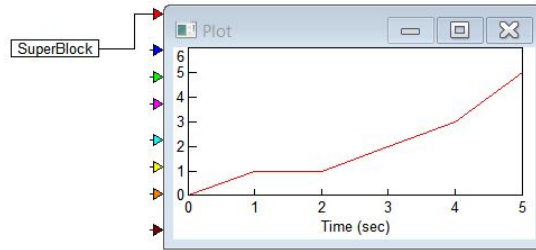[9]**Altair Embed** is not available under Linux.

Figure 22.12: The resutling Fibonacci block used in **Altair Embed**.

to these functions is the Super Block for which code is generated. These functions, which may have additional optional arguments, are presented below.

22.3.1   Classical code generator

Generating a CCustomBlock

The following function is used to generate C code for the Super Block specified by the input argument, replacing it in the current model with a new Super Block including the corresponding **CCustomBlock**:

```
cblock = vssGenerateCBlock(bdeblock)
```

The output argument `cblock` is a handle to the new Super Block, which replaces the original Super Block in the diagram, after the execution of this instruction.

The input argument `bdeblock` is the handle to the Super Block for which code is generated. It can be obtained for example by selecting the Super Block in the current diagram and using the following command:

```
bdeBlock=bdeGetSelectedBlock(bdeGetCurrentDiagram());
```

The handle can also be obtained and used as part of an **OML** script automating the code generation process as in the following example.

This function is used in the Code generation and export GUI for the **Twin Activate** block target.

FMU export

FMU can be exported by the following function, where all the input arguments are optional:

```
fmufilename=vssGenerateFMUBlock(bdeBlock,options);
```

The first argument `bdeblock` is the handle to the Super Block for which code is generated. If it is omitted, the selected block in the current diagram is used.

The second argument `options` contains two fields:

- `mecs`: Integer, indicating what type of **FMU** should be generated.
- `fmufile`: String. Full name of the exported **FMU** file.

The field `mecs` is used to specify what type of **FMU** should be generated:

- `mecs=0`: model exchange only 'default value'

- `mecs=1`: co-simulation only

- `mecs=2`: both model exchange and co-simulation

The field `fmufilename` can be used to choose where and under what name the generated **FMU** should be saved. This optional string can include the full name of the destination file. If an empty string is used, an interactive GUI opens to select the folder where to save the generated **FMU**.

This function is used for the **FMU** target in the Code generation and export GUI.

22.3.2 Inlined code generation APIs

The code generation functions available by the classical code generator are also available by the P code generator.

Generating a CCustomBlock

For generating a new Super Block and using it to replace the original Super Block in the diagram, the following function can be used.

`vssCompileToInlinedC(sblock,options)`

The first input argument `sblock` can be either the name of the Super Block, its handle, or a cell containing multiple block names and/or block handles. If omitted, the selected Super Block in the diagram is used.

The second input argument `options` is an optional **OML** structure with the following fields:

- `setinputsdept`: Vector of type logical setting the time dependency properties for inputs. The default value is false.

- `setatomic` Logical. If true, the Super Block is considered having Atomic property. True is de default value.

- `comments` Logical. If true, block annotations are used to place comments in the generated C code. The default value is true.

Exporting FMU

For exporting the generated code as an **FMU**, the following function is used

`vssPProjExportToFMU(sblock,options)`

The first input argument `sblock` can be either the name of the Super Block or its handle, or a cell containing multiple block names and/or block handles. If omitted, the selected Super Block in the diagram is used.

The second input argument `options` is an optional **OML** structure with the following fields:

- `mecs`: Integer, indicating what type of **FMU** should be generated.

- `fmufile`: String. Full name of the exported **FMU** file.

The field `mecs` is used to specify what type of **FMU** should be generated:

- `mecs=0`: model exchange only 'default value'

- `mecs=1`: co-simulation only

- `mecs=2`: both model exchange and co-simulation

The field `fmufilename` can be used to choose where and under what name the generated **FMU** should be saved. This optional string can include the full name of the destination file. If an empty string is used, an interactive GUI opens to select the folder where to save the generated **FMU**.

This function is used for the **FMU** target in the Code generation and export GUI.

Generating a standalone code

The P code generator can also be used to generate standalone code, which can be compiled and executed under Windows and Linux.

`vssCompileToStandalone(sblock,options)`

The first input argument `sblock` can be either the name of the Super Block or its handle.

The second input argument `options` is an optional **OML** structure with the following fields:

- `folder`: String. The path to the folder where the generate files are saved.

There are 4 files generated for the standalone target:

- `body.c`: C file containing the body of the code generated from the Super Block. It defines in particular functions to initialize, compute the output, compute the derivative (if the Super Block had continuous-time states), and update the dicrete states.

- `body.h`: Include file exporting the functions initialize, updateoutput, updatestate and possibly updatederivative, defined in `body.c`.

- `main.c`: A generic main code which can be built to produce a functional executable. Generic input and output functions are defined in this file, by default reading input values using the C scanf function and writing the outputs using the C printf functions. The activation of the code is controlled by the nexttime function, which by default periodically activates the code every 0.1 unit of time.

  The functions input, output and nexttime defined in `main.c` should be customized for the standalone application.

- `Makefile`: The Makefile to build the exectuable `main.exe`.

Generating code for use in Python

The `vssGenerateSimpleSolver` function is used to generate APIs for performing simulation in co-simulation mode (with integrated solver) for different targets.

`[inouts,outparams]=vssGenerateSimpleSolver(sblock,params)`

The first input argument `sblock` can be either the name of the Super Block or its handle.

The second input argument `options` is an optional **OML** structure with the following fields:

- `python`: Boolean with value true if the function is used for **Python** target.

- `folder`: String. The path to the folder where the generate files are saved.

- `modulename`: String. The name of the **Python** module to be generated.

The output `inouts` is a cell of structures containing the type and sizes of the inputs and outputs. The

output `outparams` contains the folder and modulename used by the function in case the information is not provided in the second input argument.

The **Python** functions provided in the module are

- `init()`: This function is called once at the start of a simulation. It instantiates the co-simulation mechanism (in particular by linking the shared library) and preforms initialization operations (if any) at time 0 for a new simulation.

- `step(inputs, t, dt)`: Simulates the system up to time `t` taking Euler steps of size `dt` for given inputs.

- `reset(inputs, t)`: if needed, simulates the system up to time `t`, then executes the code corresponding to the unique activation of the Super Block.[10] If called with `t` equal to 0, the simulation is initialized and the activation operation is performed at time 0. This operation is often used to restart a simulation with a new continuous-time state.

- `finish`: must be used to end the simulation. This operation performs final actions (if any) and closes the shared library linked in `init()`.

Example   The main use of **Python** target is to generate code to be used with specialized **Python** libraries. In the demo example `reinforcement_learning_ball_beam_python_Pcode`, shown in Fig. 22.13, the controller for a cart on beam problem (a variation of the classical the ball and the beam problem) is constructed by reinforcement learning.

In this model, a **ExecOMLScript** block is used to generate code for the Super Block containing the dynamics of the plant for which a controller is designed. The **OML** script that generates code for **Python** target is as follows:

```
params=struct;
params.python=true;
params.folder=bdeGetCurrentModel;
params.folder=bdeGetModelTempDir(params.folder);
params.folder=fullfile(params.folder,'solver');
params.modulename= 'sb';
vssGenerateSimpleSolver('cart on beam',params)
```

The module `sb` is generated in the model temporary directory under a folder named `solver`.

Once the module is created, the **ExecPythonScript** block is used to perform reinforcement learning using the **Python** library *stable-baselines3*. The result of the learning process is saved in a temporary folder which can be used subsequently by the **PyCustomBlock** `controller` during simulation.

For details, examine the contents of the `Reinforcement learning` and the `controller` blocks.

Core generator

The basic function generating the code of the Super Block and used for various generators presented above is

```
res = vssGeneratePProjectBody(sblock,options)
```

---

[10]Generally reset is used at time t, so no time simulation is performed.

First execute the OML script in the ExecOmlScript block to generate
   C code for the cart on beam super block, compile and link with Python.
Then execute the Python Script in the Reinforcement learning block.
This will take few minutes (longer the first time if Python libraries need to be installed).
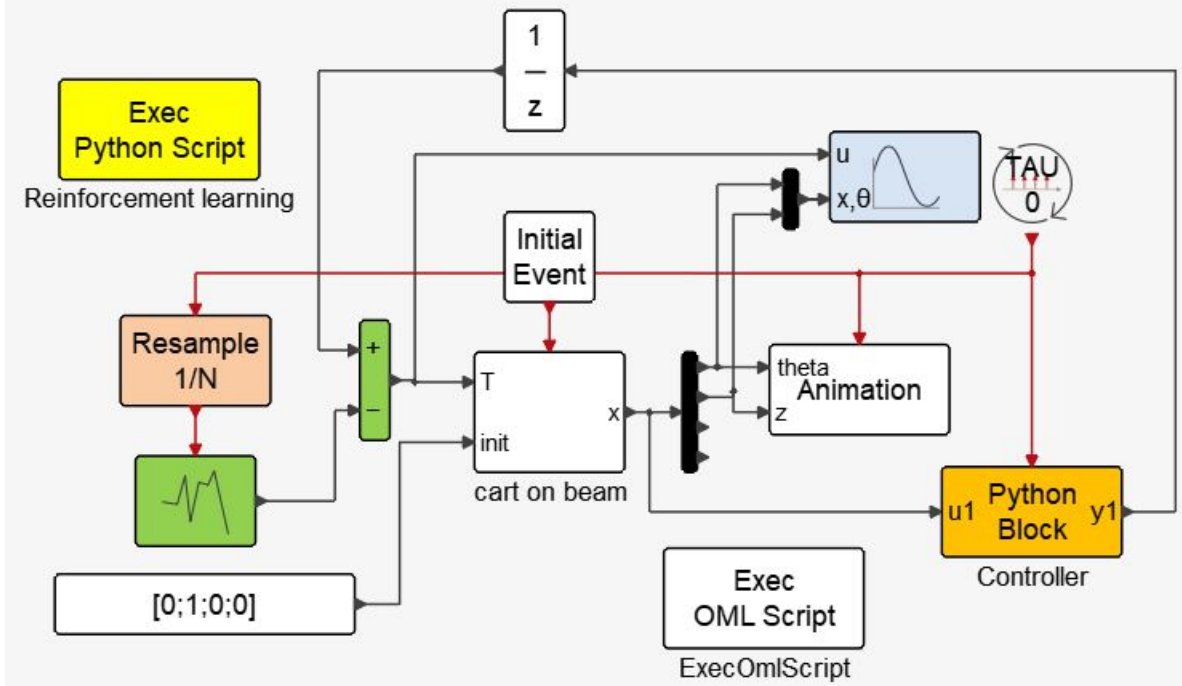Finally run a simulation to see the result.

Figure 22.13: Controller is constructed by reinforcement learning.

This function returns the result of the code generation including the C codes and corresponding auxiliary information in the **OML** structure `res`.

22.4   Code generation options

Generating code for a Super Block reduces the behavior of the Super Block to that of a basic block. But the behavior of all Super Blocks cannot be perfectly replicated by a basic block. A Super Block in **Twin Activate** is a graphical construction and does not impose an "atomic" behavior, unless the Super Block is declared Atomic. In most cases, code generation is performed for Atomic Super Blocks; in fact the P code generator, by default considers the Super Block to be Atomic.

If the Super Block is declared Atomic in the model, the generated code should provide identical behavior. If not, the code generator is done is such a way to make the behavior of the generated code as close as possible to that of the Super Block in the simulator.

The main factor that could create discrepancy between the behavior of the Super Block and the resulting basic block has to do with the way the blocks in the Super Block are activated. The discrepancy can occur in particular if the blocks of the Super Block are not explicitly activated by the activation signals received by the Super Block. In that case the activation during simulation is inherited from the input signals, and, the code generator has to anticipate the activation inheritances by introducing additional

input activations.

22.4.1 Case of Atomic Super Blocks

When the Super Block is declared Atomic, even for simulation, code is generated for the Super Block during model compilation and the Super Block is replaced with a corresponding regular block. So, the generated code should behave exactly as the Super Block does during simulation. This should be the case whether the Atomic Super Block is explicitly activated or not.

Atomic Super Block with no input activation port   If the Super Block has no activation input, then a single activation input port is added to the Super Block, receiving the union of the activations inherited from the Super Block inputs, except for the inputs declared time dependent (corresponding **Input** block with "Time dependency" property).

The resulting Super Block with the activation input is then processed as explained in the next section.

Atomic Super Block with one or more activation input ports   When the Atomic Super Block has activation input(s), no activation is inherited from the Super Block inputs.

In addition to the existing activation links inside Super Block, other activation links are added to the diagram connecting all the activation **EventInput** blocks to all the **Input** blocks, prior to the compilation and code generation for the diagram inside the Super Block.

The Atomic Super Block may have more than one activation input. The input activations are considered asynchronous and discrete; initial and always activation signals should not be used to drive the Super Block via activation signals.

Excluding the case where input activations may be synchronized significantly reduces the complexity of the generated code. If the Super Block has n input activation ports, then n different codes need to be generated, one corresponding to each activation input. Whereas in the general case, the Super Block may be activated in $2^n - 1$ different ways, requiring a specific code for each one.
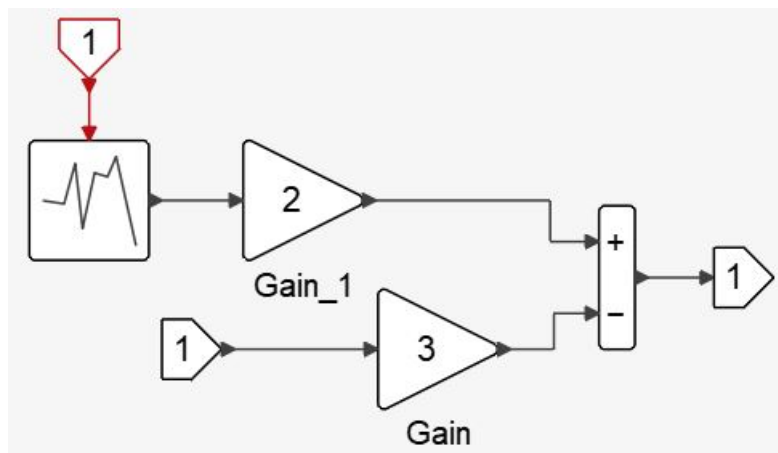
22.4.2 Case of non-Atomic Super Blocks

Figure 22.14: Content of the non-Atomic Super Block.

If the Super Block is not Atomic, activation may occur both through explicit activation signals entering the Super Block and by inheritance. Moreover, code generation considers all potential synchronization possibilities. consider a non-Atomic Super Block containing the diagram illustrated in Fig. 22.14. The **Input** block is not declared time dependent. Subsequent to code generation the diagram is transformed as in Fig. 22.15.
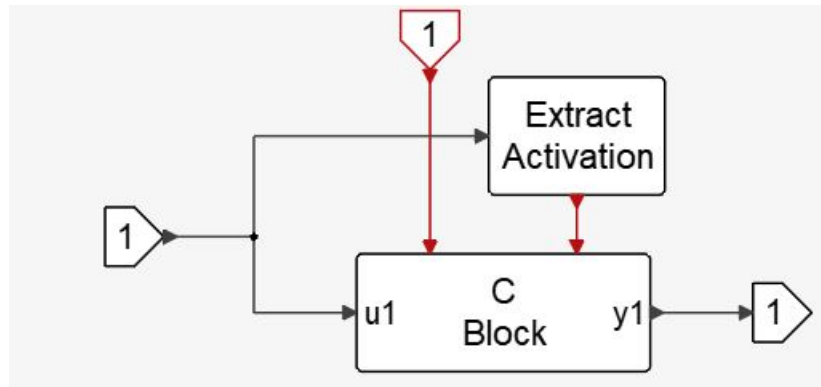

Figure 22.15: Content of the Super Block after code generation.

The generated code, the simulation function of the **CCustomBlock**, contains three different codes associated with the different ways the block can be activated: through its first activation input port, its second activation input port, and synchronously through both.

Note that the possible synchronization of inherited activations and explicit activations is considered in the generated code. This exhaustive treatment of activation signals is done to make sure the behavior of the Super Block remains as before but it is not required in most cases.

The example in Fig. 22.16 illustrates a situation where rigorous treatment of activation signals is required. In this model the Super Block contains blocks with internal states inheriting their activations through different inputs. These inputs are connected to signals with different clocks. Differences are observed if the Super Block is declared Atomic or not, as seen in the **Scope** block. In this case, to
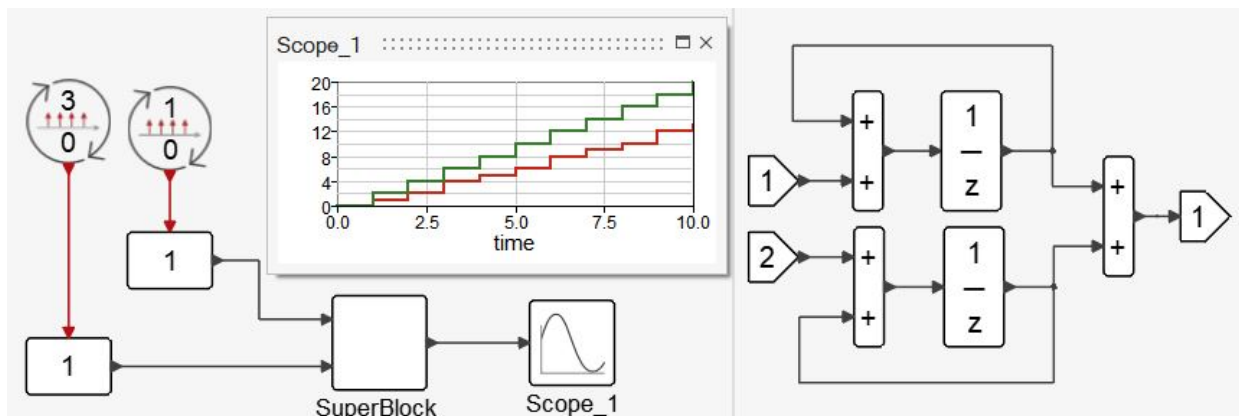

Figure 22.16: Super Block declared Atomic or not could make a difference.

preserve the original behavior, the inhertied activation signals must be considered seperately. In the Atomic case, only their union is used to activate the blocks, resulting in a different behavior.

22.5 Exposable parameters

There is clear distinction between model parameters and signals in **Twin Activate**. Signals in models, which in most cases vary as a function of time during simulation, are exchanged through block ports. Block parameters on the other hand are constant during simulation and are defined by **OML** scripts and expressions which are evaluated when the model is compiled. In particular, the block parameter values are replaced by their numerical values following the execution of the model Initialization script, diagram Contexts and the evaluation of the corresponding **OML** expressions. So, no code corresponding to the evaluation of block parameters is executed during simulation or generated by code generators.

However when a block parameter, or more generally an **OML** variable used in the definition of a block parameter, is defined as exposable, then it is treated as a (constant) signal originating from outside the model. This means in particular that it becomes a parameter of the generated code, so its value can be changed without having to generate a new code. The code generator generates a specific code to account for the value of the exposable parameter; this code is executed only once at the beginning.

22.5.1 Definition

When code is generated for a Super Block, for example, to create a new block or export an **FMU**, the **OML** variables used in the parameterization of the diagram inside the Super Block can be considered fixed, in which case their numerical values are used for the generation of the code, but they can also be considered *exposable*, in which case their values can be modified even after code generation. The exposable parameters of the Super Block can be used as block parameters of the new block created by code generation, and the parameters of the exported **FMU**.

22.5.2 Selection of exposable parameters

There are two ways of designating a variable as an exposable parameter: using the **OML** function exposable, and through Super Block masking.

Selection using the exposable function   Using the exposable function explicitly is one way to define a parameter as exposable. This is a simple way to define a parameter as exposable, anywhere in the Super Block. For example, in the following Super Block, the Gain parameter of the **Gain** block is defined as an exposable parameter named A with the default value of 3.
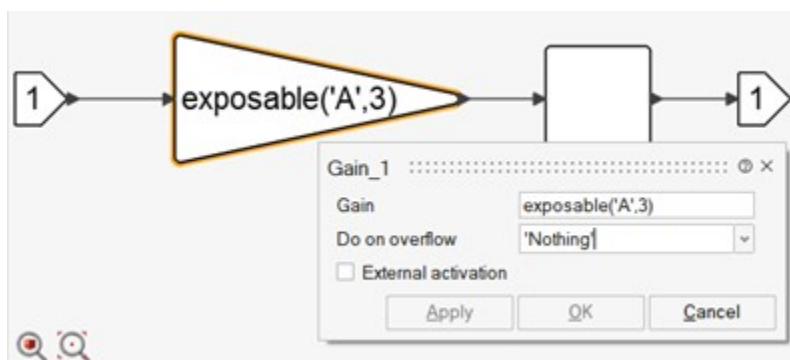

The parameter can also be equivalently defined in the Context of the diagram, as follows:

```
A=exposable('A',3);
```

and used to define the Gain parameter value of the block.

In this example, when code is generated for the Super Block, the parameter `A` (the gain value) is exposed. The second argument of the exposable function defines the default value of the parameter. The exposing of the parameter `A` and its default value can be seen by using any of the code generators to create a **CCustomBlock** or generating the **FMU** of the Super Block:
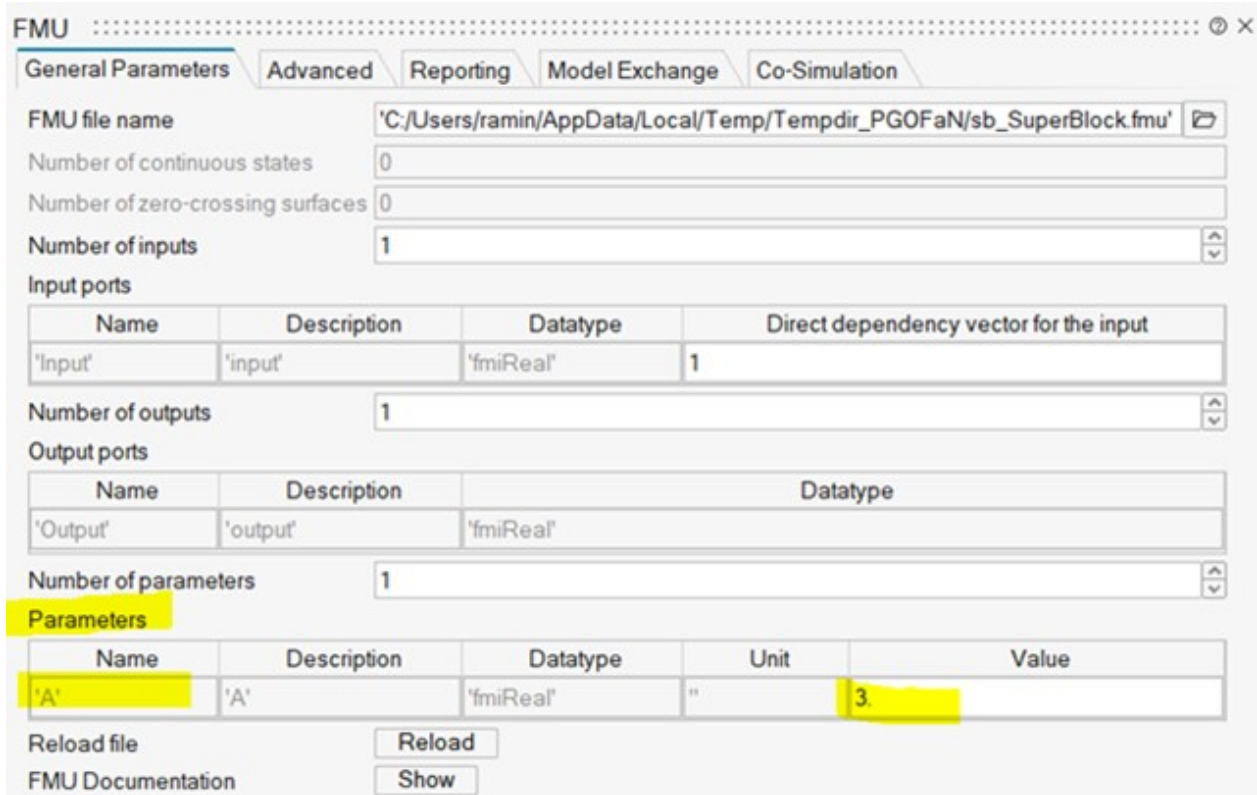

The exposable parameters are defined primarily for code generation purposes but defining them explicitly using the exposable function affects the simulation of the model as well. In particular, if a variable with the same name exists in the **OML** base environment, then the exposed parameter takes the value of this variable for Model Evaluation. So, in the above example, if a variable `A` is defined having a value of 4 in the **OML** base environment, then the simulation uses the value 4 for parameter `A`. If `A` is not defined in the **OML** base environment, then the simulation runs normally as if `A` were not exposed (with value 3).

The exposable function behaves like the GetFromBase function during the Model Evaluation phase. This way of designating a parameter as exposable is useful in some cases, for example when a deep level parameter is to be made exposable without having to modify the model, but in most cases, it is not the most convenient way of defining exposable parameters. The natural way to select exposable parameters is to mask the Super Block (by auto-masking for example), specifying the variables used but not defined inside the Super Block, and then use these variables to select the exposable parameters.


Selection by masking   For an unmasked Super Block, no diagram parameter is considered exposable (other than any explicitly defined using the exposable function, as presented previously). This is the case even if a variable used inside the Super Block is defined outside the Super Block. But in this latter

Altair Twin Activate 2023

case if the block is auto-masked, the variable becomes a mask parameter, and it can then be defined as an exposed parameter if desired.

Consider the following model where the variable `A` is defined in the Model Initialization script and used inside the Super Block (to define the parameter value of the **Gain** block), if the Super Block is not masked, no parameter is exposed and the generated code for the Super Block uses the numerical value of `A=1`. It would be the same as if the value of the parameter of the **Gain** block was set to 1.
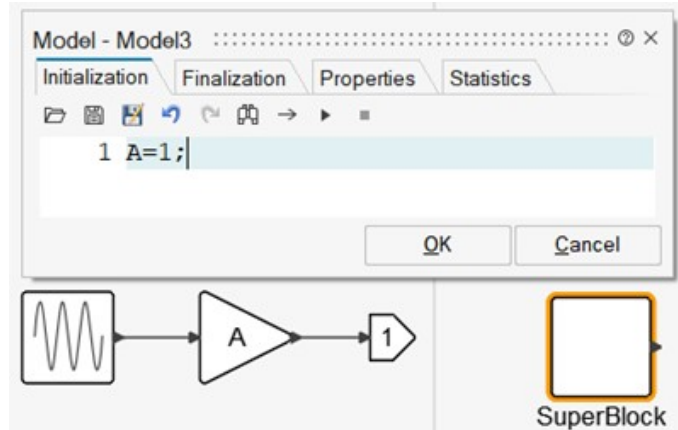
The variable `A` in this case is not defined as an exposable parameter even though it becomes a mask parameter if the Super Block is auto-masked (auto-masking operation identifies the variables used inside and defined outside the Super Block, in this case `A`), as shown below.
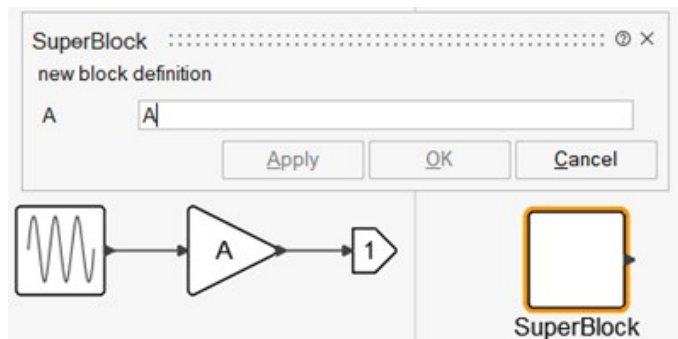
Masking the Super Block is a way to define exposable parameters. In particular, when code is generated for a masked Super Block, all the variables used in the definition of the values of mask parameters are defined as exposable parameters. Note that if the masking is done using the auto-masking operation, the variables correspond exactly to the mask parameters. But, more generally, the expressions defining the values of the mask parameters may contain any variables defined in the model outside the Super Block.

Consider the model shown in the previous figure. The Super Block mask parameter is `A`, and so is its value. In code generation for this Super Block, `A` is considered exposable, and its default value is taken to be 1. Note that `A` is considered exposable because the value of the mask parameter is defined as `A` and not because of the name of the mask parameter, which also happens to be `A` and corresponds to a local variable of the Super Block. So, if the value of the mask parameter `A` is defined numerically as shown below prior to code generation:
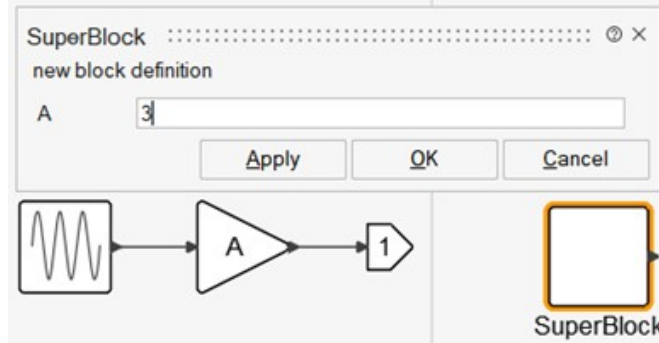
`A` is not exposed when code is generated for the Super Block.

In the following example, on the other hand, where the parameter value of `A` is defined as a function of other variables (`B` and `C` in particular):
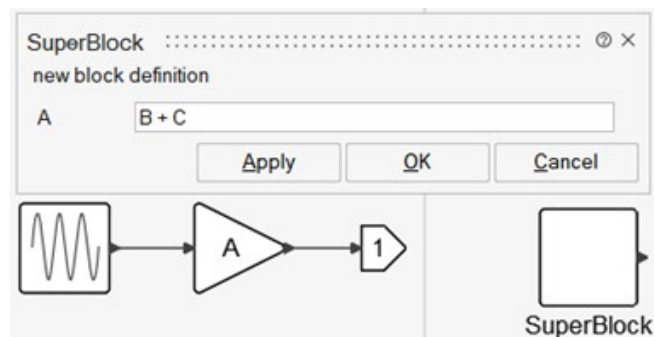
the exposed parameters are `B` and `C` (these variables must be defined outside the Super Block for the model to be correct). `B` and `C` become the parameters of the new block obtained by generating code for the Super Block:
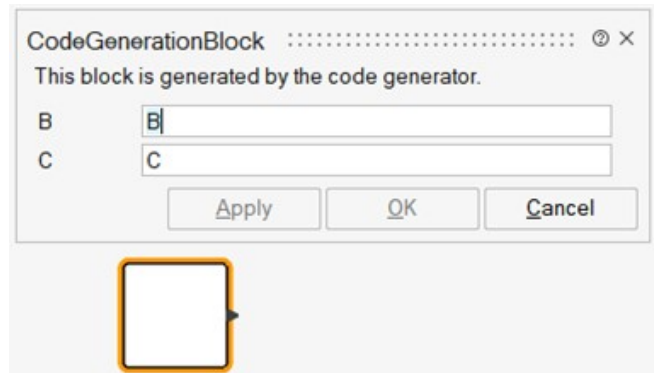
Note that `A` is not exposable in this case; it is simply a local variable of the Super Block.

The two methods for defining exposable parameters (via Super Block mask parameters and the explicit use of the exposable function) can be used simultaneously for the same diagram. The resulting exposed parameters then include both the parameters exposed by masking and by the use of the exposable function.
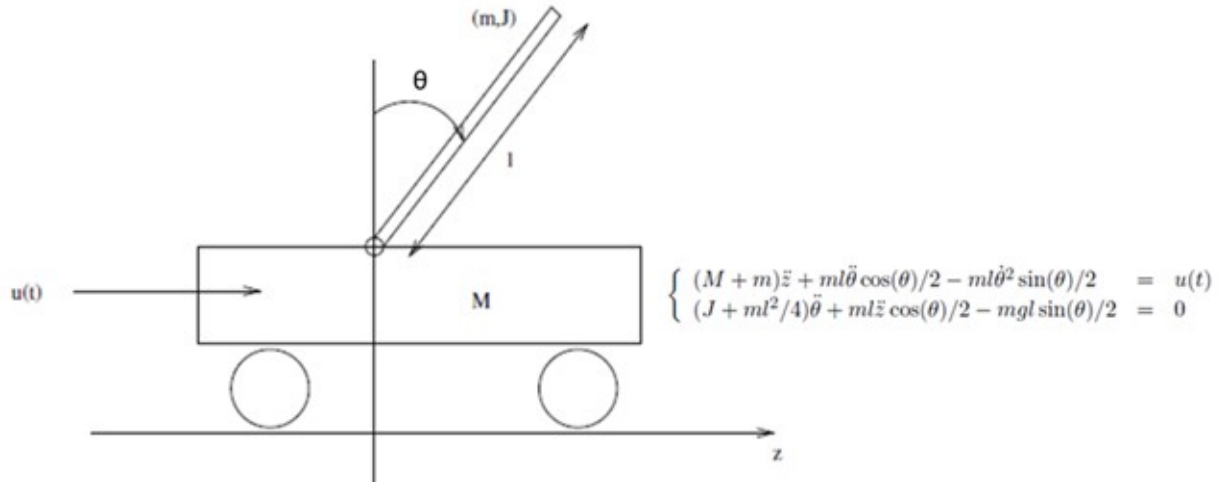
Figure 22.17: Dynamics of an inverted pendulum on a cart system.

22.5.3 Example: Inverted Pendulum on a Cart

Consider the classical problem of the inverted pendulum on a cart illustrated in Fig. 22.17. The system state contains the position and speed of the cart, and the angle and angular velocity of the pendulum. State feedback control is used to stabilize the pendulum in the vertical upright position. The model contains a delay block for studying the effect of control delay in the performance of the controller as shown in Fig. 22.18.
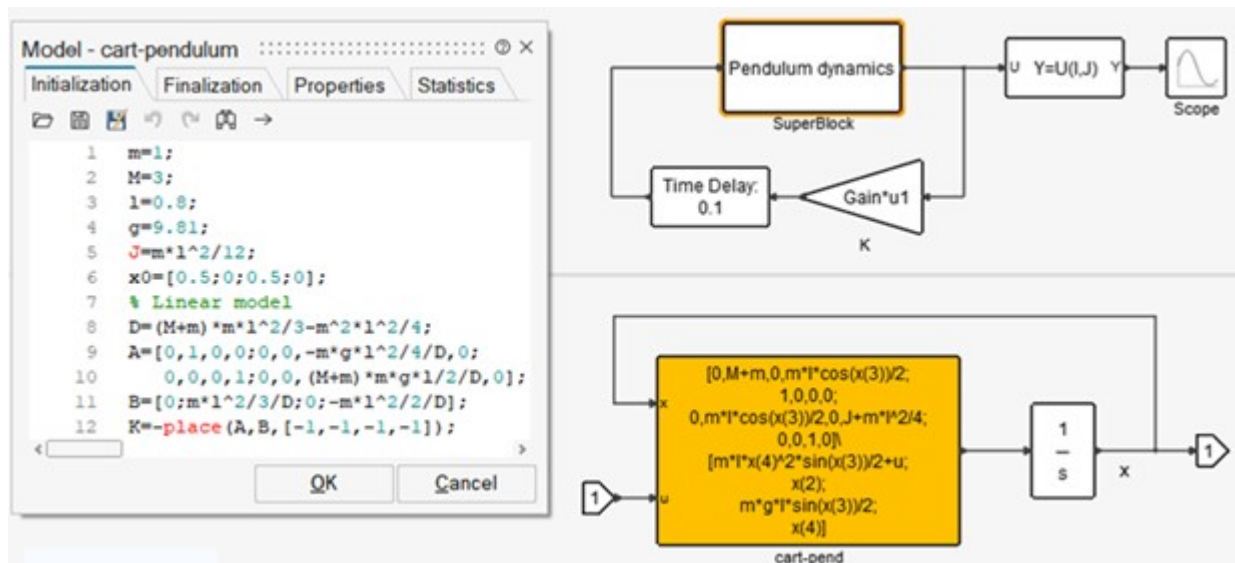

Figure 22.18: **Twin Activate** model of the controlled system.

The model of the pendulum-cart system is placed inside a Super Block and various code generation methods are applied to it to illustrate how the methods work and differ. The Super Block has one input (force applied to the cart) and one output (vector containing all four elements of the state). The parameters used inside, and defined outside, are the cart and pendulum masses, M, m, the pendulum length, l, its moment of inertia, J, the initial state x0 (x_0, the initial state of the **Integral** block) and the

gravity constant `g`. These parameters can be seen by auto-masking the Super Block:
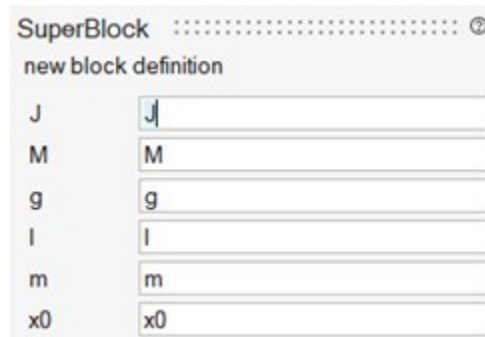
The Super Block can now be used like a regular block, with the mask as the parameter GUI. This GUI can be customized by editing the mask, for example as follows:
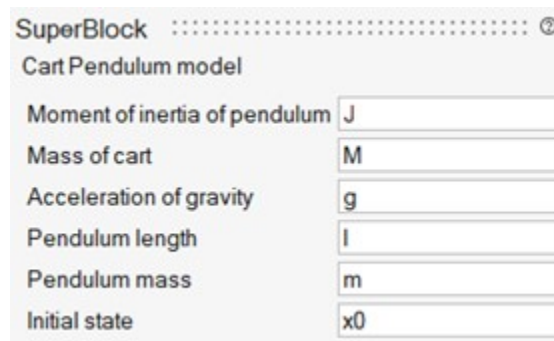
Code Generation   The application of code generation to this Super Block, as discussed earlier, exposes all the mask parameters because their values are defined as variables with the same names. The application of the code generators produces a **CCustomBlock** with parameters `x0`, `J`, `M`, `g`, `l` and `m`, as shown in Fig. 22.19. The **CCustomBlock** is placed automatically in a masked Super Block with
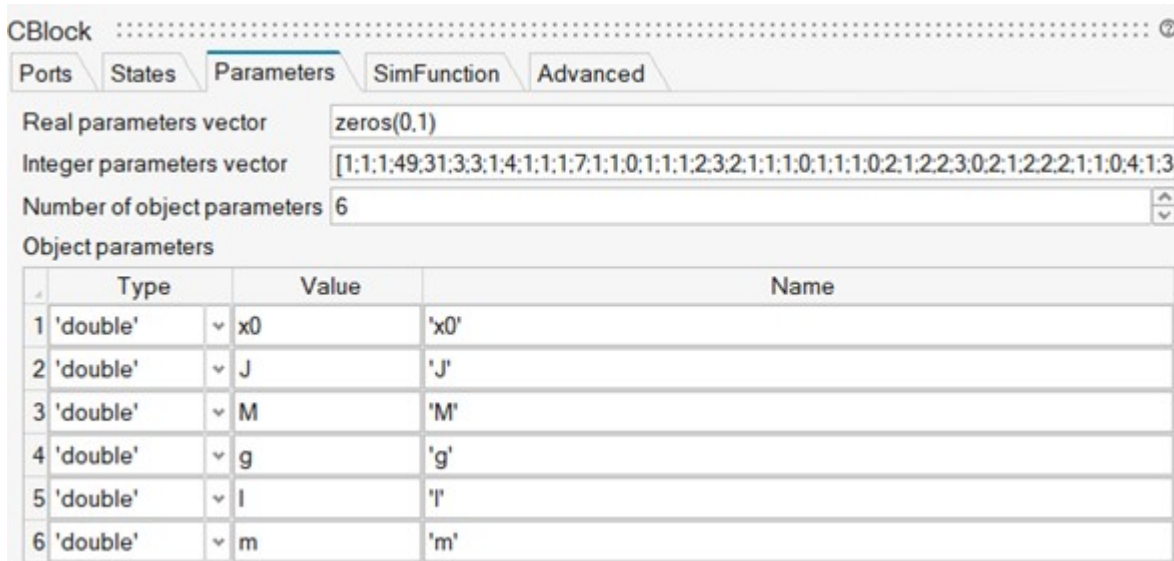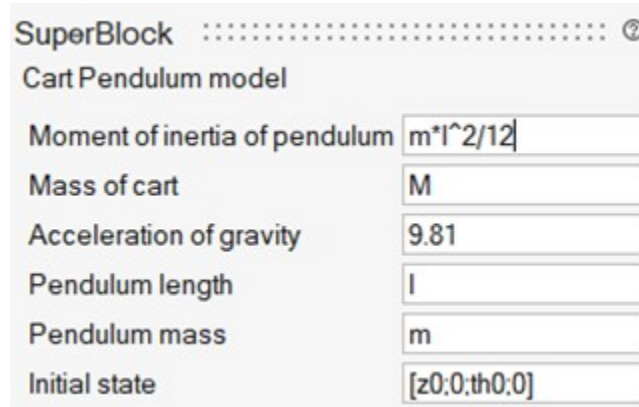

Figure 22.19: Exposed parameters become parameters of the generated **CCustomBlock**.

the same parameters.

Selection of Exposable Parameters   Parameters that you would want to expose are not necessarily the parameters obtained by masking the Super Block. In the cart-pendulum system, the more natural set of parameters to expose are `m`, `M`, `l` and the initial position of the cart and intial angle of the pendulum. The moment of inertia is a function of the pendulum mass and length ($J=ml2/12$) and `g=9.81`. The full initial state is also often not required; only the initial cart position and pendulum angle, `z0 = x0(1)` and `th0 = x0(3)`, are considered. To use these as exposed parameters, the mask parameter values of the original system can be modified as shown below:

SuperBlock
Cart Pendulum model

Moment of inertia of pendulum	m*l^2/12
Mass of cart	M
Acceleration of gravity	9.81
Pendulum length	l
Pendulum mass	m
Initial state	[z0;0;th0;0]

Note that the initial velocities are set to zero.

After defining initial values for `z0` (z_0) and `th0` (θ_0), for example in the Model Initialization script, the application of code generation, as expected, results in the following set of exposed parameters (the values of the exposed parameters here are manually replaced by numerical values):

CodeGenerationBlock
This block is generated by the code generator.

M	3
l	0.8
m	1
th0	0.5
z0	0.5
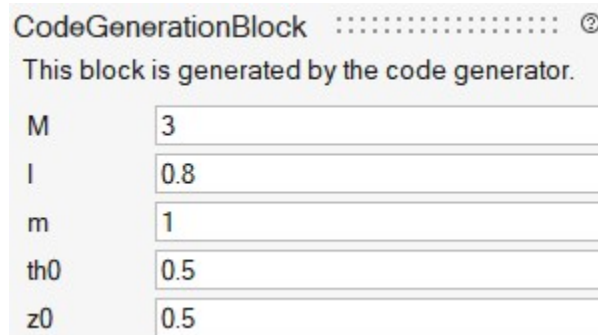
Similarly, if the Super Block is used to export an **FMU**, the exported **FMU** will have the same parameters as shown in Fig. 22.20.

22.5.4   Limitations with Exposable Parameters

Exposable parameters of a Super Block can be used directly in expressions defining the values of the parameters of the blocks inside the diagram of the Super Block, or be used in the Super Block Context to define other variables used as such. There are however limitations in the usage of exposable parameters.

Exposable parameters can be used in **OML** expressions involving basic operators (addition, subtraction, multiplication, etc.), matrix extraction and concatenation, and basic math and trigonometric functions.
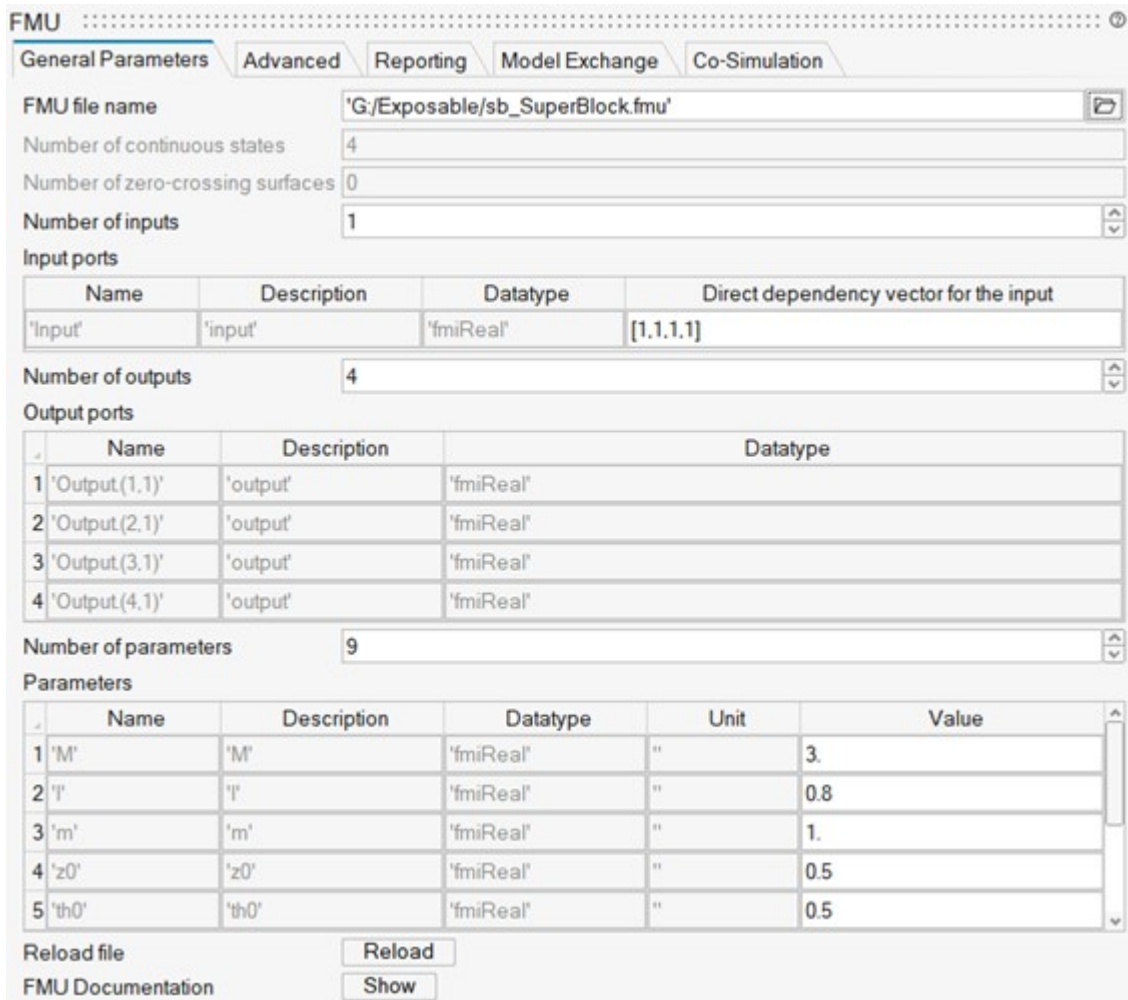
Figure 22.20: Exposed parameters become parameters of the exported **FMU**.

These functions are overloaded to accept exposable parameters. Since the implementation uses operator overloading, expressions including calls to functions including basic operators and overloaded functions are also supported.

However, exposable parameters cannot be used in non-overloaded built-in functions. For example, in the implementation of a Chebyshev filter block where the block parameters are computed using the **OML** built-in function cheby1, the parameters of the mask (filter parameters) cannot be exposed because cheby1 is not overloaded.

Even if an expression involving exposable parameters is valid, it cannot necessarily be used for the definition of all block parameter values. The reason is that not all block parameters in **Twin Activate** libraries accept expressions with exposable parameters. Structural block parameters are clearly excluded, and so are any parameters that affect the size or type of signals. Other block parameters, for the most part, accept expressions with exposable parameters.

22.5.5 Block Parameters supporting exposable parameters

The **Twin Activate** block parameters supporting expressions with exposable parameters are shown in Table 22.2.

22.5.6 OML operators and functions supporting exposable parameters

Exposable parameters can be used in the computation of the values of exposable block parameters. This computation can be limited to an **OML** expression or be based on the execution of one or more scripts included in Super Block Contexts. Even though there is no explicit limit on the complexity of the resulting expression, the usage should be limited to "simple" expressions; complex expressions may not result in efficient generated code and may even fail.

Exposable parameters, and other variables depending on exposable parameters, can be matrices of arbitrary sizes. Their usage is supported for the following operators and functions:

- Basic operators:

    + -* / \ .* .\ ./ ^ .^ == ~= ><>= <= & | '.':

- Matrix row and column concatenations, and matrix extraction (with access via ':' and 'end' supported)

- Primitive functions: sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh, exp, log, sqrt, min, max, transpose, floor, ceil, round, svd

- Primitive functions returning numeric values: size, length, isvector, isscalar, issquare. Since the sizes of exposable parameters are fixed and do not depend on their values, the output of these functions can be determined at compile time.

- Any **OML** function that uses only the above operators and functions within its definition, or other **OML** functions of the same type.

Suppose X and Y are exposable parameters, then the following expressions and instructions are valid

- `X + Y`
- `Z = X .* Y; sin(Z)^2`
- `X(1,:) * X(:,1)`
- `Z = [X, Y]; Z'`
- `if isscalar(X), Z=X; else Z=Y; end`

The last statement is valid because at compile time, it is known whether X is scalar or not based on its default value.

The following expressions and instructions are not valid

- `eig(X)`
- `eye(X,Y)`
- `if X > 0, Z = X; else Z = 0; end`

The last statement however can be expressed as an equivalent valid expression as follows
`Z = \max(0, X)`

Twin Activate block	Exposable Parameters
Accumulator	x0
Bias	B
CompareToConstant	C
ConditionalSelect	Thra
Constant	C
ContStateSpace	x0
ContTransFunc	Num, den
Counter	Minim, step
Deadzone	lower, upper
DFlipFlop	init
DLatch	init
DiscreteDelay	init_cond, maxim, Initv
DiscreteIntegral	x0
DiscrTransFunc	num, den
EventDelay	delay
EventGenerate	etimes.time1
FixedDelay	T
Gain	gain
Horner	coeffs
Integral	x0
JKFlipFlop	init
LookupTable	xx, yy
LookupTable2D	xx, yy, zz
LookupTableND	Ff, dimii
MatrixGain	gain
MathExpression	All parameters used in the expression
MatrixExpression	All parameters used in the expression
ModuloCounter	ini_state, base, step
Power	power
PID	Kp, Ki, Kd
Ramp	slope, startt, initout
RampSaturate	Height, duration, offset, startTime
Random	A, B
Saturation	upper, lower
Sawtooth	Period, up, down
SignalGenerator	xx, yy (only scalar double signals, without output derivatives)
SineWaveGenerator	M, F, P, offs
SquareWaveGenerator	FV, SV, period, dutyCycle, offset
SRFlipFlop	init
StepGenerator	steptime, iniv, finv
Trapezoid	Amplitude, rising, width, falling, period, startTime, offset

Table 22.2: Block parameters supporting exposable parameters.

22.5.7 Exposable Parameters and Modelica Blocks

The P code generator does not support **Modelica** blocks, but the code generator using block simulation functions does. In that case, the values of **Modelica** block parameters can be defined using expressions with exposable parameters, as in the case of non **Modelica** blocks.

The process of defining, manipulating, and exposing parameters is similar for Activate and **Modelica** blocks. Indeed, what is referred to as **Modelica** blocks, that is **Twin Activate** blocks representing **Modelica** components, are parameterized in the same way as "regular" **Twin Activate** blocks are. So, not only can Activate and **Modelica** blocks with exposable parameters be used in the same diagram, the same exposable parameters can be used in the expressions used to define the parameter values of both types of blocks.

Not all **Modelica** block parameters however support expressions with exposed parameters; the restrictions are similar to those for **Twin Activate** block parameters, mainly that the block parameter should not be structural.

22.6 Inlined (P) Code generater

Both the standard and P code generators can be applied to any Super Block but unlike the standard generator, which succeeds to generate code systematically, P code generation may fail because of limitations and restrictions associated with this generator. To understand these limitations, and the way the Super Block can be modified to avoid them, a description of the way the P code generator functions is presented in this section.

22.6.1 Basic idea

Most **Twin Activate** block simulation functions are not expressed in **OML**; they are hard coded in C for better performance during simulation. But Block operations correspond often to **OML** basic operations: blocks operate on vector and matrix signals with semantics very similar to corresponding **OML** operations, for example matrix addition, multiplication, concatenation, extraction... So, the block simulation functions, in most cases, can be expressed very simply in **OML** language. For example, the **MatrixMultiplication** block simulation code is complex, having to handle different cases where inputs are of matrix, vector or scalar types but it can be expressed as a simple **OML** product. This is the case because the **OML** multiplication operator supports the same type of inputs and provides the same functionality as the **Twin Activate** block.

Based on this observation, the basic idea behind the P code generator is to express block simulation functions in **OML** (referred to as the P block simulation functions), and generate an **OML** code associated with the Super Block for which code is generated calling the **OML** simulation functions of the blocks present in the diagram. This provides an **OML** program, which is used to generate C code for the Super Block.

In general, it is not possible to generate efficient C code from an **OML** program because **OML** is not a strongly typed language but the **OML** program resulting from the Super Block code generation is very constrained (only fixed size variables, no recursive calls, limited conditional statements...), and size and type information regarding many variables are provided by the compiler. Thanks to these restrictions and the available information, very efficient code can be generated for the **OML** program associated with the Super Block.

The C code generated by P is in general far more efficient than the code generated by the classical generator. Block library simulation functions are more or less generic, whereas P generates code for specific instance of blocks (in particular known signal sizes and parameter values). For example, the **MatrixGain** block supports different multiplication methods: left and right matrix products, element-wise product, Multiplication with a scalar. The simulation function of this block contains a generic multiplication code where the method and input sizes are tested at every simulation step but P generates code for the specific configuration, resulting in a specific code. The following example illustrates this point.

Simple example   Consider the generating code for the following diagram
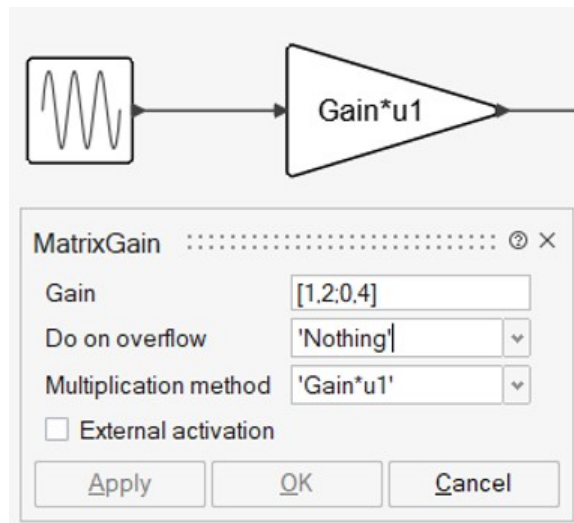
The **SineWaveGenerator** block produces a vector signal of size 2 with magnitude `[1;0]`. The **Matrix-Gain** block parameter is a 2×2 matrix, which is multiplied with the sine wave signal. The Standard code generator realizes this operation by calling the block simulation function:

```
VSS_EXPORT void gainblk_r(vss_block *block,int flag){
  double *u=GetRealInPortPtrs(block,1);
  int ru=GetInPortRows(block,1);
  int cu=GetInPortCols(block,1);
  double *y=GetRealOutPortPtrs(block,1);
  int ry=GetOutPortRows(block,1);
  int cy=GetOutPortCols(block,1);
  SCSREAL_COP *Gain=GetRealOparPtrs (block,1);
  int rGain=GetOparSize(block,1,1);
  int cGain=GetOparSize(block,1,2);
  SCSINT32_COP *Mulmet=Getint32OparPtrs (block,2);
  SCSREAL_COP *Over=GetRealOparPtrs (block,3);
  int rOver=GetOparSize(block,3,1);
  int cOver=GetOparSize(block,3,2);
  int i;
  if (rGain*cGain==1) {
    for (i=0;i<ry*cy;++i){
      y[i]=Gain[0]*u[i];
```

```
      }
      return;
   }
   if (ru*cu==1) {
      for (i=0;i<rGain*cGain;++i){
         y[i]=Gain[i]*u[0];
      }
      return;
   }
   switch(Mulmet[0])
      {
      case 0: /* K.*u */
         for (i=0;i<ry*cy;++i){
y[i]=Gain[i]*u[i];
         }
         break;
      case 1: /*  K*u  */
          dmmul(Gain,&rGain,u,&ru,y,&ry,&rGain,&cGain,&cu);
         break;
      case 2: /*  u*K  */
         dmmul(u,&ru,Gain,&rGain,y,&ry,&ru,&cu,&cGain);
         break;
      }
}
```

The P code generator on the other hand generates the following inlined code

```
/* MatrixGain: Gain block begins */
  io_1_Output[0] = (sin(sim_t[0])+((2)*(SineWaveGenerator[1])));
  io_1_Output[1] = ((4)*(SineWaveGenerator[1]));
/* MatrixGain: Gain block ends */
```

Note that the code is inlined: no function calls; it contains no run-time check for multiplication method and scalar type; the matrix multiplication code is inlined (no call to generic multiplication code is present because the matrix is smaller than a user definable threshold); and the multiplication by zero is optimized-out from the code.

22.6.2   Block coverage and other limitations

For a basic block to be supported by the P code generator, it must have a corresponding **OML** P function. P functions are already available for most basic blocks in the **Twin Activate** library but such functions need to be provided for third-party libraries, for P support.

Note that library blocks constructed as Super Blocks using blocks already supported by P are automatically supported by P. For example, the blocks in the ASM library are all of this type, so automatically supported by P. Many third-party libraries are of this nature.

Not all **Twin Activate** library blocks are supported by P. The list of supported **Twin Activate** library blocks is provided in Appendix A.1. Some blocks, such as the **Scope** or **PCustomBlock**, are not supported because they require the **Twin Activate** environment (even the classical generator does not support them for targets other tha **Twin Activate**), others are not supported because their operation is

based on **Twin Activate** features not supported by P.

Indeed, P does not support all **Twin Activate** features. This code generator had been originally designed for periodic discrete-time systems. It had been extended to support general external activations, and later to support limited continuous-time dynamics mainly for use with fixed-step solvers (so no support for zero-crossings or mode changes).

So, no blocks generating events (other than **IfThenElse**, **SwitchCase**, **IfExpression** and **EdgetTrigger** blocks which subsample existing events) or blocks requiring DAE solvers, are supported.

22.6.3 PCustomBlock

The **PCustomBlock** can be used to define a custom block to be used both for simulation and P code generation. The main parameter of this block is an **OML** code corresponding to the P function of the block. This code is interpreted during simulation (low performance but easy to debug) and is used for code generation by P to generate highly efficient code.

The P function of the **Twin Activate** library blocks are examples of such **OML** codes. The following is the P function of the **MatrixGain** block:

```
function block=P_System_MatrixGain(block,flag)
  if flag==1
    setoverflow(block.params.overflow);
    gain=convert(block.params.gain,datatype(block.io{1}));
    if strcmp(block.params.mulmethod,'Gain*u1')
      block.io{2}=gain*block.io{1};
    elseif strcmp(block.params.mulmethod,'u1*Gain')
      block.io{2}=block.io{1}*gain;
    elseif strcmp(block.params.mulmethod,'Gain.*u1')
      block.io{2}=gain.*block.io{1};
    else
      error(['Unknown Multiplication Method: ',block.params.mulmethod]);
    end
  end
end
```

Note that this code also handles integer datatypes. The parameter overflow indicates how the overflow should be treated (saturation, modulo computation or error generation). Comparing this code with the simulation C code of the library block (given in the previous section for double datatypes `gainblk_r`) shows the simplicity of the P implementation of the block.

PCustomBlockOML function

The **OML** P function defining a **PCustomBlock** has the following calling sequence

```
block=PBlockFunction(block,flag)
```

where

- `block` is a structure containing
  - `block.io`: Inputs/outputs. A cell containing inputs followed by outputs.
  - `block.params`: Parameters.A structure containing the block parameters.

- – `block.state`: Discrete states. A cell containing block discrete states

- – `block.nevprt`: Activation code. Integer indicating how block is activated (same as for simulation functions)

- – `block.x`: Continuous-time state. A cell containing vector of block state and its derivative

- – `block.t`: Current time.

- • `flag` indicates the task to perform: 1 (output update), 2 (state update), 0 (state derivative), as for simulation functions, and -1, used for instantiation (for example to define and initialize block discrete states).

Simple example

The **PCustomBlock** can be used to interactively define a P block, inside a **Twin Activate** diagram. Wide classe of blocks can be realized, including blocks supporting integer datatypes.

The example considered here is a block computing the norm of a vector v, as $v'Pv$. The block may have 1 or 2 inputs: the first input is thea vector v of type double. The second input, which is optional, is the matrix P also of type double. The block has an optional parameter P, present if block has only one input. The output is $v'Pv$.

Both configurations of the block can be realized with a unique P function:

Note that the first block input, `io{1}`, is v, and P is either the second input, if present, or a parameter. The output is `io{2}` or `io{3}` depending on the number of inputs.

This code can be used in **PCustomBlock** with both one and two inputs as shown in the diagram in Fig. .

The generated code by P corresponding to two blocks `P-norm_1` and `P-norm_2` in the diagram, are as follows: for `P-norm_1` and parameter `P = [2,1;1,2]`, the generated code is

```
static void updateOutput (double *io_1_Input, double *io_2_Output)
{
```
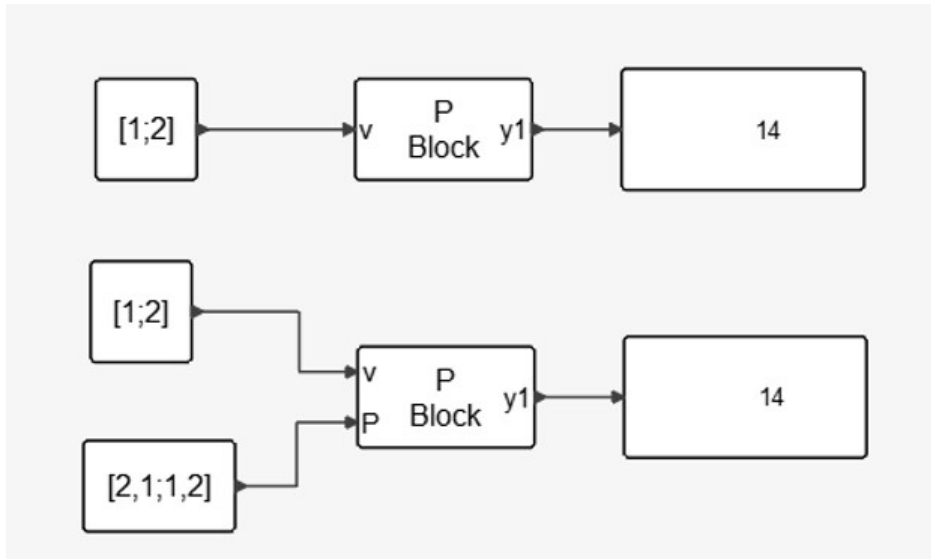
Figure 22.21: The same P code used in two **PCustomBlock** blocks with different configurations.

```
  double PCustomBlock [2U] = { 0.0 };
  PCustomBlock_3[0] = (io_1_Input[0]);
  PCustomBlock_3[1] = (io_1_Input[1]);
  io_2_Output[0] =
  (((((PCustomBlock [0])*(2))+(PCustomBlock [1]))*(io_1_Input[0]))+
      (((PCustomBlock [0])+((PCustomBlock [1])*(2)))*(io_1_Input[1])));
}
```

For `P-norm_2`, the generated code is

```
static void updateOutput
  (double *io_1_Input, double *io_2_Input_1, double *io_3_Output)
{
  double PCustomBlock_3[2U] = { 0.0 };
  PCustomBlock [0] = (io_1_Input[0]);
  PCustomBlock [1] = (io_1_Input[1]);
  io_3_Output[0] =
  (((((PCustomBlock [0])*(io_2_Input_1[0]))+((PCustomBlock [1])*
  (io_2_Input_1[1])))*(io_1_Input[0]))+ ((((PCustomBlock [0])*
  (io_2_Input_1[2]))+((PCustomBlock[1])*(io_2_Input_1[3])))*
  (io_1_Input[1])));
}
```

Example: inverted pendulum on a cart

The diagram in Fig. 22.22 models the inverted pendulum cart system (plant) controlled with an observer-based controller. The plant is realized using a **PCustomBlock**, implementing the equations of the system as shown in Fig. 22.23.

415 Extended Definitions   Altair Twin Activate 2023

$$\dot{x} = f(x, u) = F^{-1}v$$

$$F = \begin{pmatrix} 0 & M+m & 0 & ml\cos(x_3)/2 \\ 1 & 0 & 0 & 0 \\ 0 & ml\cos(x_3)/2 & 0 & J+ml^2/4 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad v = \begin{pmatrix} mlx_4^2\sin(x_3)/2 + u \\ x_2 \\ mgl\sin(x_3)/2 \\ x_4 \end{pmatrix}$$
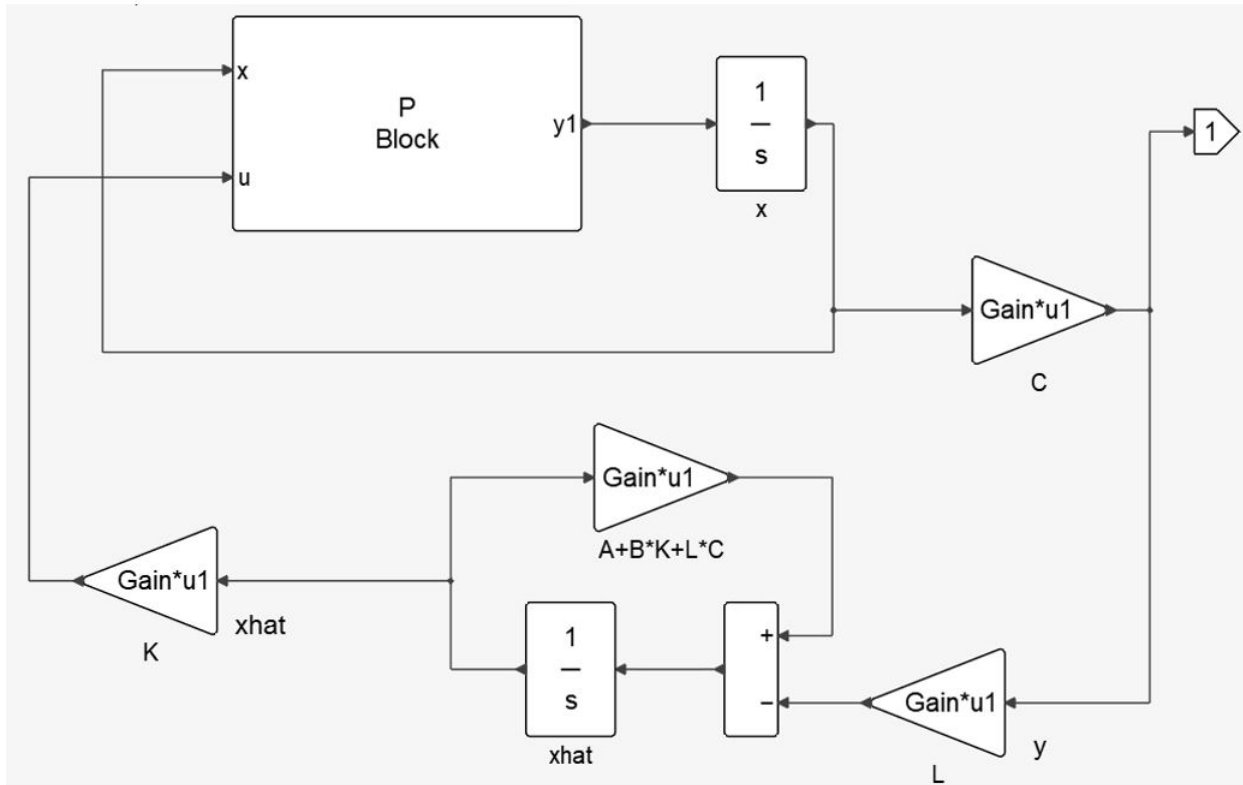

Figure 22.22: In this model, the plant is realized using a **PCustomBlock**.

OML P code limitations

There are limitations on the **OML** programs that can be used for defining P code. These **OML** programs are converted to a subset of C without features such as dynamic memory allocation, and recursion, thus limiting their coverage of the **OML** language.

There are three main limitations on **OML** programs that can be used for defining P code: the use of variables changing size, some conditional statements and the use of some **OML** functions.

Variables must have known sizes   Programs including operations creating variables where the size of the variable depends on a function (block) input value, and thus cannot be determined at compile time, are not supported. For example, `y=ones(u)` , `y= [1:u]` and `y=find(0==u)` are not in general supported because in these examples, the size of `y` depends on the value of `u`. The operation is supported only if `u` is known at compile time, i.e., if it is constant, or if its value can be determined because of constant propagation (parameter or expression involving constants and parameters).

Figure 22.23: The P code of the **PCustomBlock**.

Note that **Twin Activate** blocks cannot be created for these operations anyway since signal sizes (block input output port sizes) cannot vary during simulation.

Conditional statements   In **OML**, the two branches of an if-statement may contain very different codes. This is supported by P if the condition is known at compile time. For example: the P block code in Fig. 22.24 is valid.

Here all variable sizes and datatypes are known, so, n can be determined at compile time, and the generated C code only includes one of the branches. For example in case v has size 2, the generated C code is

```
static void updateOutput (double *io_1_Input, double *io_2_Output)
{
    io_2_Output[0]=(sqrt(((((io_1_Input[0])*(io_1_Input[0]))+
      ((io_1_Input[1])*(io_1_Input[1])))))/(1));
}
```

If the condition of an if or switch expression depends on a signal value, then the expression is not supported. But in some cases, these expressions can be replaced by expressional if and switch functions, which are supported by P. For example:

```
if abs(u)<1e-12, y = 1; else y = sin(u)/u; end
```

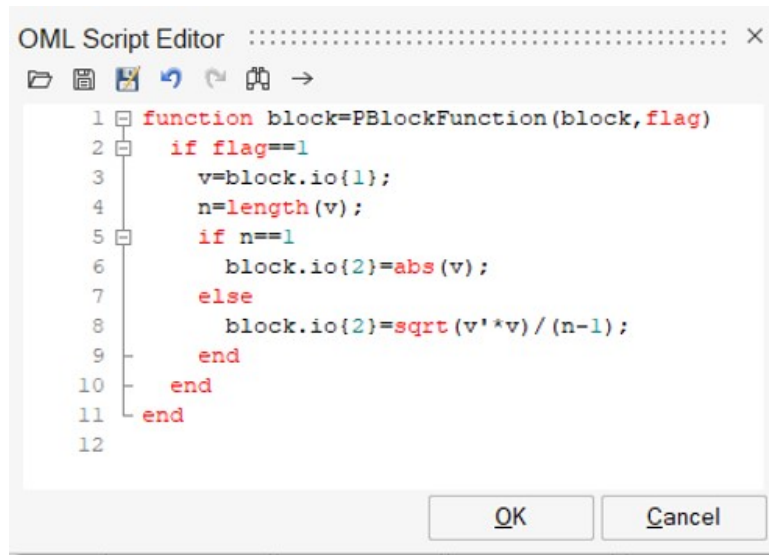is not supported. But it can be expressed as follows

417 Extended Definitions   Altair Twin Activate 2023

Figure 22.24: Valid P code because the condition is known at compile time.

```
y = If_exp(abs(u)<1e-12, 1, sin(u)/u);
```

which is supported:

The following code is generated in this example

```
static void updateOutput_20220312_1
        (double *io_1_Input, double *io_2_Output)
{
  io_2_Output[0] = ((((io_1_Input[0]>(0)) ?
    io_1_Input[0] : (-(io_1_Input[0])))<(1e-12)) ?
    (1) :  (sin(io_1_Input[0])/io_1_Input[0]));
}
```

Note that the conditional operations in **Twin Activate** diagrams are natively realized by two specific blocks (language constructs): **IfThenElse** and **SwitchCase**. The previous example can be implemented using the **IfThenElse** block as shown in Fig. 22.25.

Unsupported OML functions   An **OML** P code may use all functions, primitives and operators, overloaded for P usage. Basic **OML** functions, primitives and operators such as sin, cos, exp, inv, diag, *, +, .*, /, \, matrix concatenation, extraction, are overloaded. **OML** P code may also use other **OML** functions, even not overloaded, if their arguments and operands are constants or propagated constants
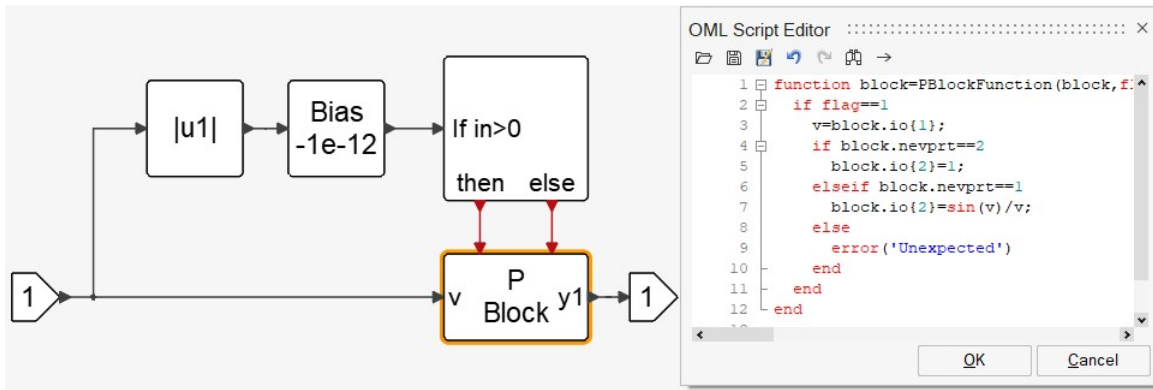
Figure 22.25: P code has access to block `nevprt`.

(parameters or expressions depending on parameters).

For example, if `u` is an input

```
num=2; den=[1,2]; [a,b,c,d]=tf2ss(num,den);
```

is supported but

```
num=2; den=u; [a,b,c,d]=tf2ss(num,den);
```

is not because the function tf2ss is not overloaded for P.

Code optimization by constant propagation

In **Twin Activate**, parameter evaluations and signal computations are done very differently. Since parameters are constants, parameter evaluation is done in **OML** before simulation and code generation, once: simulation and the generated-code do not use the **OML** interpreter.

Twin Activate blocks are developed in such a way that computations are done during evaluation phase, as much as possible, reducing runtime computations. But often there are trade-offs to avoid complexity and the need for a large number of simulation functions. For example, the performance of the **Matrix-Gain** block can be slightly improved by using different simulation functions for different multiplication methods but the number of simulation functions are then significantly increased.

In the P code generation approach, this type of optimization is obtained automatically and naturally thanks to constant propagation. Not only code optimization is realized by propagating the block parameter values but also constant signals.

Consider the construction of a block implementing a digital lowpass filter. This can be done using the **OML** function butter to construct the filter and the function tf2ss to obtain its `(A,B,C,D)` state-space representation, which is used during simulation as follows:

$$
\begin{aligned}
x(k+1) &= Ax(k) + Bu(k) \\
y(k) &= Cx(k) + Du(k).
\end{aligned}
$$

The block parameters are

- the sampling frequency f_s,
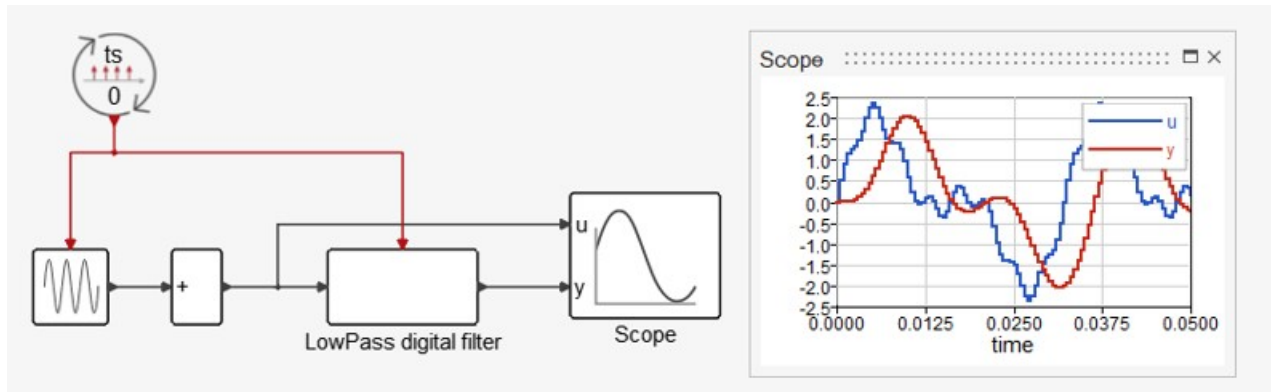- the cut-off frequency f_c,

- the filter order n.

Figure 22.26: The low pass filter is a new block to be implemented.

The model in Fig. 22.26 shows a model using such a block. It actually implements the **OML** filter function documentation example with $n = 4$, $f_s = 2000$, $f_c = 100$ and

$$u(t) = \sqrt{(2)}\sin(0.5\omega_1 t) + \sin(\omega_1 t) + 0.25\sin(\omega_2 t).$$

The usual way to construct such a filter in **Twin Activate** would be to compute `(A,B,C,D)` in the block **OML** interface function or a diagram context and use these matrices as parameters to implement the dynamics of the linear filter in a C simulation code (in case of a new block) or as parameters of the **DiscrStateSpace** block. This separates the computation of the filter coefficients, which is done in the evaluation phase from the code of the dynamics of the filter, which is used during simulation.

If this filter is implemented as a P Block (for example using the **PCustomBlock**), separating the codes for parameter evaluation and run-time execution is not required: thanks to constant propagation, the generated code includes the result of the parameter evaluations.

Note that the **OML** P code is in general inefficient for simulation, that is why library blocks are not built as P Blocks, but the generated code, is very efficient. So, the **PCustomBlock** block should be considered only when the objective is to use P code generation.

To emphasize the contrast between the simulation performance and the efficiency of the generated code, consider the "lazy" programming approach for implementing the P code, shown in Fig. 22.27. The `(A,B,C,D)` matrices are recomputed at every step, slowing down significantly the simulation.

However, thanks to constant propagation, the generated code uses the result of the parameter evaluation as seen below:

```
static double z_1[4U] = { 0.0, 0.0, 0.0, 0.0 };
static void updateOutput (double *io_1_u, double *io_2_y) {
io_2_y[0] =
((((((0.002991448307)*(z_1[0]))+((0.0008910247326)*(z_1[1])))+
   ((0.002546319058)*(z_1[2])))+((0.0002340182948)*(z_1[3])))+
   ((0.0004165992044)*io_1_u[0]));
}
static void updateState (double *io_1_u, double *io_2_y) {
  double PCustomBlock_111[4U] = { 0.0 };
  PCustomBlock_111[0] =
```

Figure 22.27: A lazy implementation of the P code for the low pass filter.

```
       ((((((3.180638549)*(z_1[0]))+((-3.861194349)*(z_1[1])))+
         ((2.112155355)*(z_1[2])))+((-0.4382651423)*(z_1[3])))+
         io_1_u[0]);
  PCustomBlock_111[1] = ((z_1[0]));
  PCustomBlock_111[2] = ((z_1[1]));
  PCustomBlock_111[3] = ((z_1[2]));
  memcpy (z_1, PCustomBlock_111, 4 * sizeof (double));
}
```
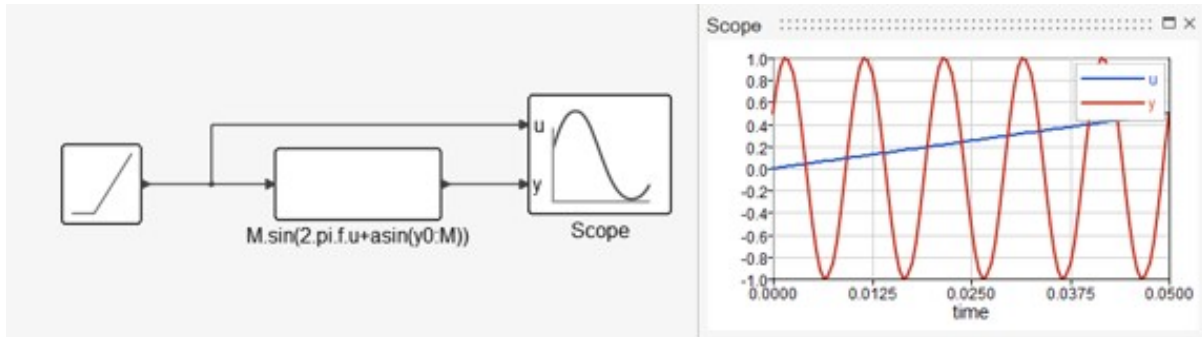
Constant signal propagation Constant propagation operates also at diagram level, for constant signals. Consider the following diagram of a Super Block
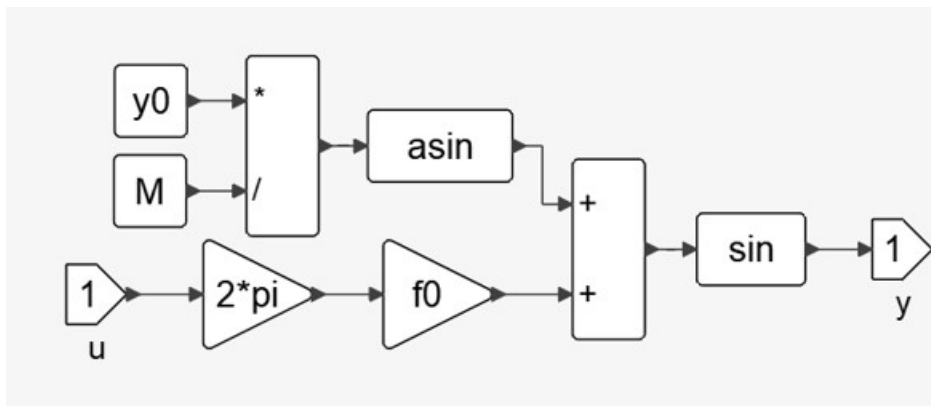
realizing the function

$$y(t) = M \sin(2\pi f_0 u(t) + \arcsin(y_0/M)).$$

If this Super Block is used in the following model

The simulation (as well as standard code generation) computes only once the expression `asin(y0/M)` but two function calls are made at every step to multiply the input by 2π and `f_0`, in addition to calls to the **Sum** and **TrigFunc** blocks.

Generating C code using P for this Super Block results in a single instruction to compute the output:

```
static void updateOutput (double *io_1_u, double *io_2_y)
{
   io_2_y[0] = sin(((0.5235987756)+((10)*((6.283185307)*io_1_u[0])))));
}
```

There is no initialization code: all the constant values are computed at code generation time.

Note that if full inlining of numerical values is not desired, it is possible to leave some parameters in the generated code (parameterize the generated code) by defining them as exposable.

22.6.4 Embedded solver option

Solver inlining transformation is a method for embedding a numerical solver within the generated code. This transformation can be applied to a general **Twin Activate** 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.

With solver inlining, models with Modelica components and co-simulation FMUs are supported by the P code generation.

A simple way to impose solver inlining is by making the choice in the code generation GUI as shown in Figs. 22.28 and 22.29. Solver inlining can also be specified within the model. This latter method has advantages over making the choice in the GUI:

- When the solver inlining choice is made in the model, the simulation of the model uses the embedded solver as well. So, the impact of solver inlining and its solver properties can be assessed at the simulation level, before code generation.

- Solver properties can be set for each super block individually. This provides finer control over the generated code, in particular for the management of Atomic super blocks.

- The inlining choices and their corresponding solver choices, are part of the model, making the model self contained. When the choices are made in the code generation GUI, they are lost when
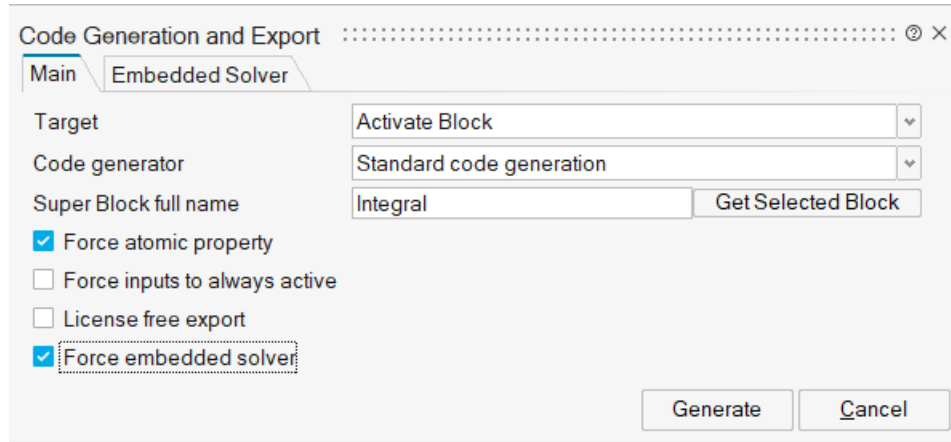
Figure 22.28: The choice of Embedded solver (solver inlining) is made using the last checkbox in the code generation GUI.
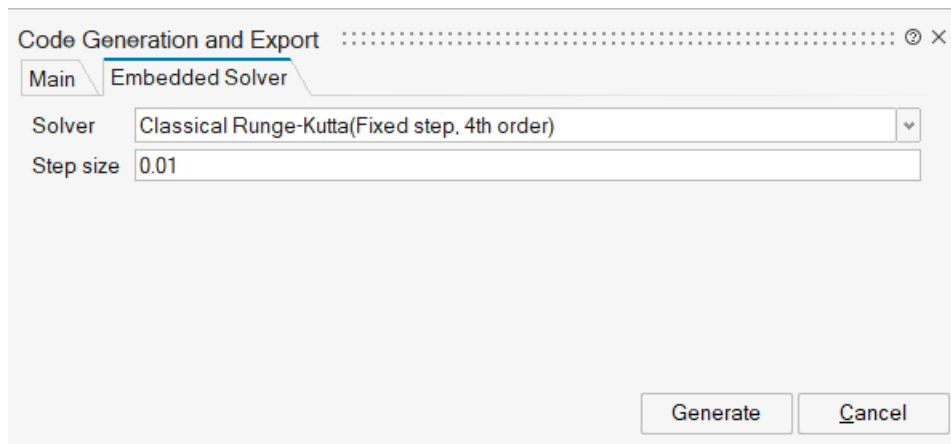

Figure 22.29: Solver properties are defined in the second tab.

the GUI is closed.

A description of solver inlining process and how the choice can be made in model are provided in the rest of this section.

Solver inlining

The mode 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, in addition to redefining the behavior of some of the **Twin Activate** blocks accordingly.

The objective of this transformation is to remove the dependency of the model on external numerical solvers, both for simulation in **Twin Activate** and model export using code generation for different targets. The exported model can be considered both as a discrete-time block and a Co-simulation component.

Even though the solver inlining can be used both by P and the classical code generators, its main usage

is for model exported by the P code generator for embedded applications.

Note that by applying solver inlining, the block coverage in the P code generator is significantly improved. see Section 22.6.4.

Model transformation Process

The model transformation process is completely transparent to the user; it is performed during the model compilation phase. There are two distinct operations performed during this transformation:

1. Insertion, at the top of the transformed model, of a set of blocks implementing the iterations required to run the numerical solver algorithm.

2. Replacement of some blocks with alternative realizations specifically designed for usage with embedded solvers. The replaced blocks are not limited to those including continuous-time states but concern many blocks, in particular the blocks with dynamics depending on time and those generating events.

Basic idea

If the embedded solver is the first order explicit Euler solver, the model transformation is straightforward. The differential equations associated with the dynamics of blocks with internal continuous-time states can be replaced by time difference equations, which in turn can be implemented by discrete-time blocks running on a base clock. This however does not work for high order 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 **Twin Activate** by two blocks: an integrator block and a memoryless block realizing the function f.

$$\dot{y} = f(y)$$

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

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

where

$$y_k = y(t_k), \ t_{k+1} = t_k + h.$$

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 (**Twin Activate** model) by separating the system into an integrator block and a memoryless block by noting that it can be rewritten as follows

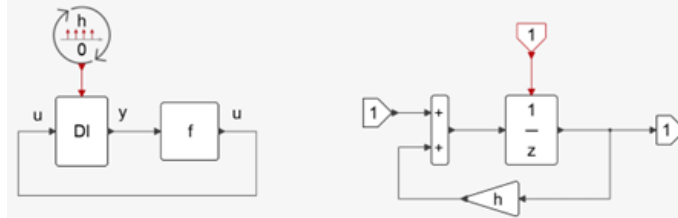$$\begin{aligned} \dot{y} &= u \\ u &= f(y) \end{aligned}$$

The corresponding **Twin Activate** model can be constructed as follows
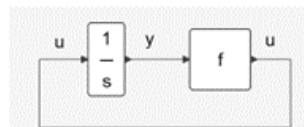
In this case, the Euler discretization yields

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

So, the discrete-time version of the **Twin Activate** model is obtained by simply replacing the integrator block by a discrete block as shown below

The content of the DI block is shown on the right. This new **Twin Activate** model is activated by a `SampleClock` block with period h; it is a purely discrete-time model. The original model is simply transformed by replacing the integrator with the DI block and a `SampleClock` block is added. In a more general model with multiple integrator blocks, each integrator block can be replaced by its discrete-time equivalent DI, activated the `SampleClock` block. The stateless blocks of the model, 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:

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

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 rewritten as follows

$$A0 : \begin{cases} y_k^0 = y_k \\ u_k^0 = f(y_k^0) \end{cases}$$

$$A1 : \begin{cases} y_k^1 = y_k + \frac{h}{2}.u_k^0 \\ u_k^1 = f(y_k^1) \end{cases}$$

$$A2 : \begin{cases} y_k^2 = y_k + \frac{h}{2}.u_k^1 \\ u_k^0 = f(y_k^2) \end{cases}$$

$$A3 : \begin{cases} y_k^3 = y_k + h.u_k^2 \\ u_k^3 = f(y_k^3) \\ y_{k+1} = \frac{1}{6}(2y_k^1 + 4y_k^2 + 2y_k^3 - 2y_k + h.u_k^3) \end{cases}$$

The computation over one time step is divided into 4 separate computations associated with actions A0 through A3. Each action can be realized by a discrete activation. In this formulation, a state is associated with each action; these states are the arguments of the function f. This system can be implemented as a **Twin Activate** model as follows
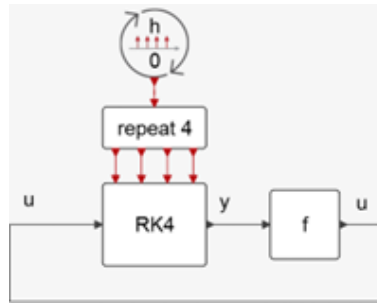
The `repeat4` block generates four activation events when it is activated. The first event is synchronous with the clock event that activates it. The subsequent ones follow without the advancement of time. The actual implementation uses the `RepeatActivation` block which forces the reactivation of the event which has activated it.

The (super) block RK4 realizes the four different actions depending on the way through which it is activated. The content of the RK4 is more complex than that of DI; the following diagram gives an idea of the complexity of its implementation:
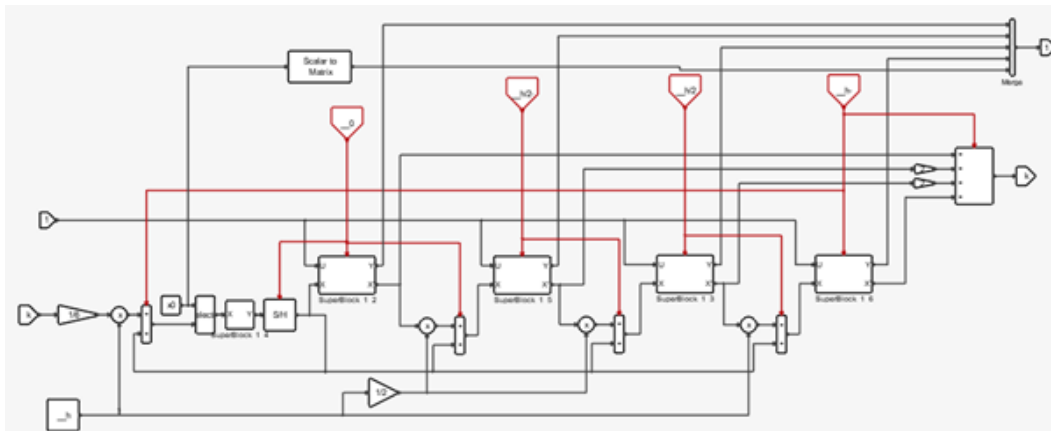
The embedding of the solver can again be implemented by replacing the integrator blocks with discrete-time blocks (in this case RK4 as opposed to DI) and adding blocks to drive them (in this case a `repeat4` block in addition to the `SampleClock` block). So, the transformation process is very similar.

This model transformation process can be used to transform continuous-time **Twin Activate** models made of integrator blocks and memoryless function blocks. **Twin Activate** models however often contain other types of blocks, in particular event generating blocks, such as clocks and zero-crossings, blocks activating discrete-time dynamics, blocks with internal continuous-time states other than integrator blocks and signal source blocks.

Support for general Twin Activate models   The model transformation discussed in the previous section, which consists of replacing integrator blocks and adding the event generation mechanism, may not create correct models if the model contains blocks generating and consuming events, blocks using explicitly the variable time, and blocks with internal continuous-time states other than integrator blocks.

Some of these blocks should also be replaced.

The replacement block is in general a super block realizing an alternative implementation of the original block containing supported blocks. In the following, the overall idea of the procedure is presented for different classes of blocks, which may require replacement. The description is made via examples without providing details.

Time dependent blocks   Some of **Twin Activate** blocks contain explicit dependence on time, as it is allowed in **Twin Activate**. This means that the dynamics of a **Twin Activate** model may explicitly depend on time. In that case, for example the simple continuous-time model considered in the previous section will have explicit time dependence as follows

$$\dot{y} = f(t, y)$$

In fact, the Euler and RK4 equations are in general presented for such a system. But the explicit treatment of the time variable can be avoided by including t in the state vector with the corresponding differential equation

$$\dot{t} = 1$$

This method is what is used for replacing blocks with time dependency. For example, the block Time is replaced by the block P_time:
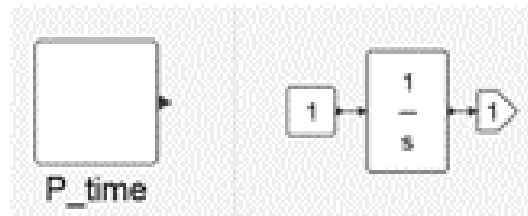
The content of P_time (super)block is shown on the right. Its output is obtained by integrating the constant 1. The integrator block is automatically replaced subsequently by its discrete-time version during the transformation process. Note that the time output will have the following values for the different actions A0, A1, A2 and A3:

$$
\begin{aligned}
A0: \quad t_k^0 &= t_k \\
A1: \quad t_k^1 &= t_k + \frac{h}{2} \\
A2: \quad t_k^2 &= t_k + \frac{h}{2} \\
A3: \quad t_k^3 &= t_k + h
\end{aligned}
$$

This implementation of time used for modeling the time dependent differential equation above yields a discrete-time system, equivalent to the RK4 algorithm for time dependent systems:

$$
\begin{aligned}
y_{k+1} &= y_k + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\
k_1 &= h.f(t_k, y_k) \\
k_2 &= h.f(t_k + \frac{h}{2}, y_k + \frac{k_1}{2}) \\
k_3 &= h.f(t_k + \frac{h}{2}, y_k + \frac{k_2}{2}) \\
k_4 &= h.f(t_k + h, y_k + k_3)
\end{aligned}
$$

427 Extended Definitions   Altair Twin Activate 2023

Activation signals If the model contains discrete-time dynamics, so events, then block replacements may be necessary to synchronize all the activations. The objective of the transformation is to obtain a synchronous model running on a single clock with period h. So, all the activations in the model must be obtained by conditional subsampling of the activation signal produced by this unique `SampleClock` block.

Several blocks in **Twin Activate** generate events by detecting the zero-crossings of continuous-time signals. In the original model using a variable-step solver for simulation, the event time is detected by pinpointing (through iterations) the exact time of the zero-crossing. The resulting event is not synchronized with any other event. In the embedded solver implementation, the solver step is fixed, and all actions should occur at sampling time instants. This means that they must be synchronous with the base clock.

To detect the zero-crossing sampling times, the zero-crossing block must be replaced by a discrete-time block which compares its input signal value and its previous value to decide if a zero-crossing has occurred.

Other sources of asynchronous events are event delay blocks. The activation signals generated by these blocks are not synchronous with any other event in the original diagram. In transformed models, they have to be synchronized with the base clock of period h. For this, the delay is expressed as a multiple of h and implemented by subsampling the base clock using a modulo counter as shown below
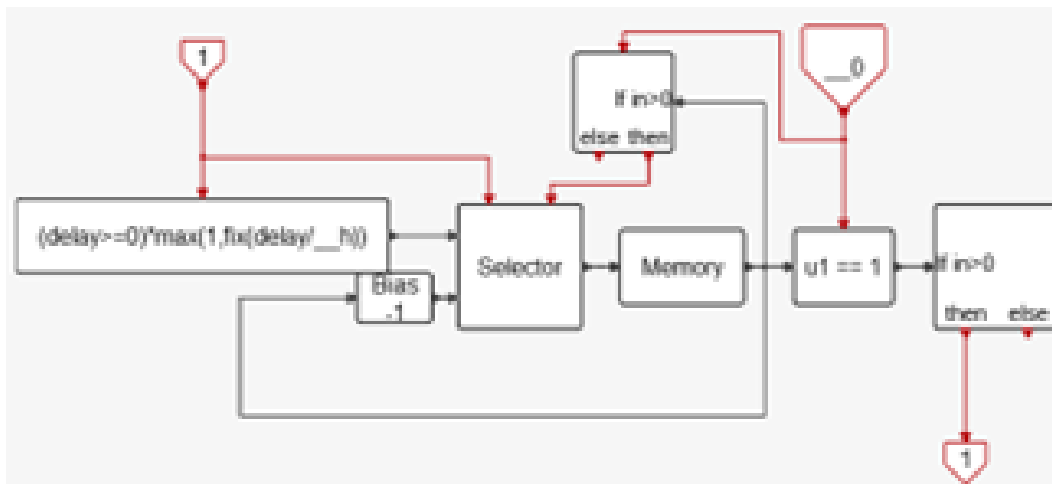

This block is the replacement for the fixed-time event delay block. The replacements of variable-time delay blocks are similarly implemented.

Note that the block replacement process, which can be seen as a function overloading operation in textual programming languages, automatically provides support for many blocks. In particular, if a library block is a super block (programmable or not), and the blocks inside the super block are already supported (replaced/overloaded if needed), there is no need to provide a replacement for it.

Other blocks with continuous-time states The replacement for the integrator block was discussed previously. The replacement of other blocks with internal continuous-time states are done using the integrator block. This is done in particular by replacing such blocks with super blocks realizing their dynamics using integrator blocks. Some of these blocks could have been already implemented in that way at the origin but they had been realized differently for minor performance gains.

For example, the continuous-time state-space block is replaced with the following (super)block
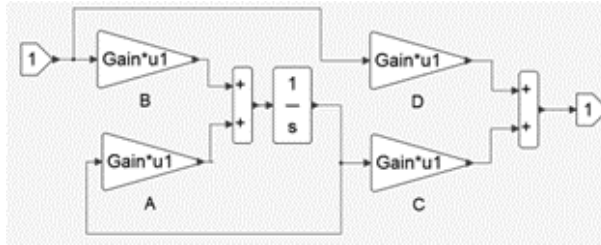
Since the integrator block is already supported (and so are the other blocks because they are stateless and not explicitly activated), this replacement provides support for embedded solver for the state-space block.

Making the solver inlining choice in the model

To designate one or more super blocks in the model for solver inlinling, The OML `vssSetInlineSolver` function must be used in the context of the super block for which solver inlining is performed. This function also allows setting the solver parameters. The Context of the super block may contain for example:

```
opt=struct;
opt.h=.5;
opt.solver='RK2';
vssSetInlineSolver(opt);
```

Solver inlining can be done for different super blocks in the same model, with possibly different solver choicies and step-sizes . However, if solver-embedded super blocks are nested, the step-size associated with the inner super block should be equal to, or be a multiple of that of the outer super block.

When the model is simulated, the designated super blocks run on their embedded solvers. By commenting out the call to vssSetInlineSolver instructions in diagram contexts, the original model is restored, facilitating the comparisons for evaluating the effect of inlining.

Both the classical and P Code cogenerators can be used with solver-embedded super blocks. In most cases, solver inlining is done when the P code generator is used.

Support for Modelica and co-simulation FMUs

Super blocks containing Modelica components are not supported by the P code generator. But they can be when the embedded solver is used.

For Modelica to be supported by Embedded solver, the `MaplesimCompilerSetting` block must be placed in the model to set the option co-simulation to on, as shown in Fig. 22.30.

The way Modelica is supported in **Twin Activate** is by regrouping the Modelica components in the model and transforming them into an FMU. General FMUs however are not supported by the embedded solver; only Co-Simulation FMUs are. By selecting the co-simulation option in the `MaplesimCompilerSetting` block, the Modelica part is transformed into a co-simulation FMU, which is then supported by the embedded solver.

The `MaplesimCompilerSetting` block must be placed in the super block for which solver inlining is performed (and P code is generated). Note that this block is not required if the choice to use Embedded solver is made in the code generation GUI. In that case, the co-simulation option is enforced antomatically and the solver for the Modelica part uses the same properties as that of the Embedded solver.
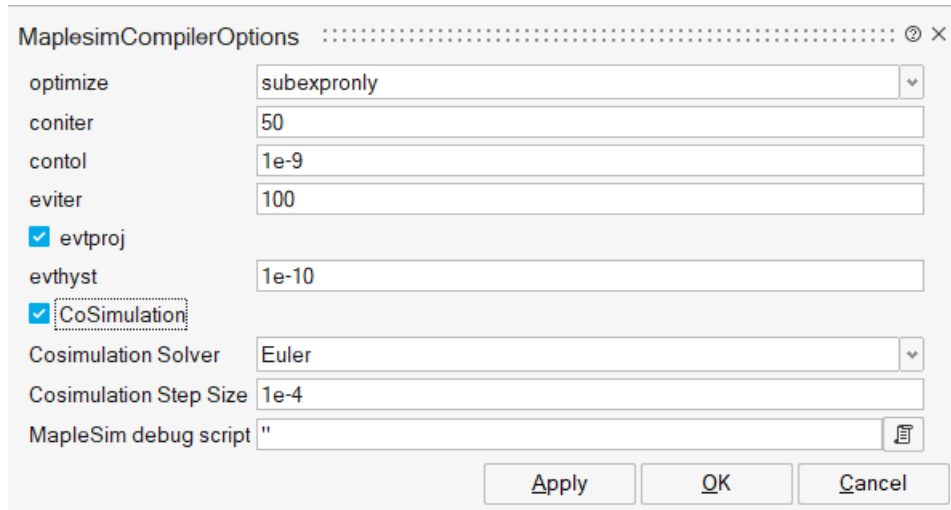
Figure 22.30: By checking the box CoSimulation, The Modelica part of the model interacts with the rest of the model in co-simulation mode. In this case, the solver properties for the Modelica part can be defined independenty of the embedded solver properties

- ContPoleZero
- ContStateSpace
- ContTransFunc
- Derivative
- EventClock_Support
- EventDelay
- EventGenerate

- EventVariableDelay
- FixedDelay
- Integral
- JumpStateSpace
- LastInput
- MaxMinInterval
- Ramp

- SineWaveGenerator
- StepGenerator
- Time
- VariableDelay
- ZeroCross
- ZeroCrossDown
- ZeroCrossUp

Table 22.3: Additional **Twin Activate** blocks supported by P when embedded solver option is used.

Limitations

The model transformation method presented here is used to provide support for most **Twin Activate** blocks other than input and output blocks. The choice of embedded solver, extends the set of blocks supported by the P code generator: in addition to blocks already supported by the P code generator as indicated in Appendix A.1, the blocks shown in Table 22.3 are also supported.

But even when all the blocks in the super block are supported by solver inlining, the resulting model behavior may not be identical to that of the original model. This is due more to the use of a fixed step solver, which imposes constraints in particular on the timing and synchronization of events, than the block replacements required by solver embedding per se. The dynamic behavior of the transformed model however should approach that of the original model (running on a variable-step solver) as smaller values of h are used for the sampling time.

Note that the super block, selected for solver inlining, should not include any activation input ports. Any activation sources used in the diagram must be included inside the super block.

Note also that if a super block present inside the diagram selected for inlining is atomic, it should also be explicitly designated as a super block to which solver inlining is applied.

A.1 Twin Activate blocks supported by the P code generator

Note that some of these blocks are only partially supported.

- Abs
- Accumulator
- ActivationIntersection
- ActivationSignalScoping
- AlwaysActive
- AndActivations
- Assignment
- Atan2
- Backlash
- Bias
- BusAssignment
- BusConstant
- BusCreator
- BusExtractor
- BusInput
- BusInput
- BusOutput
- BusOutput
- BusSignalScoping
- ConditionalNSelect
- ConditionalSelect
- Constant
- ContPoleZero
- ContStateSpace
- ContTransFunc
- ControlledSaturation
- Counter
- CrossProduct
- CumulativeSum
- DFlipFlop
- DLatch
- DataTypeConversion
- Deadzone
- Demux

- DiagonalMatrix
- DiscrEdgeTrigger
- DiscrPoleZero
- DiscrStateSpace
- DiscrTransFunc
- DiscreteDelay
- DiscreteIntegral
- EdgeTrigger
- End
- EventClock
- EventPortZero
- EventToCode
- EventUnion
- Exponential
- ExtractActivation
- Extractor
- FormattedString
- FromBase
- Gain
- GetActivationSignal
- GetBusSignal
- GetSignal
- Horner
- Hypot
- Hysteresis
- Identity
- IfExpressions
- IfThenElse
- IncludeDiagram
- InitialEvent
- Input
- Integral
- JKFlipFlop
- JumpStateSpace

- LastInput
- Log
- LogicalOp
- LookupTable
- LookupTable2D
- LookupTable2D_port
- LookupTable_port
- MathExpression
- MathExpressions
- MathFunc
- MatrixConcatenation
- MatrixDivision
- MatrixExpression
- MatrixExtractor
- MatrixGain
- MatrixInverse
- MatrixMultiplication
- MatrixReshape
- MaxElements
- MaxMin
- Memory
- MinElements
- Modulo
- ModuloCounter
- Mux
- Negate
- OmlExpression
- Output
- PID
- PCustomBlock
- Pow
- Power
- Product
- Quantization

- Ramp
- Random
- RateLimiter
- RealTime
- RelationalOp
- RepeatActivation
- RoundProduct
- SRFlipFlop
- SampleHold
- Saturation
- ScalarExpand

- SelectInput
- SelectOutput
- SetActivationSignal
- SetBusSignal
- SetSignal
- ShiftRegister
- Sign
- SineWaveGenerator
- Sort
- Step
- Sum

- SumElements
- Switch2
- SwitchCase
- SwitchN
- Time
- Toggle
- Transpose
- TrigFunc
- TruthTable
- UnitConversion