State Machine Library - User's Manual

Altair® Twin Activate® 2025

Contents

Preface

Altair® **Twin Activate**® is a modeling and simulation tools for dynamical systems. It provides a graphical block-diagram editor for constructing models using blocks. State machines [8, 10] provide a different approach to modeling synamical systems. Basic **Twin Activate** does not provide facilities for constructing state machines within models. **Twin Activate** semantics however is particularly well adapted for such constructions [1, 3].

The **State Machine Library** provides a set of blocks for constructing dynamical system components based on the paradigm of state machines. This user's manual provides information about the type of state machines that can be constructed using this library, and illustrates the use of its blocks through examples.

The **State Machine Library** provides one way of constructing state machines; it is not the only way. It provides a State machine formalism very much in the spirit of **Twin Activate**. It is a simple, yet powerful formalism, offering many features available in other formalisms, such as different variants of Statecharts, Stateflow [11], Scade [7], SyncCharts [4], UML State Machine [12], but it is not fully compatible with any of them.

The manual contains two parts. Part I is dedicated to discrete-time synchronous state machines, used in particular for modeling embedded controllers. Part II presents an extension to the continous-time case, used for modeling asynchronous state machines and hybrid systems.

The example models used for illustration in this manual are available as demo models of the library.

Part I

Discrete-time state machines

Chapter 1

Introduction

Twin Activate diagrams are based on interconnected components, called blocks, performing mathematical operations, and exchanging data. In simple cases, all the blocks are activated by the same clock at the same times (synchronously) and the order in which the block operations are performed is determined by the **Twin Activate** compiler. For example, in the following model
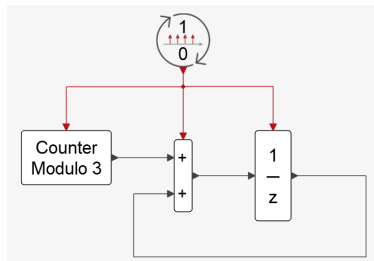
the Counter, the Sum and the Delay blocks are activated synchronously every unit of time[2]. The state of this system/model is constituted of the internal states of the blocks, in particular the internal states of the Counter and the Delay blocks. The model state is not apparent on the diagram but hidden inside its individual constituents.

A state machine model is a different way of modeling dynamical systems. In a state machine diagram, only one component is active at any time, and it represents the complete state of the model. There are no hidden states inside the component. For example, the following diagram represents a state machine modeling a simple thermostat[3]
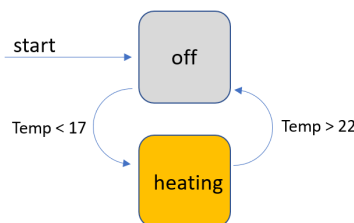
where the system can be in one and only one state at a time (system is in off mode or in heating

---

[2]Since the Delay block does not require its input value to produce its output value, it can update its output before the Sum block does, so the order of output update operations could be either Counter, Delay followed by Sum, or, Delay, Counter followed by Sum. The block states are then updated in any order.

[3]The temperatures are expressed in centigrade in this model.

mode). This representation of states is sometimes referred to as diagrammed states. The block diagrams and the state machines represent two orthogonal and complementary approaches to modeling dynamical systems. Even when they are both available in a modeling environment and can be used in the creation of the same model, they are often provided as separate, communicating, tools each with its own semantics. This can create ambiguity in the behavior of the overall model and imposes many limitations.

The state machine implementation presented here is fully based on **Twin Activate** synchronous reactive semantics. This allows for very intimate interactions between the block diagram and state machine components of the model without any risk of ambiguity. Blocks in block diagrams may contain state machines, and states in state machines, block diagrams. The overall system being implemented in the same environment, model compilation, both for simulation and code generation purposes, is done by a single compiler.

1.1 Block diagrams vs State machines

In reality, the distinction between block diagram modeling and state machine modeling is not so categorical. Many block diagram formalisms accept blocks running on different clocks in the same model (with different interaction constraints) and there exist different flavors of state machine formalisms where multiple components can be active simultaneously by allowing concurrent (parallel, orthogonal) and hierarchical constructions, with support for extended (not diagrammed) states. These extensions bring closer the two approaches. The State machine formalism presented here is an attempt to fill this gap even further and even completely close it.

Twin Activate provides direct manipulation of "activation signals" (clocks) based on the semantics of synchronous languages. This makes it particularly amenable to use for constructing the dynamics of discrete finite state machines. Consider for example the simple thermostat with two states presented above and the following implementation of it in **Twin Activate**.
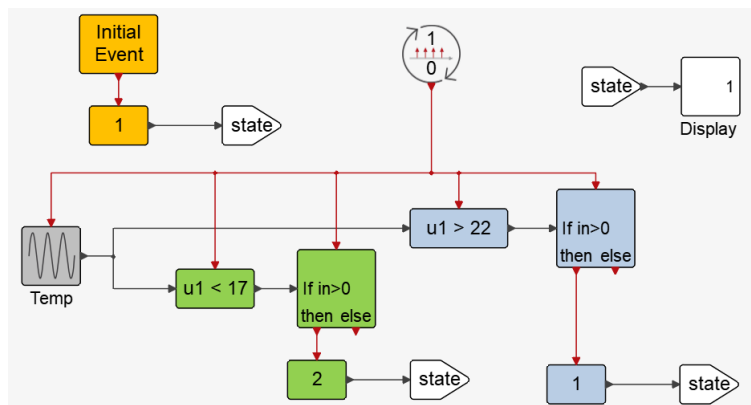
The state signal represents the state taking values 1 and 2 corresponding, respectively, to the Off and the Heating states of the state machine. This state is initially set to 1 (Off) and then it is updated by the **Constant** blocks, which are activated when the temperature falls below 17 or goes above 22. The explicit manipulation of activation signals through condition blocks (in particular, the **IfThenElse** block[1] here) is used to activate the two **Constant** blocks at different instants of times to change the state.

---

[1]**IfThenElse** block is strictly speaking not a block. Along with **SwitchCase**, they are language constructs presented as blocks, but their output events are not generated by them; they are redirections of their input events, so, the input and output events are synchronous. They can be compared to if-then-else and switch constructs of most programming language.

This model remains relatively simple because the underlying state machine transitions can be made solely dependable on the input (note that this is done by accepting extra useless operations; for example, if the temperature is below 17, the state is systematically set to 1 regardless of whether it is already 1, or not). This kind of construction however is not possible in general. The construction of an **Twin Activate** model to allow the dependency of the new state on the previous state is possible but can be complicated. For example, an alternative realization for the same state machine, not producing the extra activations, is given below.

The activations of the **Constant** blocks now depend both on the previous value of the state and the input. Based on the value of the previous state, only the relevant test of the input temperature is performed. So, for example if the state is 1, the input (temperature) is only tested against 17 and changed if needed; it is not tested against 22. It turns out that this way of storing the previous value of the state and using it with the **SwitchCase** (Even Select) block to activate the part of the model corresponding to that state can systematically be used in the construction of models for any state machine.

This example illustrates three points: the construction of state machine models in **Twin Activate** from scratch can become very complex and users should not be expected to do it without assistance; all that is needed to construct them is already available in **Twin Activate** and the construction can be systematic; and finally, the resulting model does not look like a state machine. These observations have been the motivation and starting point for the development of the **State Machine Library**.

1.2   Discrete-time assumption

The state machines considered here are activated by external events. All state transitions and activations occur by these events, and all the signals used in the state machine may only be updated synchronously at these event times. The top diagram encapsulating the state machines realizes a discrete-time **Twin Activate** block (an Atomic super block to be exact), whose activations provide the external events activating the state machines. So, the block inputs are read, processed, and the outputs are generated when the block is activated. The processing may include multiple iterations of the same event in the state machines inside the block, but the iterations are transparent from the outside.

The block activation in the model is often done by a periodic clock (**SampleCLK** block) but it can be done by any series of events. For example, the block can be activated only under certain conditions, such as only when the value of an input to the block is changed.

11 ASM library

Future extensions of the ASM library will provide blocks to support asynchronous state machines and hybrid systems. For asynchronous state machines, the activation times are not imposed by external events; events can be internally generated at any time. Queuing processes, for example the Poisson process, can be generated by such state machines. The resulting blocks would be "always active". If in addition the state machines support continuous-time states, they can be used to model hybrid systems.

Asynchronous processes and hybrid systems can already be modeled in **Twin Activate**, and the demos present several examples. But these constructions are not always based on a state machine methodology. A specialized library would provide a systematic state machine approach for the construction of such models.

Chapter 2

State Machine Library (ASM)

The ASM library provides special blocks for the construction of state machines. These blocks transparently set up the underlying infrastructure required for defining and updating states for State machines. The library can be used to construct machines with hierarchy and parallelism (concurrence). It also provides options such as Run-to-completion and History.

To facilitate the usage of the library, the blocks are designed so that the resulting **Twin Activate** diagram graphically resembles standard state machine diagrams, as it can be seen in specialized tools and languages for constructing state machines. This is done by representing states and state-transitions respectively by (special) Blocks and Activation signals. The definition of state transitions and activations are greatly simplified, and the required constraints are automatically imposed, for example the constraint that two exclusive states should never be active synchronously.

The following diagram shows an implementation of the thermostat example using ASM library blocks
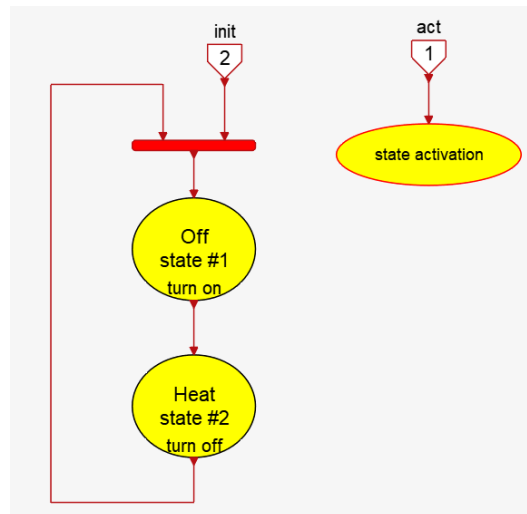
Using the library blocks, user needs not worry about the implementation of the state update mechanism. He/she creates the diagram by placing the state blocks and a state activation block in the diagram (state blocks must be given unique numbers; this constraint is not imposed at construction time, but it is verified by the compiler). The contents of the block states implement their actions.

The state blocks come equipped with an **Initialize** block and an **Action** block. The **Initialize** block generates an initialization event, and the **Action** block provides subsequent action events as long as

the state block is active. The Off state is for example implemented as follows
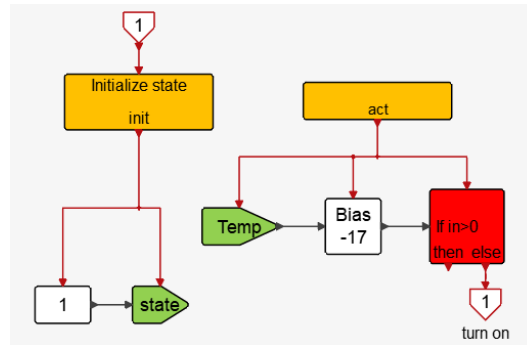

The **Initialize** block receives the event entering the state block, i.e., the event associated with the transition to this block. The initialization event here is used to set the value of the output signal state to 1. After initialization, the act block provides activation signals, which are used to test if the input temperature (Temp signal) has fallen below 17, in which case the event exits the block, initializing the Heat state. The content of the Heat state is shown below
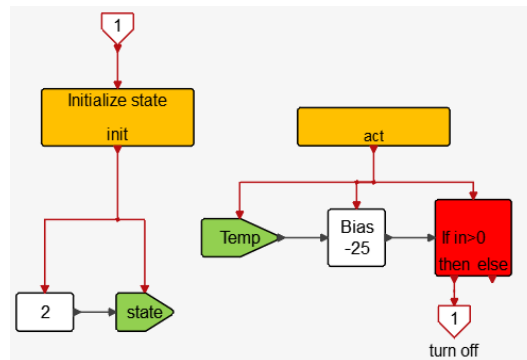
Its operation is straightforward to understand: the state signal is initially set to 2, and subsequently the temperature is tested, leading to an exit if it goes beyond 25.

Note the use of activated Set and Get signal blocks to read and write the input temperature and the output state signals. Use of these blocks is not an obligation; state blocks are "normal" super blocks, so regular ports may be added to them. But for simplicity and in particular to keep the diagram visually consistent with state machine diagrams, it is recommended to use activated Set and Get signal blocks and avoid the use of regular links to cross the block state boundaries. Note also that all the blocks inside the state block are explicitly activated. This is not an obligation either; for example, the Bias and the **IfThenElse** blocks would function identically through the inheritance mechanism in the absence of input activation ports. But the use of explicit activation is recommended because it makes the diagram more readable.

A Thermostat block can then be created by encapsulating the above state machine diagram in the StateMachine super block below
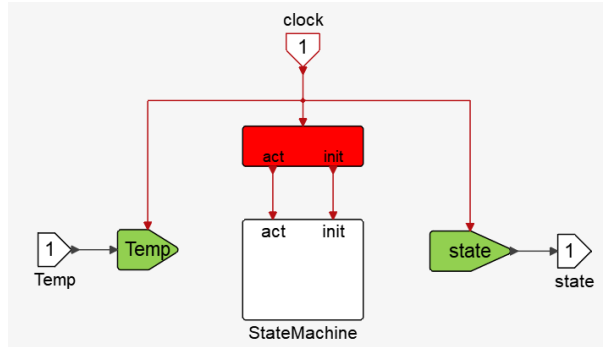
The StateMachine super block layer is not required in this case because the model does not contain parallel state machines. In case it did, the diagram would include multiple StateMachine super blocks, one for each state machine. The systematic use of a super block layer however is useful because it separates the management of the input and outputs from the state machine. In this top diagram, the inputs and outputs are associated with named signals. The activation signal of the block (the clock in most cases), is separated into an initial event ($init$) and the rest of the events (act). The initial event is used for the initialization of the state machine.

Note that all the user defined blocks in this model are specific to this particular example. The user has not implemented any logic for implementing the state machine mechanism. The mechanism is present inside the library blocks but is not visible, and more importantly, needs not be manipulated by the user.

The resulting Thermostat block can now be used as any other block, in an **Twin Activate** model. The following is a simple test model showing its operation.
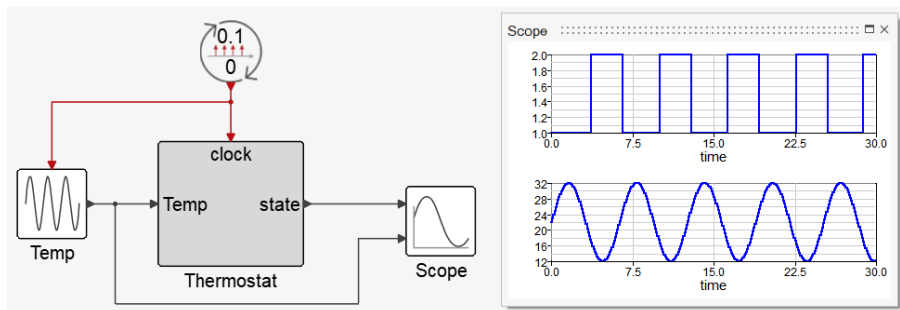

This example shows the basic idea and methodology behind the **State Machine Library**. It does not show the details of the way the events cause state activations and transitions (in particular the exact timings and the synchronization constraints are not illustrated). These issues are presented and discussed later in this document.

2.1   ASM library palette

The ASM library blocks are standard **Twin Activate** blocks or super blocks based on such blocks. Some contain invisible parameters used for the internal state-machine machinery, made transparent to the user, some are programmable super blocks, so with no visible corresponding diagram, but all use only blocks from the Activate Base library. There are no "special" blocks or semantic extensions used behind the scenes. This means that all the functionalities available in **Twin Activate** can be used with models including ASM library blocks. These include model parameterization, code generation, etc.

The ASM library contains blocks specifically developed to be used for the construction of state ma-

15 ASM library
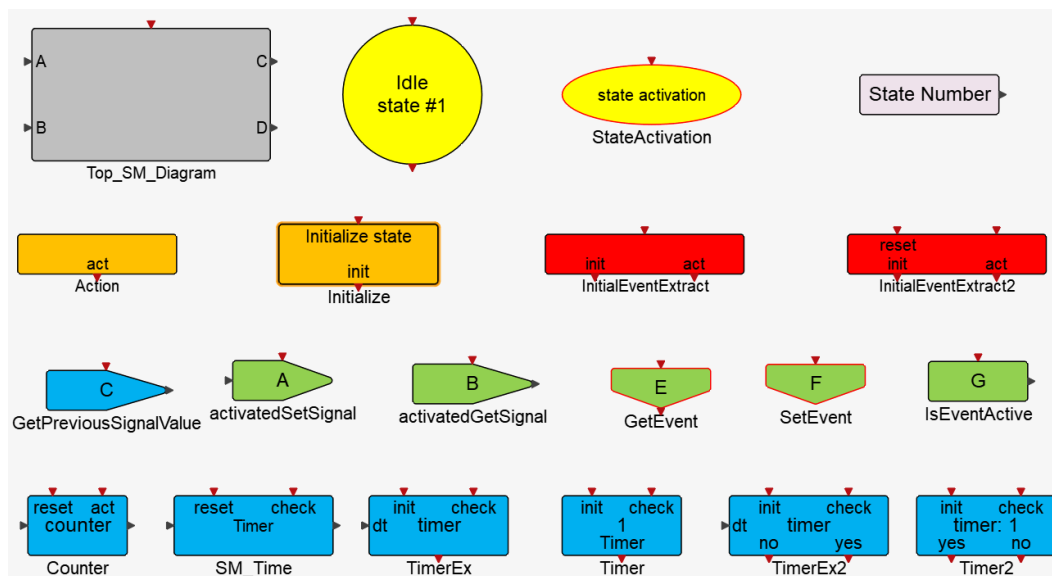
chines:

The library includes also utilities blocks for realizing the behavior of various switches and buttons often needed to prepare the input signals to the state machines, in particular signals produced interactively for example through the use of the **Keyboard** block, or imported from data files. The library also includes additional blocks, already available in the Base library, parameterized specifically to be used in the construction of state machines. The sub-palette **ContinuouslyActive** contains blocks dedicated to continuous-time state machines, which will be discussed in Part II.

2.2   Steps to create a state machine model using ASM library blocks

To create a model using state machines, it is important to start by identifying the states of the system. In simple cases (like the simple thermostat above), the state is fully specified by a single state machine. But in general, this is not possible or desirable, and parallel and hierarchical constructions must be envisaged. There are also cases where extended states (non-diagrammed states) must be used.

Once the overall architecture of the model is specified, the construction can begin. The construction should follow certain rules to reduce the risk of errors and to obtain a visually acceptable result. In the method based on ASM library blocks, diagrammed states are represented by **StateBlock** blocks, which are masked super blocks. To obtain graphical consistency with the way state machines are represented, these super blocks do not have regular inputs and outputs (all data exchanges with the outside is carried out by activated Set and Get signal blocks). They do however have Activation input and output ports. These ports and associated activations are used exclusively to model state transitions.

The construction can start by placing a **Top_SM_Diagram** block in the model. This super block represents the top level of the state machines. To facilitate model construction, it is created as a prototype; it contains already some of the blocks commonly used at this level: StateMachine blocks, Get and Set signals connected to inputs and outputs, and an **InitialEventExtract** block. The StateMachine blocks contain **StateBlock** and **StateActivation** blocks, and the **StateBlock** contain **Action** and **Initialize** blocks, etc. This skeleton of the model is provided to facilitate the construction of the model by reducing the need to copy additional blocks from the ASM library.

The steps are illustrated here on an example. The original model is an example model from Embed[1]: StateChartTank.vsm. This model contains a simple Statechart state machine used to control the fluid level in a Tank. The Embed model is imported in **Twin Activate**, but the importer does not support Embed StateChart diagrams. So, this part is left as an empty super block, to be implemented by the user. Here the focus is on this part of the model.
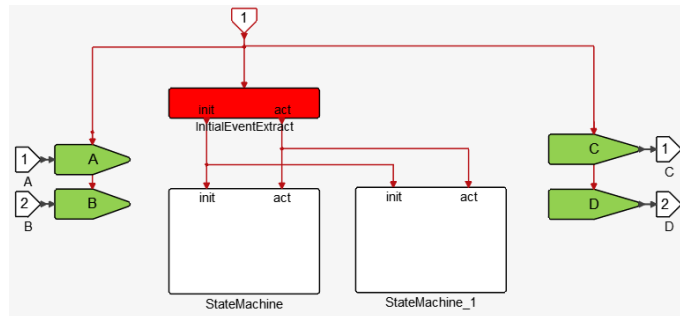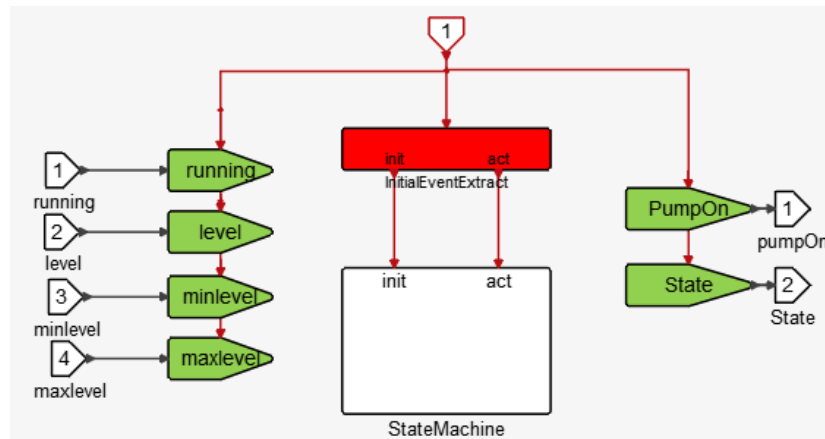
The State Chart block in Embed is shown below
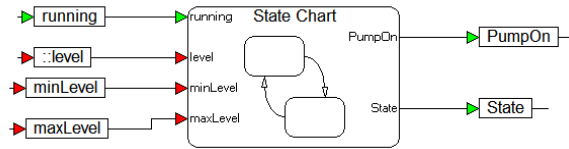
In **Twin Activate** it looks similar
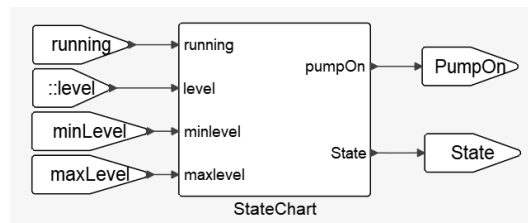
but its content is empty. The StateChart super block will now be considered a **Top_SM_Diagram**. This block can be copied from the ASM library palette and placed in the diagram. Its content is shown below
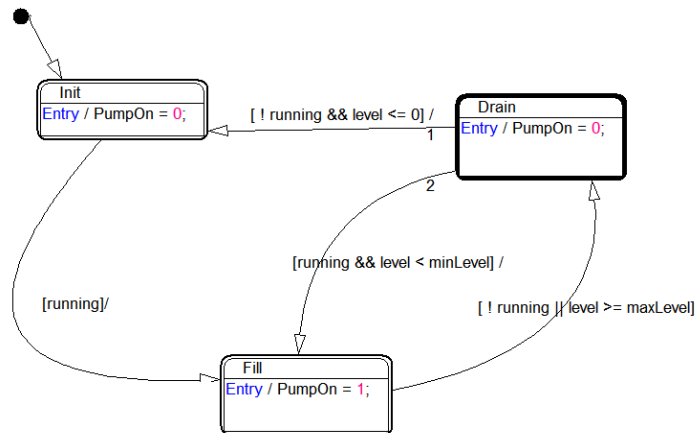

Two parallel running state machines, and two inputs and outputs are assumed in this skeleton. The content is specialized to this case as follows


[1]https://www.altair.com/embed/

17 ASM library

The content of the StateChart block in Embed is

This system has clearly 3 states and the top left state is the initial state. The $PumpOn$ value is set to 0 or 1 when entering different states, and the transition conditions are visible on the arcs connecting them. The content of the StateMachine block can now be specialized to this case. Originally it contains
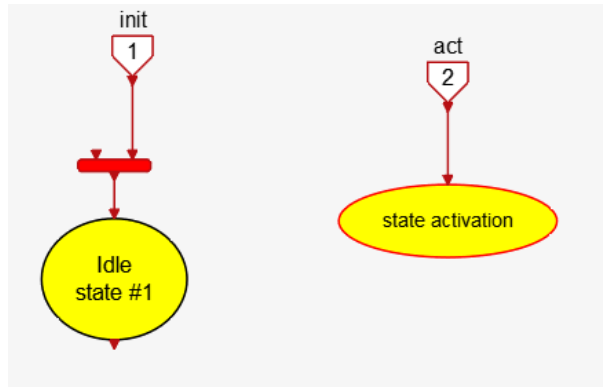

The BlockState is duplicated twice and renamed and renumbered:
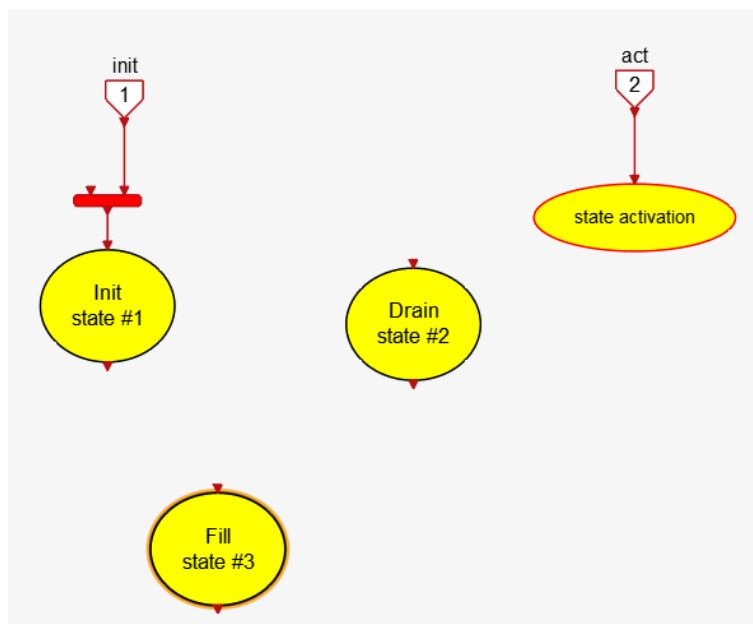
To implement the transitions and the actions, note that in the Statechart model the entry actions are placed inside the states and the transition conditions on the outside. In **Twin Activate** both are placed inside the **StateBlock** blocks. Starting with the first state, Init, the skeleton
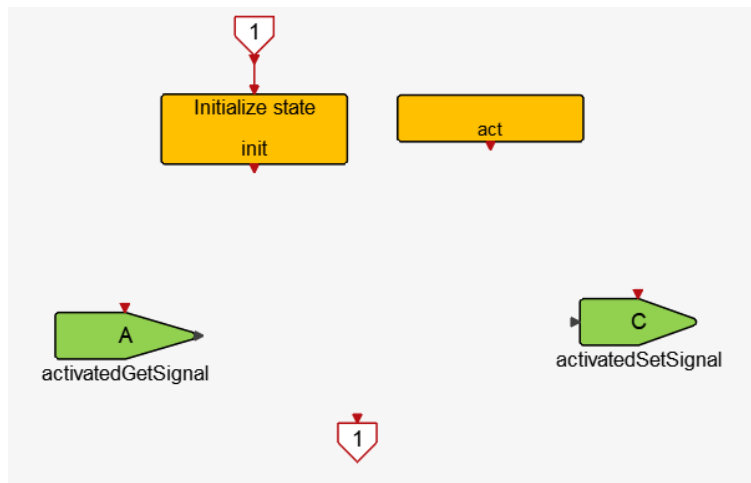

is specialized as follows
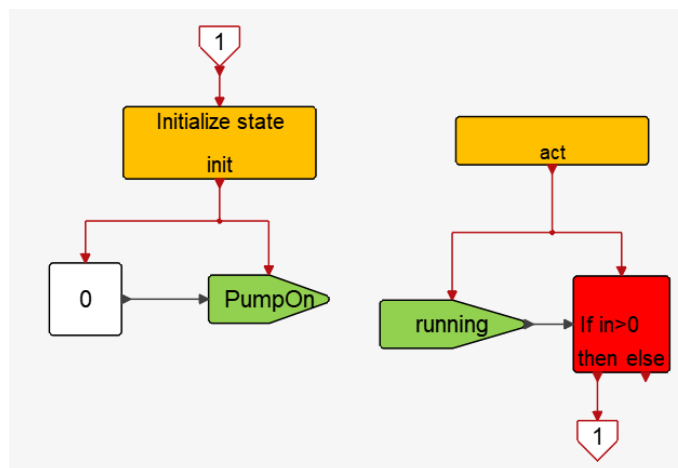
Note that both the Entry action and the exit condition are captured inside the state. **StateBlock** for state 2 is specialized as follows
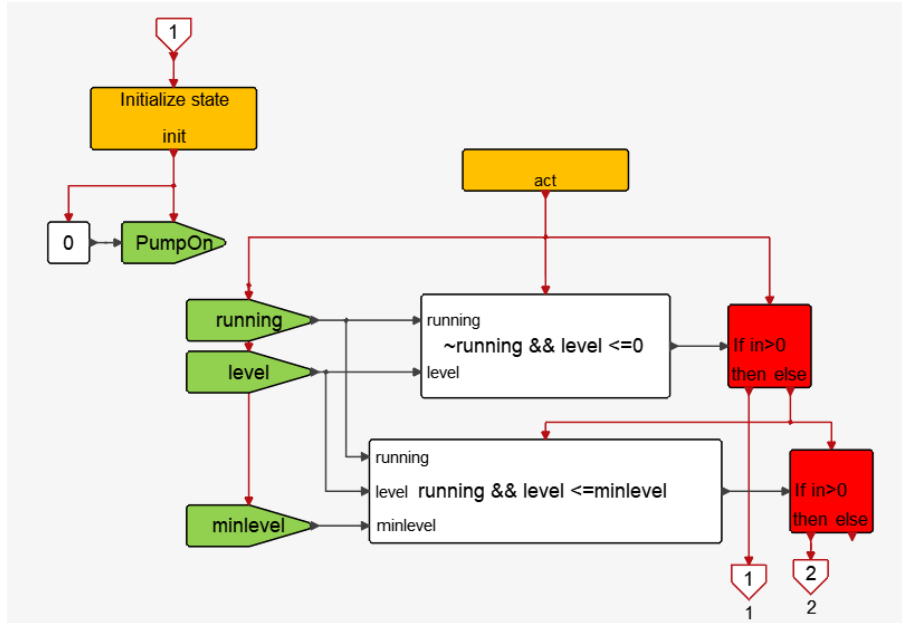
The order in which the **IfThenElse** blocks are placed gives priority to the first exit to be consistent with the arc numberings in the original Statechart diagram. These conditions can be implemented in a simpler way using a single **IfExpressions** block:
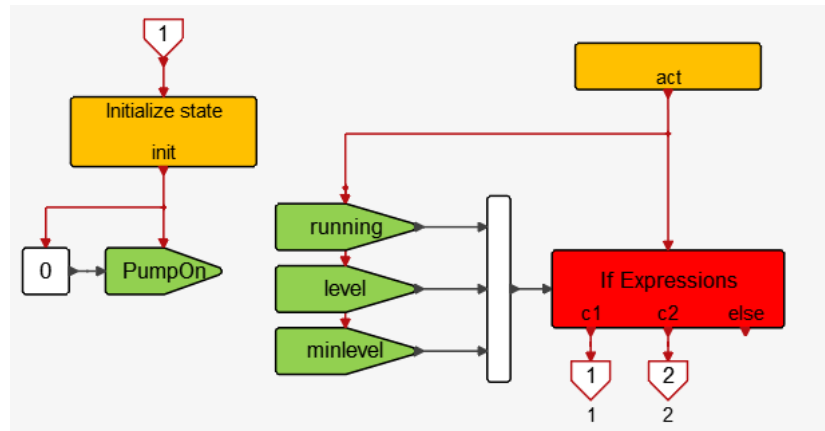

But in this construction, the expressions are not seen on the diagram, so the diagram is less readable, at least for the purpose of the presentation in this document. Note however that the **IfExpressions** block is very convenient to use for defining exit conditions. It is a super block containing a cascade of **IfThenElse** blocks associated with logical expressions.
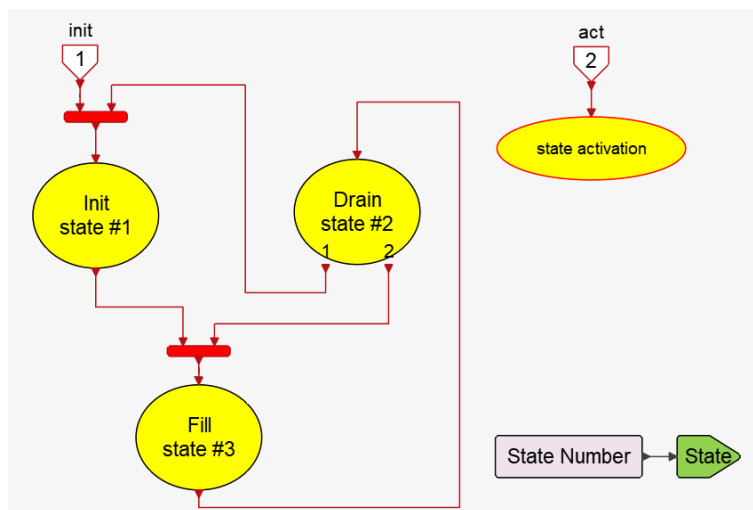
The last state is similarly specialized as follows
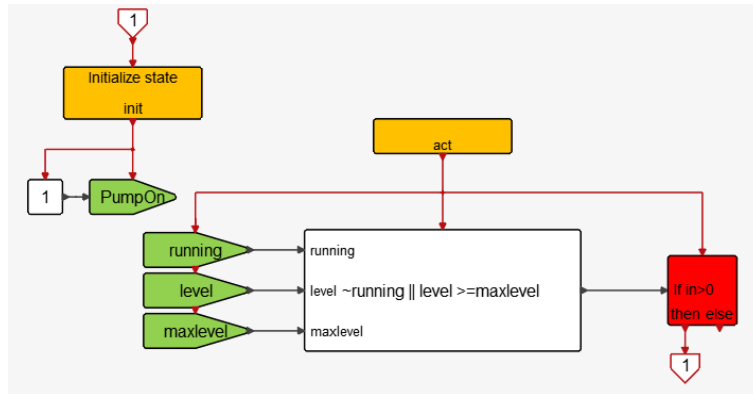
The content of the StateMachine block is now as follows
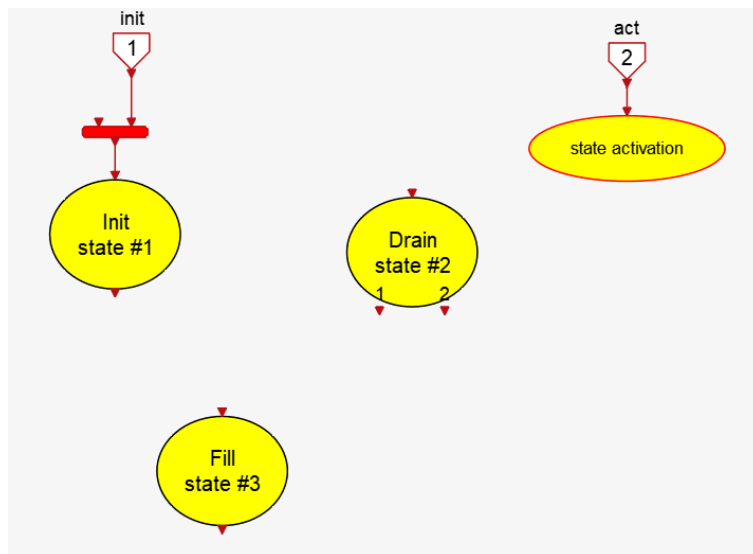
and transitions can be realized by connection the state block ports. A block to obtain the current state number is also used to provide this information as an output (used in the original diagram to display the current state):
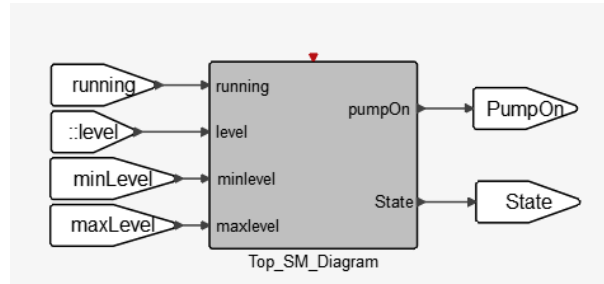

The inside of the **Top_SM_Diagram** is now complete. It can now replace the original StateChart block

21 ASM library

by a single cut and paste operation
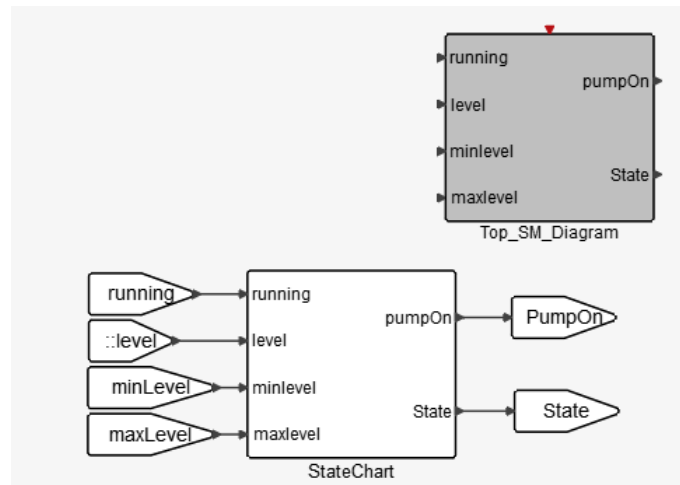

to obtain

The only question that remains is how to activate this block. In the original Embed model, the State Chart was operating in continuous time with a fixed-step solver. The ASM library blocks considered here are designed to work with discrete events. To obtain similar behavior, the activation can be done with a discrete clock having the same period time as the step time of the solver. Truly continuously activated state machines require additional considerations and considered in the second part of this document. A continuous-time implementation of this model will be in particular provided in Section 10.2.
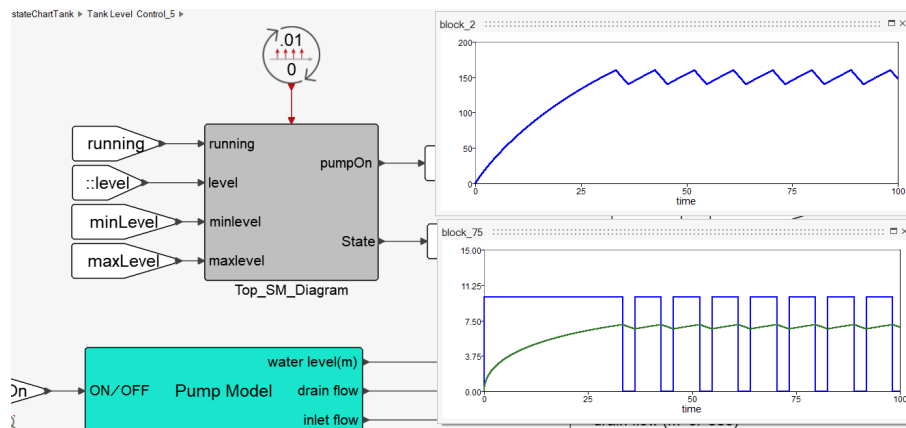
The block-based construction of state machines in **Twin Activate** seems to require more work than the use of specialized state machine modeling paradigms. However, the construction is carried out fully in "native" **Twin Activate**. A single language (**Twin Activate**) is used in the construction of the complete

model. There are many advantages in using a single environment modeling environment for the whole system.

A single view over the state machine including the contents of the states is not available in **Twin Activate** but it is possible to obtain a "global view" using the "Show in New Windows" item available in the contextual menu, applied to every state, as it can be seen below:
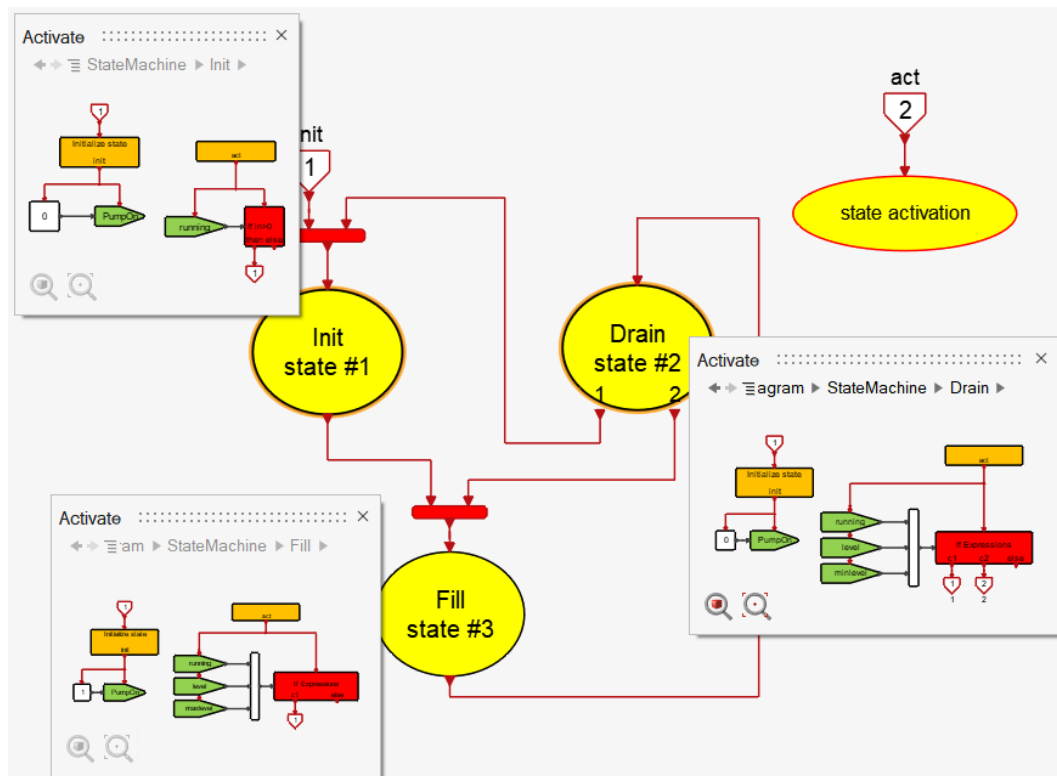
Chapter 3

How does it work

The main advantage of implementing state machines in "native" **Twin Activate** is that there are no new semantics to define. The blocks of the ASM library simply provide syntactic sugar to facilitate the construction of state machines; they do not introduce new semantics. In different state machine formalisms, the role of transitions, different actions inside the states, exit conditions (in particular in the presence of hierarchy) must be specified, in addition to the syntax and semantics of the textual description of different actions inside the states and the transition conditions. In **Twin Activate**, since the transitions are standard **Twin Activate** activation signals, and the actions and conditions are realized using standard **Twin Activate** blocks, the behavior is defined unambiguously.

The state machine modeling methodology presented here can simply be understood by examining some of the key blocks of the ASM library. These blocks have already been encountered and briefly presented in the previous examples; more detailed explanations are provided here. The action of the "run to completion" option is also explained.

3.1 State machine

The two main blocks for realizing a state machine are the **StateBlock** and the **StateActivation**:
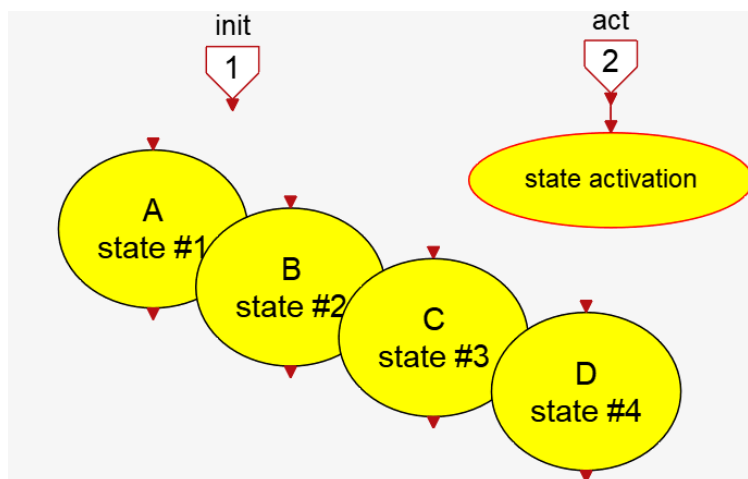
A unique number is associated with each **StateBlock**, and the **StateActivation** block redirects its activation to one and only one of these **StateBlock** blocks at any time. The **StateActivation** block is

a super block that uses the previous value of a "state variable" and a **SwitchCase** block to realize this redirection. The value of this state variable is set by the **StateBlock** when it is initialized.

A **StateBlock** contains two special blocks: one is **Initialize** for initialization, which is activated by an entering-event when transition into the state occurs; another, the **Action** block, which captures and provides the activations "sent" by the **StateActivation** block.
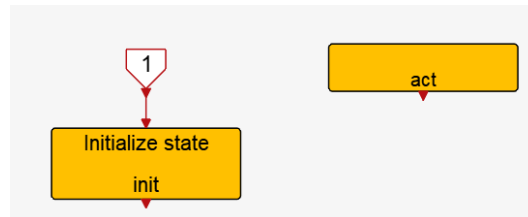
When a **StateBlock** is initialized by an entering event, the **Initialize** block sets the value of the state variable to that of the state number so that the **StateActivation** block redirects future activations to this block (until another state is activated and the state variable is changed again). The $init$ signal out of the **Initialize** block is simply a replicate of the entering event. The $init$ and act activation signals are used to realize state actions. Exiting the state is done by sending the act signal to the outside of the **StateBlock** and using it to initialize another **StateBlock**. Exit-specific actions can also be realized at this point.

The **StateBlock** can also include another state machine (or even machines). The $init$ and act signals then can be used to initialize and activate this machine. The following shows a typical configuration of part of the inside of the **StateBlock** associated with the state A above:
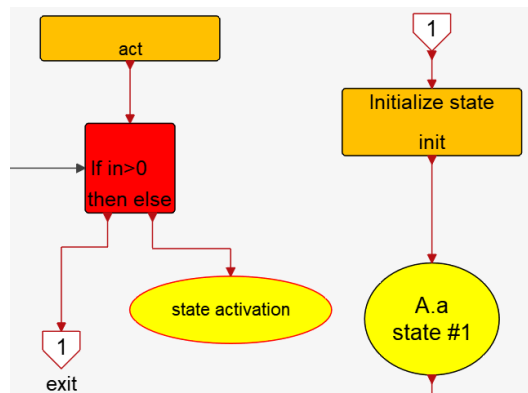

The substate $A.a$ is initially activated every time the state A is entered. Other initialization actions can also take place by init. The act signal activates actions during the activity of state A. In this example, these actions activate the state machine inside A, and when some condition is satisfied, causes an exit from A.

In this example $A.a$ is activated by init. If it were not, say it was activated by an **InitialEvent** block, then this state machine would not be initialized to state $A.a$ every time the state A is entered, instead, it would start in the state where it was when A had exited previously. This means that A would have kept its history and would restart based on that. Since a state can have multiple initialization inputs, the internal state may keep its history or be initialized depending on the way it is entered. This feature provides a lot flexibility for defining the behavior of the state machine.

Note that the original state machine and the one inside A each have their own state variable. This state variable is associated with the path of the diagram where the blocks are present, and as such, are

unique. Users need not worry about these "hidden" state variables. The **StateBlock** and **StateActivation** blocks manage it internally through invisible mask parameters. These parameters can be seen by inspecting the blocks using the "Details" editor option and the use of Mask Editor. However, they should not be modified since this may modify their behavior and thus break the consistency of the system.

The use of the diagram unique path to name the state variable means that there is no risk of confusion between different states in different state machines. But this implies also that two parallel state machines cannot live in the same diagram; an additional level of hierarchy must be used for that. This is the reason why the **Top_SM_Diagram** block by default proposes an additional level of hierarchy in its inside diagram; the default skeleton proposed inside **Top_SM_Diagram** is shown below:
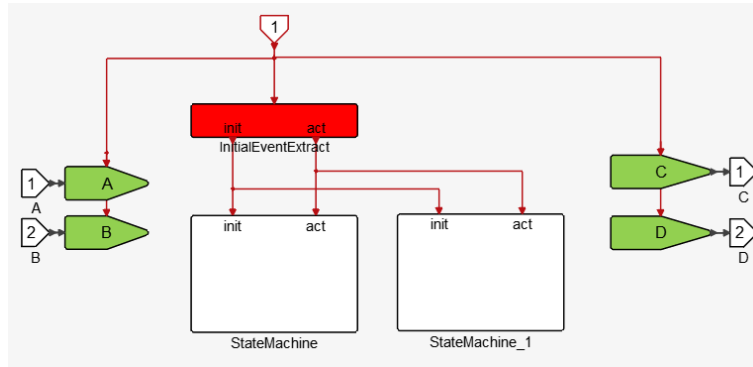
At the top level, it is recommended to use this additional level of hierarchy even if only one state machine is present. This creates a separation between the management of inputs and outputs inside the block **Top_SM_Diagram** and the state machine itself.

3.2 State machine parameterization

So far, the state machines considered did not contain any parameters or extended states. In the tank level control example, the max and min levels could have been considered as parameters but (constant) signals were used to represent them instead. In the next example, the use of parameters in the construction of ASM models is shown.

3.2.1 State machine parameter strucutre

In ASM blocks, an OML structure named **SM_Params** is used to pass parameters everywhere in the model, including inside **StateBlock** blocks. The **SM_Params** variable traverses the **StateBlock** boundaries, even though this block is masked. This is done through an invisible mask parameter named and valued **SM_Params**. **SM_Params** is defined in the context of the **Top_SM_Diagram**. This context was not edited so far because the state machines did not use parameters; so, its content was simply defining default values of some state machine options, which will be discussed later:

```
SM_Params =struct;
SM_Params._RTC_=true;
SM_Debug('on');
```

Parameters can be added to the structure **SM_Params**; they are then available for usage everywhere in the construction of the state machine. The following example shows how the state machine can be parameterized.

The state machine is used to debounce a button used to set a switch. The switch is set if the button is pressed and maintained pressed for a period of time specified by the parameter $timeout$, then released. Then the switch remains set until it is reset by an xternal signal.The following state machine is used to realize this behavior
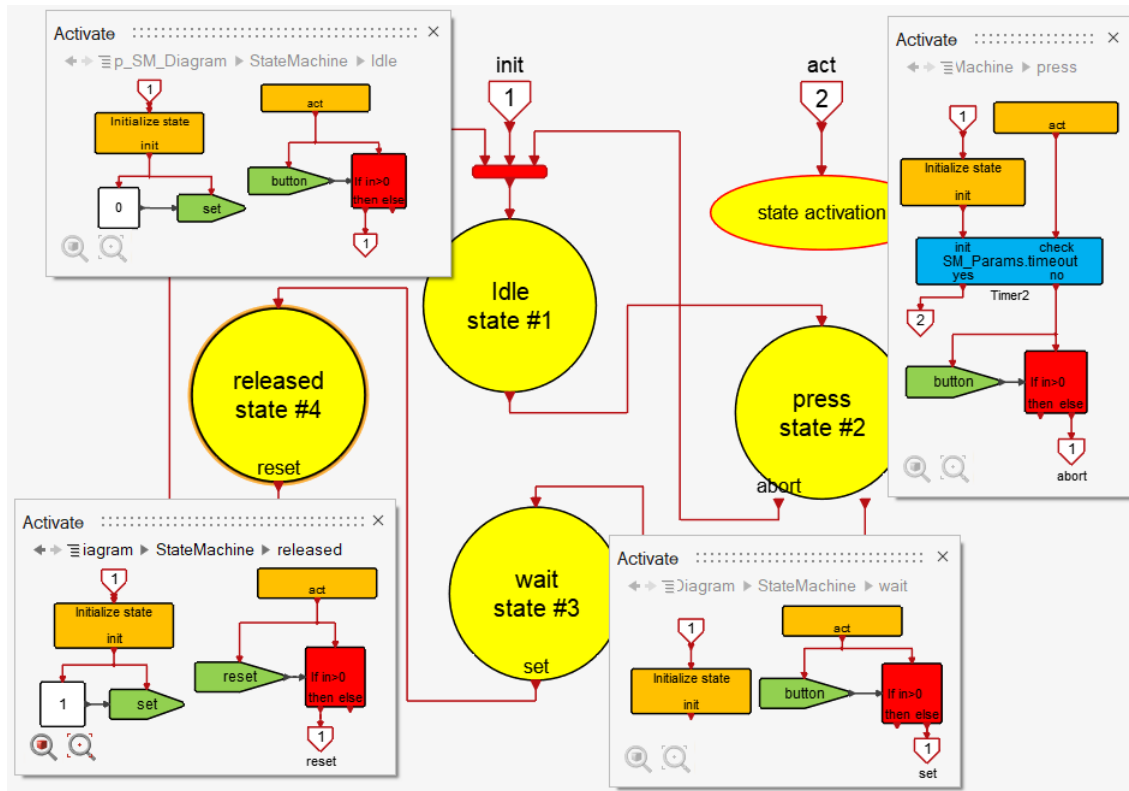
Note that the $timeout$ parameter is used in the **Timer2** block as `SM_Params.timeout`, defined in the context of the **Top_SM_Diagram** super block:

```
SM_Params =struct;
SM_Params._RTC_=true;
SM_Debug('on');
SM_Params.timeout=timeout;
```

The following model shows the operation of this state machine
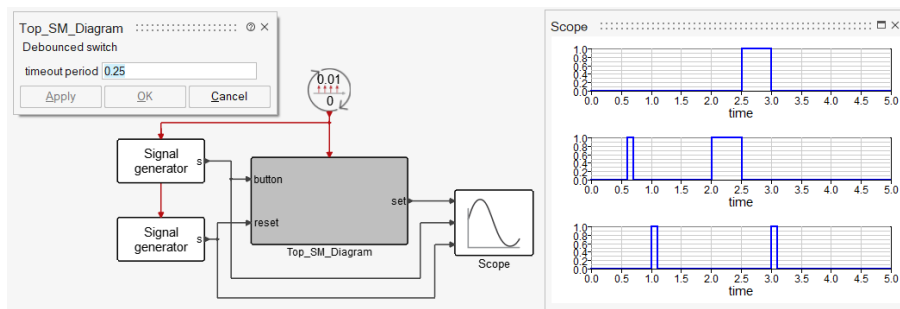
In this case $timeout = 0.25$, so the first button press, which lasts less than 0.25 seconds, does not set the switch. The second press does it once the button is released. The switch remains set until it is reset by the $reset$ signal.

3.2.2 Direct access to parameter structure

Model parameters defined in the `SM_Params` structure are available inside all states and can be used for the definition of block parameters. Referring to such a model parameter requires referring to the strucutre. So, for example to use a parameter `XX` as block parameter for a **Gain** block, the gain parameter should be defined as `SM_Params.XX`. This can be cumbersome, especially if the same parameter is used more than once.

It is possible provide the content of the `SM_Params` inside a state without having to refer to the structure. This can be done using the **OML** function `OpenTable`. This function extracts the content of a structure and use it to define the corresponding set of variables and values in the specified environment.

By placing the following instruction in the Context of a **StateBlock** diagram, the model parameters can be used inside the diagram directly:

```
OpenTable(SM_Params,getcurrentenv);
```

So, in the case of the `XX` parameter, the value of gain can be defined simply as `XX`.

3.3 Hierarchical state machines

The implementation of hierarchical state machines was discussed earlier but the examples seen up to now did not use hierarchy. An example is presented here to illustrate their implementation. This is a variation on a similar model used originally in [3].

In this model, the state machine implements a controller for a traffic light. The traffic light is placed at the intersection of a main road and a secondary road. The presence of a car on the secondary road, behind the traffic light, is detected using a sensor. The traffic light, on the side of the secondary road remains red in the absence of car. When a car arrives and is detected, the light switches to green only after a fixed delay (in case the car is still present). The light switches back to yellow, then red, after fixed time delays or as soon as no car is detected on the secondary road.
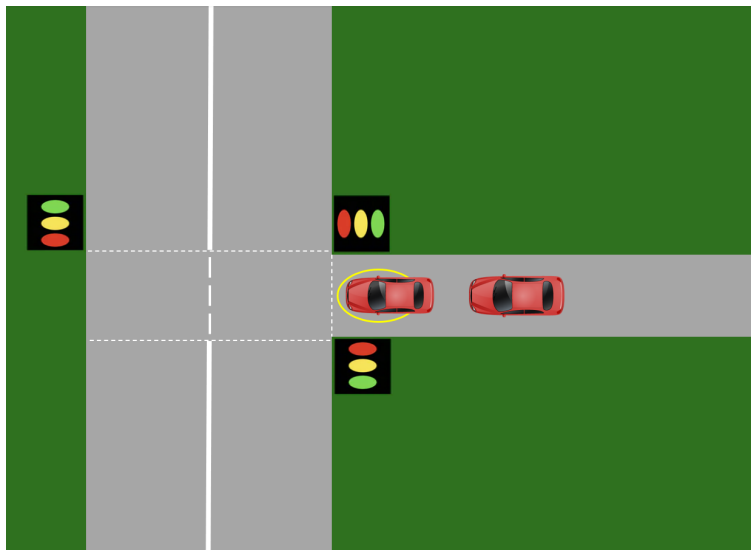
This model contains 5 states representing each a combination of the color of the traffic light on the main road and that of the secondary road: $Red|Green$, $Red|Yellow$, $Green|Red$, $Yellow|Red$; a transient Red|Red state is also considered for when the traffic light is turned on.

29 ASM library

The implementation of the states is simple in this case, they set the value of the signals representing the two traffic light colors and wait for a fixed amount of time before leaving the state. The only state which has a different behavior is $Green|Red$. When the light on the main road is green, it stays green as long as no car is detected on the secondary road, and even then, the exit from the state only may occur after a delay.

The state machine used to implement the traffic light controller is shown below (top left diagram). Below it, there is the $Yellow|Red$ state content: a simple delay implemented using the Timer block. This block from the ASM library can be used to start a timer using an init event with a timeout value given as parameter. It generates an event (synchronously) when activated by check event if the timeout period has run out. The other states function similarly, except for the $Green|Red$, which implements a more complex logic. Its content can be seen in the top right diagram. Its behavior is implemented using another state machine.[1]
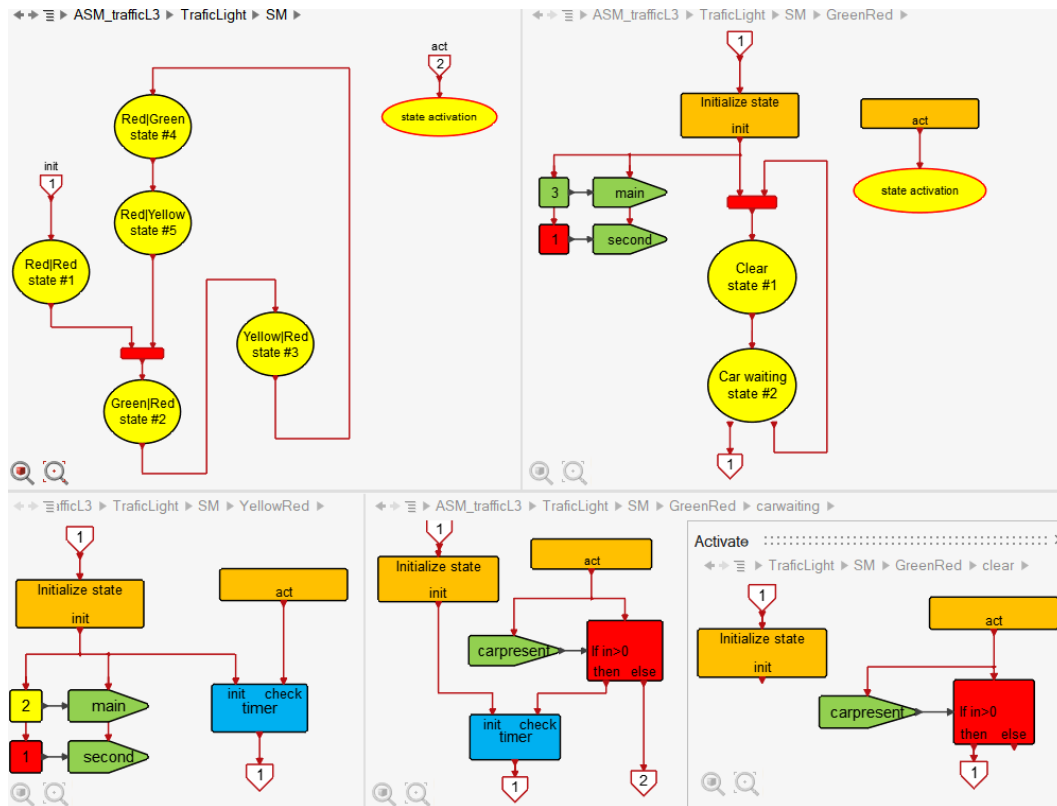
The first state named Clear, the content of which can be seen at the right bottom, simply monitors the car-sensor input to detect the presence of a car on the secondary road. As long as the secondary road is clear of vehicles, it stays active. When a car is sensed, then the second state, CarWaiting, becomes active. A timer is started, and the presence of the car is monitored. If the timer runs out while the car is still present, the state is exited causing the parent state $Green|Red$ to exit as well. If a car is not sensed anymore[2], the state is exited but the activation loops back to the clear state.

Note that here the states Clear and CarWaiting are the "substates" or "child states" of the $Green|Red$

---

[1]An additional level of hierarchy could be used here by placing this new state machine inside a StateMachine block. But this is not necessary since there are no requirements for parallel state machines and all the input output signal are named at the top.

[2]In theory this should never happen. It could if the car has run the red light or it has backed up. It happens also if the sensor fails.

state. This latter can be called their "superstate" or "parent state". The $Green|Red$ state child-states can be active only when their parent is active.

The colors of the lights are coded as integers: 1, 2 and 3 representing respectively red, yellow, and green. This number is converted to three individual 0 and 1 values corresponding to each color in the top diagram. So, outputs can be used to individually control different colors of the two traffic lights.
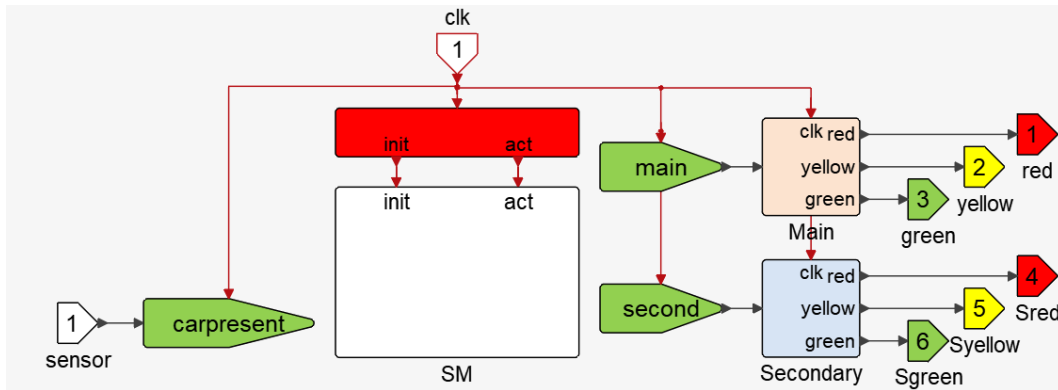
The Light Traffic block is tested in a model where the arrival of cars in the secondary road is modeled as a random Poisson process. There is no departure if the light is red. If cars are present, and the secondary traffic light is green or yellow, the departure takes a fixed amount of time for each car. In the following, the model and the corresponding simulation results are shown. The red, $yellow$, and $green$ subplots indicate the status of the traffic lights on the main road. The $Sred$, $Syellow$, and $Sgreen$, that of the secondary road. The plots also show the number of cars waiting in queue on the secondary road and the sensor output (sensing the presence of cars on the secondary road at the level of the traffic light).
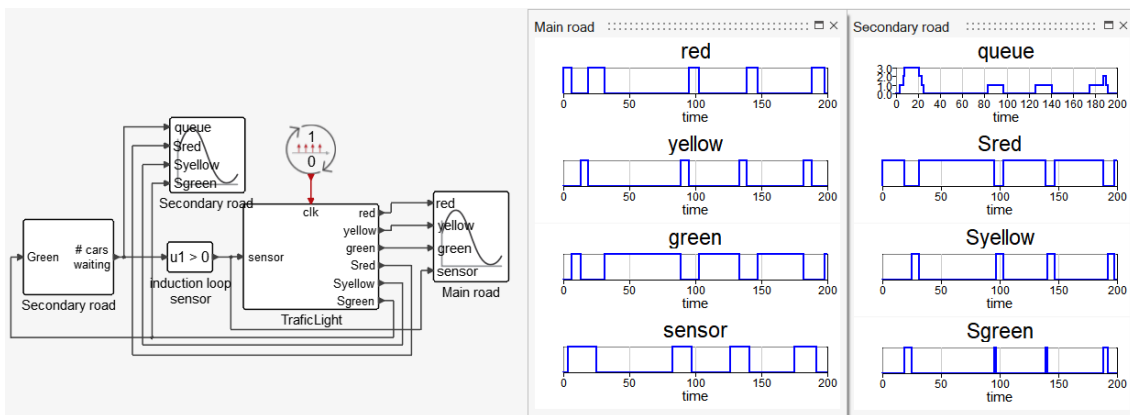
3.4   Parallel state machines

In the previous section, it was shown that two state machines can co-exist in the same model, the states of one being the substates of a state of the other. Here, the use of parallel state machines is considered. Parallel (concurrent) state machines run side by side, synchronously, exchanging information.

An obvious implementation in parallel can be realized by using two (or more) **Top_SM_Diagram** blocks in the same **Twin Activate** model. In this case, the two state machines can only exchange data through their regular signal ports, as do all regular **Twin Activate** blocks. This is similar to the use capsules in

UML-RT state machines. This construction however has limitations. Using two state machines inside a single **Top_SM_Diagram** block allows for a more intimate interaction between the two state machines.

The example presented here represents the operation of a simple elevator. The state machine is used to realize the controller.

The controller receives three signals:

1. The button number pressed to select the destination floor; it can take values 1 to 7, and 0 if no button is pressed. Only one button press (button inside the cabin or call button outside) is registered at any time

2. The output of the floor sensor; it provides the floor number when the elevator is in close vicinity of a floor and 0 if not.

3. The door sensor output. Its value is zero if the elevator door is closed, otherwise it is 1.

And it provides three signals:

1. The target-floor number; it is used in particular to control the light (on/off) on the button panel.

2. The signal to control the movement of the elevator taking values -1, 0 and 1 for, respectively, going down, staying in place, and going up.

3. The signal to control the opening and closing of the door: 1 to open, 0 to close.

Simple models of the dynamics of the movements of the elevator, and the opening and closing of its door are used for testing the system. The button press is emulated by the Keyboard block: the destination floor can be selected during model simulation by pressing the corresponding number on the keyboard:

Enter the desired destination floor (1 to 7) using the keyboard during simulation
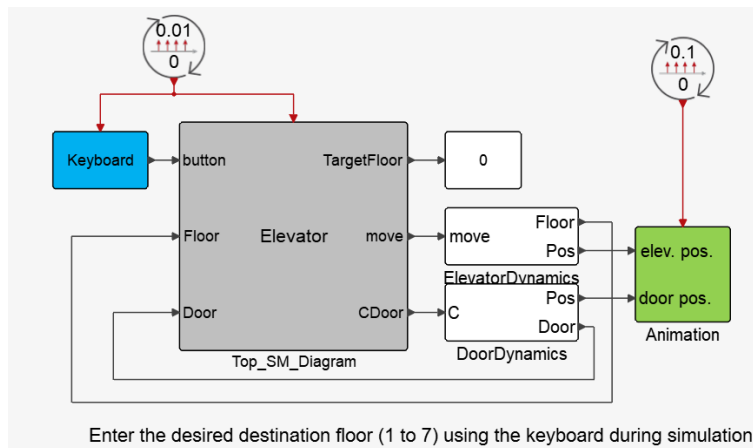
The simulation result is illustrated via a simple animation showing the movements of the elevator and its door.

To model the controller, two state machines running in parallel are used: one for the selection of the destination floor by the buttons; the other for the control of the movements of the elevator. The presence of the two state machines can be seen inside the **Top_SM_Diagram**:
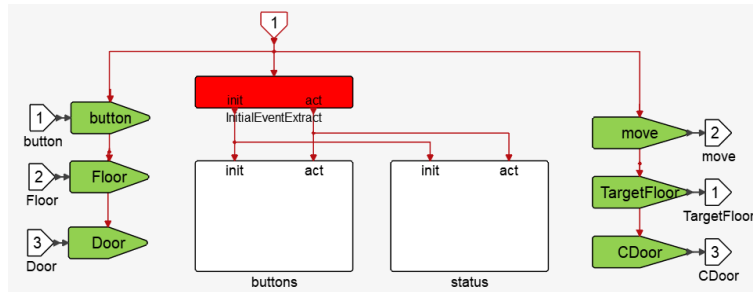
The buttons state machine has two states: a floor is selected, or it is not. The value of the signal $TargetFloor$ represents the selected floor or is zero if no floor is selected. The $TargetFloor$ value is set by pressing of the button, only if its value is zero (a selected destination cannot be modified by pressing another button before reaching destination). The un-selection, i.e., setting $TargetFloor$ to 0, can only be done by the other state machine when the elevator reaches its destination. The content of the state machine buttons is shown below
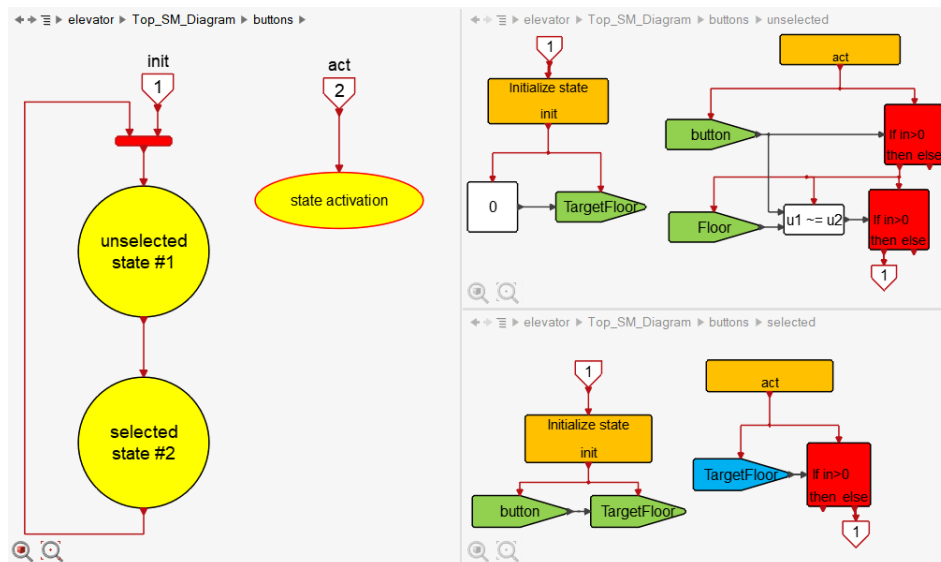
Note that if the button corresponding to the current floor is pressed, the selection is not made. The second state machine handles the movements of the elevator. When a destination is set (the value of $TargetFloor$ is not zero), if the elevator door is closed, it initiates the movement of the elevator in the proper direction. If the door is open, then it initiates the closing of the door first. When the elevator reaches its destination, the $TargetFloor$ is set to zero. This information is used in the first state machine to re-activate the buttons.

This second state machine has also two states. They can be seen blow

33 ASM library
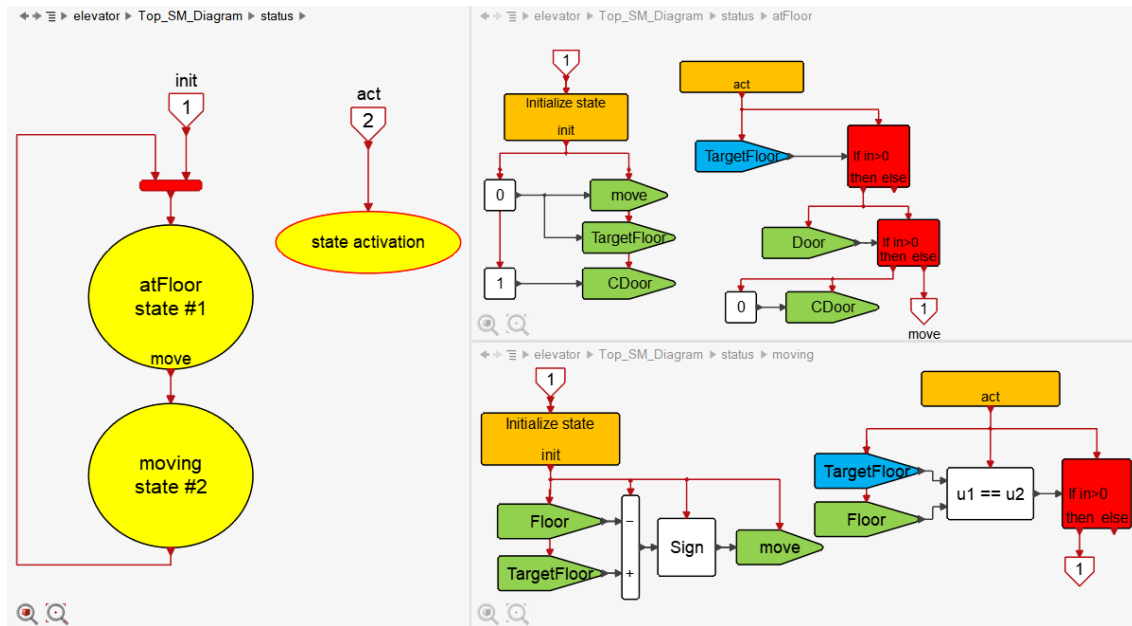
Entering in the atFloor state, the move (elevator motor command) and $TargetFloor$ signals are set to zero, and the elevator door opening is initiated. Then the $TargetFloor$ is monitored; if it becomes non-zero, then the door is checked. If closed, the state changes, if not the closing is initiated, and the checking repeated.

In the moving state, first the direction of the movement is determined, and the signal is sent to the motor. Then the current location of the elevator (floor) is checked against the destination, and the state exits when the two are equal.

3.5 State machine activation and run-to-completion option

When the **Top_SM_Diagram** is activated, its activation is directed to the state machines defined inside and causes the activation of active states and possibly state transitions and initializations. The first event is in general used for activating the initial states and other initialization tasks. This is done by separating the initial event from subsequent ones at the entrance of the **Top_SM_Diagram** block:
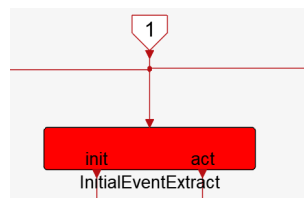
This is not imposed; the initialization may be based on the initial activation signal produced by **InitialEvent** blocks. In that case act would be obtained directly from input activations. The subsequent act signals are then used to activate all the active states. The run-to-completion option modifies the behavior of the system as follows. If this option is true, then any state transition leads to a repeat of the source activation signal. Since the **Top_SM_Diagram** is declared Atomic, this means that all the state machines inside are reactivated without time advancing. The activations repeat until no additional transition occurs. This is called run to completion.

To illustrate the difference of state machine behaviors with this option set to true and false, consider

the following example where the task of the state machine is to compute, given an input U, the largest integer I such that

This computation can be easily implemented in an OML function:

```
function I=SumTo(U)
  I=0;
  S=0;
  while S<U
    I=I+1;
    S=S+1/I;
  end
end
```

A naive state machine implementation with a single state can be done as follows
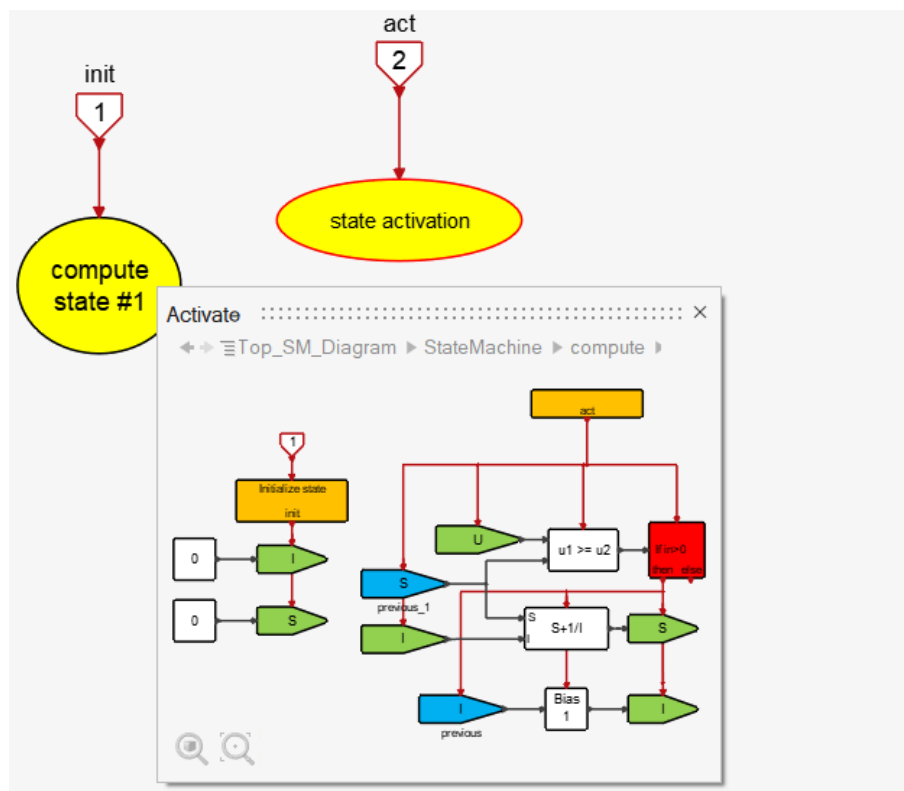

The simulation result shows that the value of I is computed but takes multiple time steps to reach the correct value as it can be seen below. This is not a satisfactory implementation since even if an initial delay at the start to reach the correct value of I is acceptable, if the input is not constant, the output may never be correct. The reason is that the signals I and S are not reset when the input changes.

35 ASM library
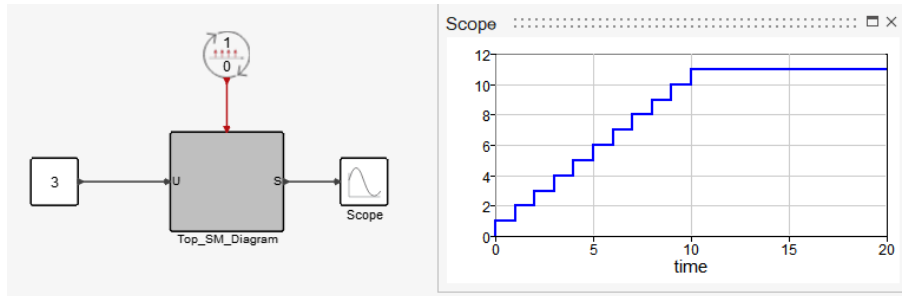
The desired behavior is that the state diagram performs all the required iterations to reach the stable value for each incoming activation. Such iterations resulting from the activation of an Atomic super block can be realized using a **RepeatActivation** block. This block is used inside the Initialize block of the states so that at each initialization, i.e., state transition, the activating event is repeated (assuming the run-to-completion option is set to true). This means that the act signal of the state, subsequent to its initialization, gets activated without advancing the time. The use of the **RepeatActivation** block is of course transparent to the user. Its usage depends on the value of SM_Params._RTC_, which can be set in the Context of the **Top_SM_Diagram** block. By default, its value is true.

In the above model, even though the run-to-completion option is true (by default), the iterations are not repeated to reach the final value during the same time instant. The reason is that the (unique) state of the state machine is only transited into once (at initial activation). An alternative formulation using two states is presented below. In this formulation, the initialization and activation parts are placed in two different states, the extended state I and S are reset to zero at the end of the iterations, and the final value of I is written in the output (a different signal called I_final). The model is shown below.
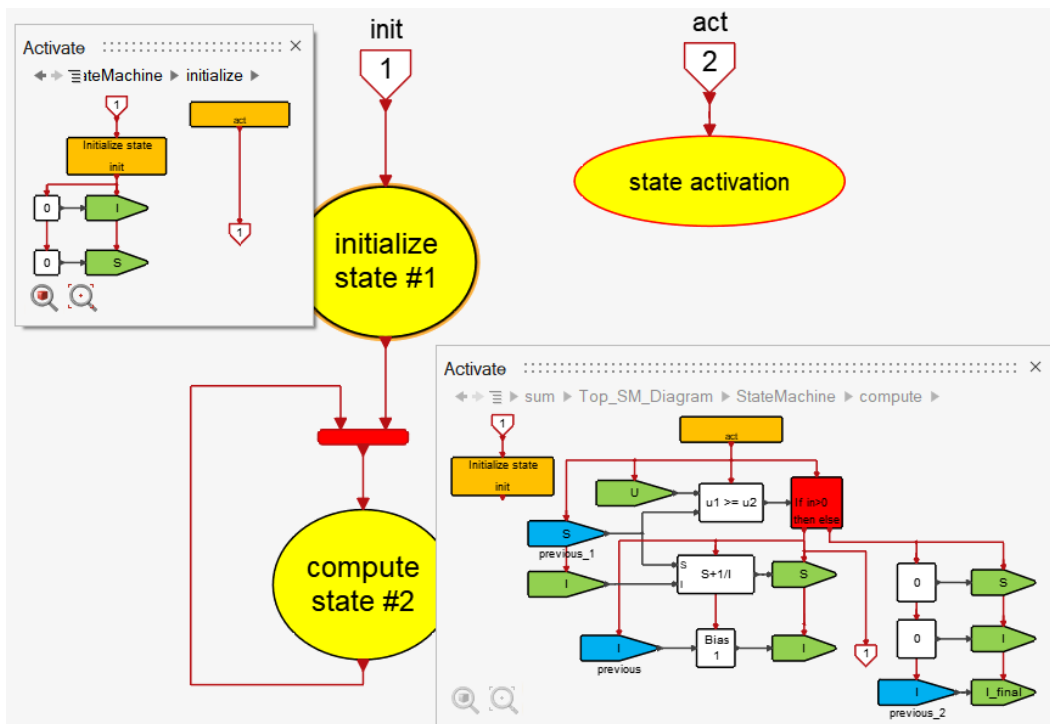

The first state initializes the counter I and the partial sum S, the second state performs one iteration for each action. But the state 2 loops back and re-initializes itself. Even though no action is defined for this re-initialization, with the run-to-completion option true, act is activated immediately, again resulting a

new re-initialization, and this "looping" continues until the threshold test provides a false result, ending the iterations, resetting I and S, and providing the final result. So as far as the block behavior in the diagram is concerned, the output of the block computes the `SumTo` function of its input at each activation, instantaneously:
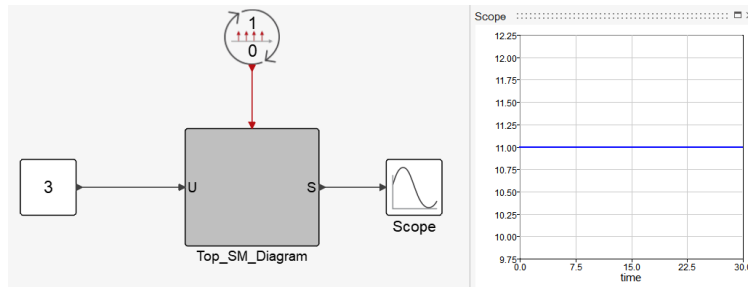

Now the block realizes the function `SumTo` and can be used with a non-constant input:
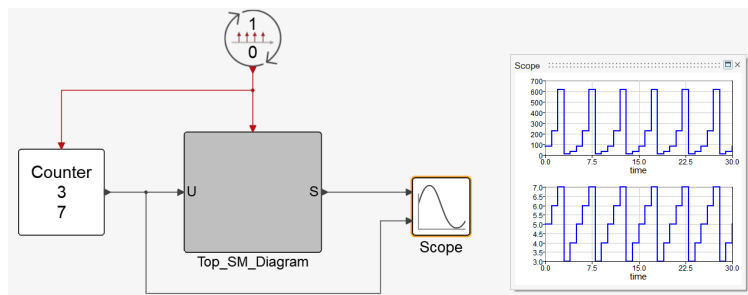
Note that in this diagram, the repeated operations taking place inside the **Top_SM_Diagram** are transparent; the block at each activation provides only one output corresponding to the result obtained at the final iteration.

With the run to completion turned off, the behavior of this block would resemble that of the previous block and would not operate correctly.

A simpler implementation of the above behavior can be obtained using a single state machine state as follows:
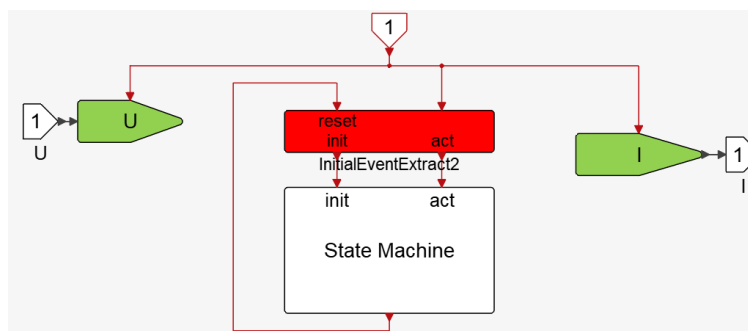
In this construction, each outside activation yields first an init event followed by act event due to the run to completion option. The **InitialEventExtract2** block has a reset port, which once activated, restores the initial behavior of the block, i.e., the next event is directed to the init port.

The state Machine is then implemented as follows

37 ASM library
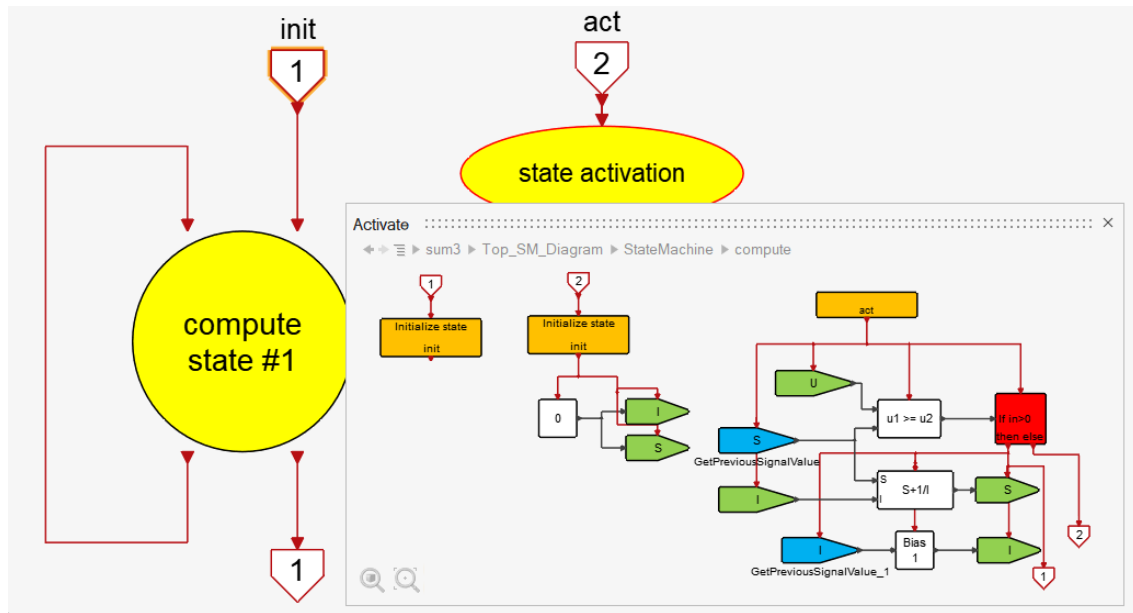
The state can be initialized in two different ways. It is initialized by the external activation through its second activation port. This sets to zero I and S variables; then the iterations start. After each iteration, if the terminating condition is not satisfied, the state is exited through its first output activation port, reinitializing itself through the first activation port. This reinitialization does not set the variables to zero but results in an immediate activation of the state (assuming the run to completion option is on). The behavior of this model is similar to that of the previous model.

The method used to implement the `SumTo` function by taking advantage of the run to completion option can be applied more generally to implement iterative algorithms. The following is an example where a generic optimization algorithm is implemented and made available as a block in the **Twin Activate** environment.
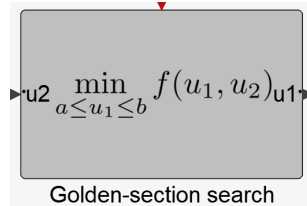
Golden-section search algorithm

This algorithm is used to find the minimum of a function $f(x)$ over an interval $a \leq x \leq b$. It finds the minimum if f is unimodal, for a given tolerance. An OML implementation of the algorithm is given below

```
function   x = golden(f, a, b)
  gr = (sqrt(5) + 1) / 2;
  tol=1e-6;
  while abs(b - a) > tol
    c = b - (b - a) / gr;
    d = a + (b - a) / gr;
    if f(c) < f(d)
      b = d;
    else
      a = c;
    end
  end
  x = (b + a) / 2;
end
```

The **Twin Activate** block would use this algorithm to solve the optimization problem

$$\min_{a \le u_1 \le b} f(u_1, u_2)$$

for the input u_2 providing as output the value of u_1 realizing the minimum every time the block is activated:

Golden-section search

The function f, the bounds a and b, and the tolerance threshold tol are the block parameters
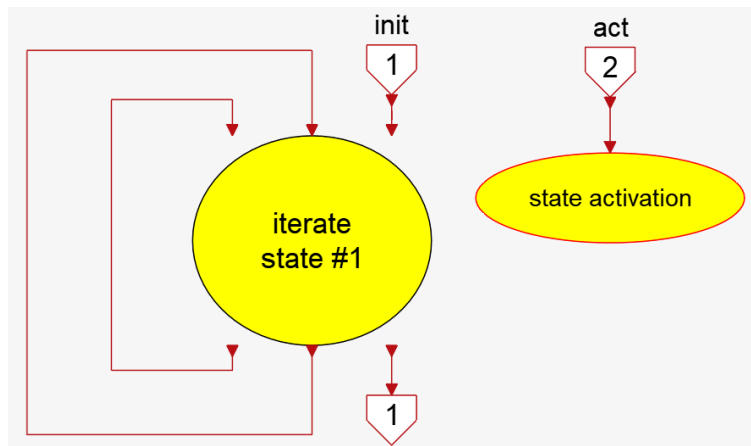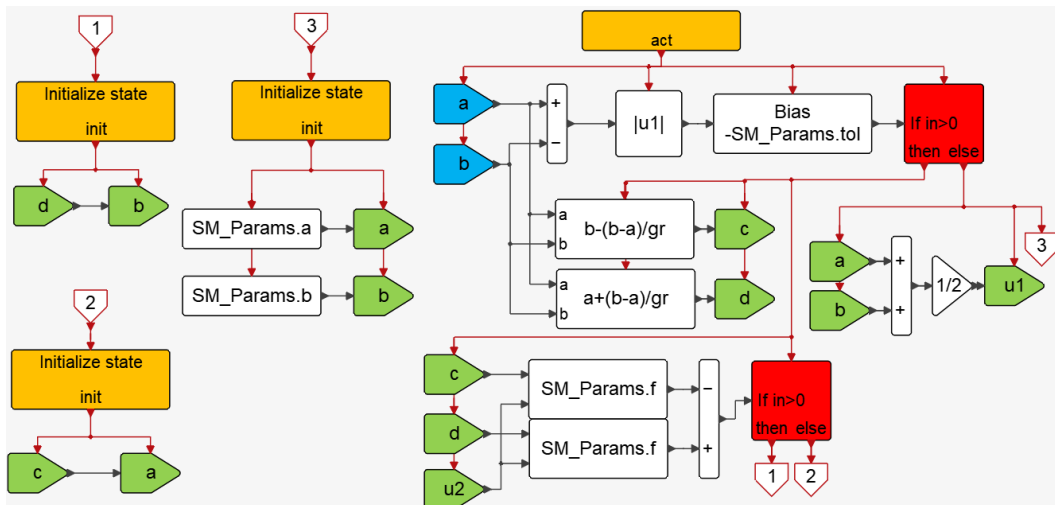

The function f is provided as a string containing any expression valid for the MatrixExpression block. The implementation is done using a single-state state machine inside the top diagram as shown below
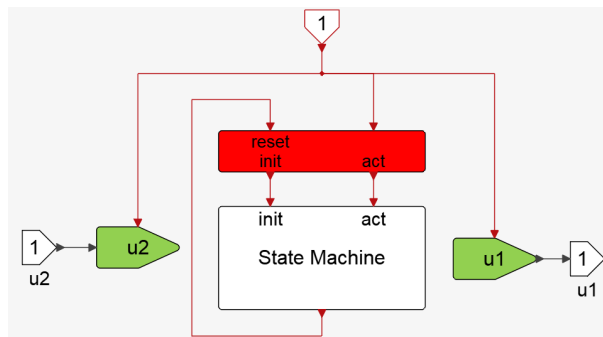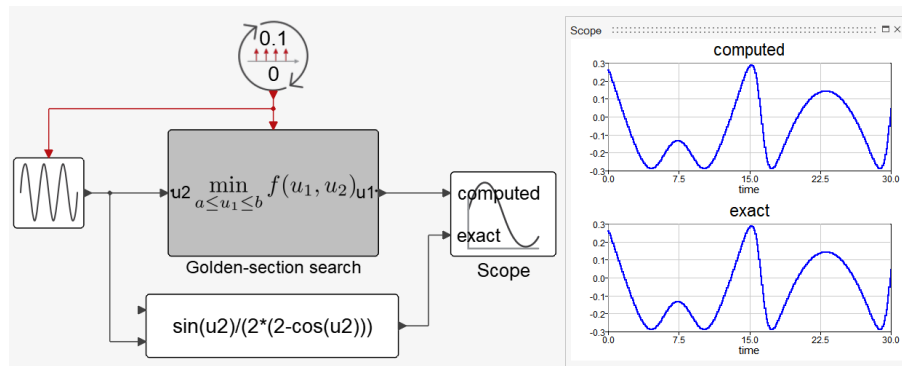

The state machine diagram is

where the content of its single state is shown below

39 ASM library

The analogy with the OML code is easily established. The simulation result for block parameters illustrated in the block GUI above and a sinusoidal input is shown below; the output of the block is compared with the exact solution:


The same block can of course be implemented as an **OmlCustomBlock**, or a **CCustomBlock** for more efficiency. The state machine implementation is converted to C code (thanks to Atomic property of the **Top_SM_Diagram**), so the performance is higher than an OML implementation but does not reach that of a well written C code used within a **CCustomBlock**.

Note that in some cases, there could be advantageous to set the run to completion option to false. Doing so guarantees that each state machine activation is due to an activation of the **Top_SM_Diagram** block. So, if the block activation is obtained from a fixed rate activation clock with known period, then time delays can be implemented by counting the state activations, and so implementing timers. If the run to completion option is true, then a single block activation may lead to multiple state diagram activations, so counters cannot be used to track the time. In that case, Timer blocks based on the Time block should be used. But the Time block may not always be convenient to use when generating code for certain targets.

The following example shows a model with false run to completion option. This model implements a simple watch, which starts running starting from zero as soon as the simulation starts. It counts the elapsed number of seconds and minutes, and displays them. Here the **Top_SM_Diagram** block is activated by a **SampleCLK** with period 1, and the simulation is running in real time mode. The seconds are incremented in a Counter block inside a state directly activated by this clock. This unique state of the state machine, loops back to itself when the count reaches 60, restarting the count from 0, and

activating the single state of a parallel, similar state machine counting the number of minutes. The activation of the counting of the minutes is done by the event min as can be seen below:
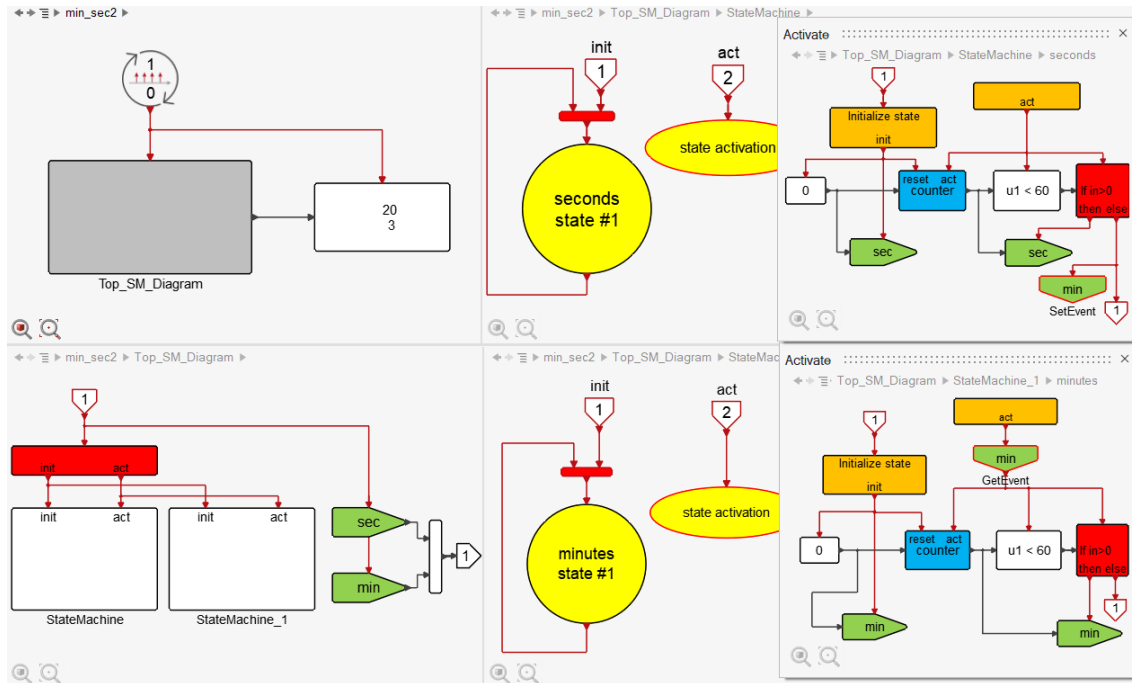
This model functions correctly if the run to completion option is set to false. For models relying on state activations to correspond exactly to source activations, setting this option to true may have unexpected results: the simulation result could be incorrect or the simulation may hang in an infinite loop at a time instant.

In the above example the consequence is an incorrect result as it can be seen below:
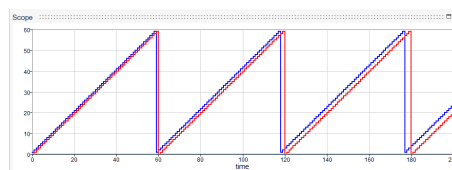
The red curve corresponds to the number of seconds, which correctly resets to zero every 60 seconds in the model with false run to completion option, whereas the choice of true option results in the blue curve, clearly showing that the watch is too fast, advancing one second every minute with respect to the simulation time.

Another example of a state machine where the run to completion option should be false will be presented later in the code generation section.

3.6   Debug mode and animation

The debug mode can always be left turned on. It must be set in the context of the **Top_SM_Diagram**. It checks the correctness of the model concerning the following points: two states of the same state machine do not have the same number. The max number of states set in the **StateActivation** block is equal or larger than the actual number of states.

41 ASM library

With the debug mode turned on, after model compilation, information about the states present in the model can be obtained in OML by calling the function **SM_Debug**. For example, in the elevator example, the following information can be obtained

```
> SM_Debug
ans = struct [
Top_SM_Diagram/buttons: struct [
NStates: 5
states:
{
[1,1] Top_SM_Diagram/buttons/unselected
[1,2] Top_SM_Diagram/buttons/selected
}
]
Top_SM_Diagram/status: struct [
NStates: 5
states:
{
[1,1] Top_SM_Diagram/status/atFloor
[1,2] Top_SM_Diagram/status/moving
}
]
]
>
```

The state changes can be animated during simulation. This can be done by placing the **Animate** block in the state machine diagram. One **Animate** block can be placed in each state machine diagram to animate the state changes realized by a change of color of the active block:
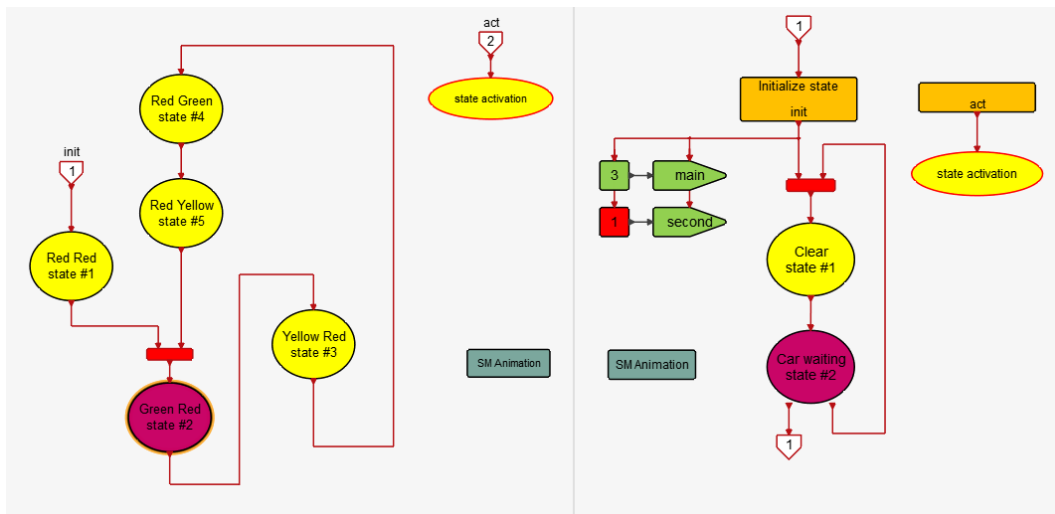
In most cases the simulation is too fast for the animation to be visible. To reduce the simulation speed, the Setup menu for Simulation Parameters can be used to set a non-zero value for the Real time scaling parameter
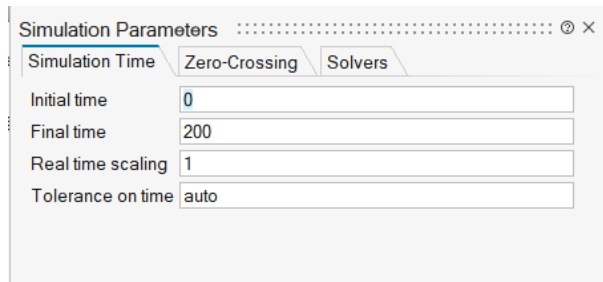
This value is set by default to 0, corresponding to no slowdown of the simulation. Non-zero values set the ratio between the simulation time and the real time; the value of 1 indicates one second per unit of simulation time. The larger this value, the more the simulation is slowed down.

43 ASM library

Chapter 4

Code Generation

Twin Activate code generators, with the same options and limitations, can be applied to models that include state machines. In fact, code generation is already applied to the content of the **Top_SM_Diagram** blocks for simulation since these super blocks are declared Atomic.

4.1 Creating a new block

Code generation for a super block creates a C code used for defining an equivalent new basic block. Using this basic block instead of the original super block is more efficient, both for model compilation and simulation. Moreover, the block can be distributed without the source code, and even licensed, for IP protection.

A simple state machine is used to illustrate how a new block can be constructed based on code generation. The block realizes a discrete-time zero-crossing detection mechanism, providing the direction of the crossings. More specifically, the output will be +1 if the crossing occurs in the upward direction, -1, in the downward direction, and 0 if no crossing occurs.

As long as the inputs are not zero, there is no ambiguity as to when crossings occur. But different definitions can be envisaged when the input can take zero value. For example, consider the following sequence of inputs: 2, 0, -3. Clearly a zero-crossing occurs in this case. The crossing can be detected when the input becomes 0 (i.e., when a positive input becomes zero), or it can be detected a step later when the input becomes -3 (so when the 0-input becomes negative), or in both cases (two detections). The definition considered here is as follows:

- If the input goes from a non-zero value to zero or a non-zero value of opposite sign, a detection is made.

- If the input goes from zero to a non-zero value, a detection is made if the previous value of the input has not had a non-zero value of opposite sign.

- Otherwise, no detection is made.

With this definition, a direct crossing, passing through zero is detected only once. This is not the only "reasonable" definition; other definitions can be used. This behavior is straightforward to code without a state machine, but here one is used for illustration purposes. The following is a possible implementation
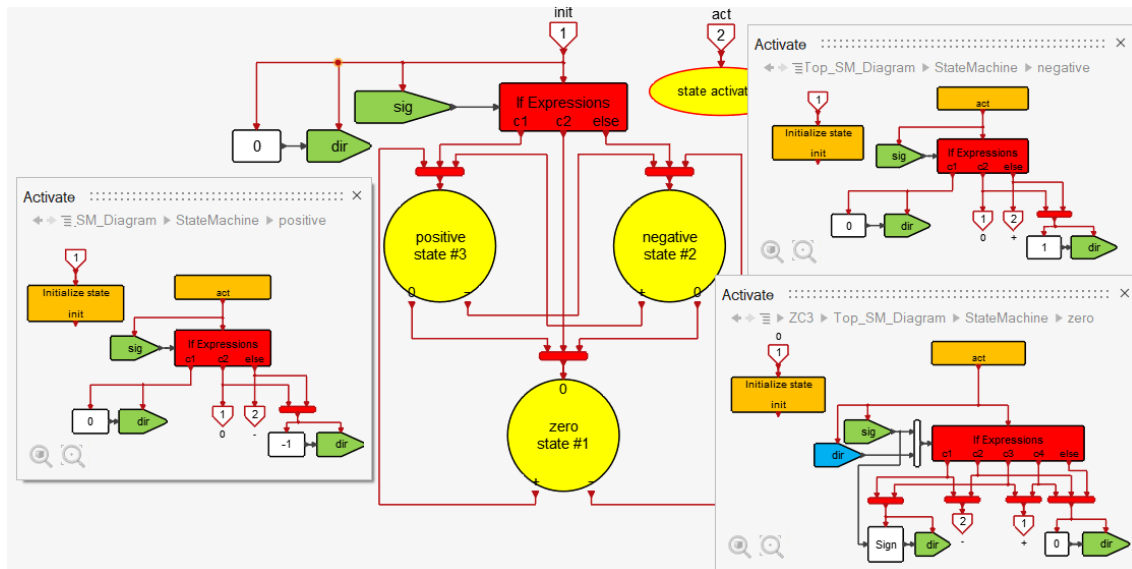
The 3 states of the state machine correspond to the sign of the latest input. The main logic of the detection is implemented by the **IfExpressions** block in the zero state block, parameterized as follows
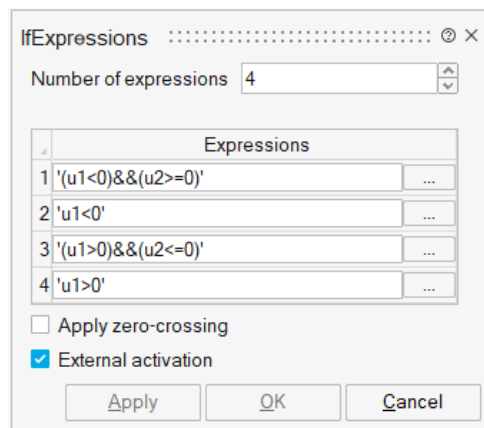
Using the C code generation option in the Tools menu, the **Top_SM_Diagram** can be converted into a CCustomBlock, and then turned into a basic block. This block can be used for example to build an alternative to the Activate **EdgeTrigger** block as follows
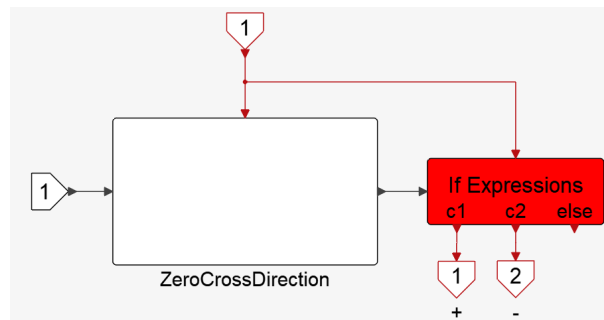

Note that the zero-crossing detection generates events that are synchronous with the input signal. There is no delay in detecting the crossings.

4.2 Creating an independent executable

State machines are often used in the construction of controllers. The inline code generator can then be used to generate embedded code for implementing controllers. The code produced by inline code generator is particularly amenable for such usages. It can also be used for creating compact and efficient FMUs since it has no or little dependency on external libraries.[1]

To illustrate the application of inline code generation to state machine models, the stopwatch demo is used to generate a Windows executable providing the functionalities of a stopwatch. The model emulates the operation of a classical stopwatch operated by two buttons: start/stop and split/reset represented respectively by the right and left arrow keys of the keyboard, during simulation. The Display block is used for the stopwatch display, indicating the time in seconds and, tenth or hundredths of seconds. The choice between tenth and hundredth is made by the unique block parameter N.

Two button stopwatch (left and right arrow keys)
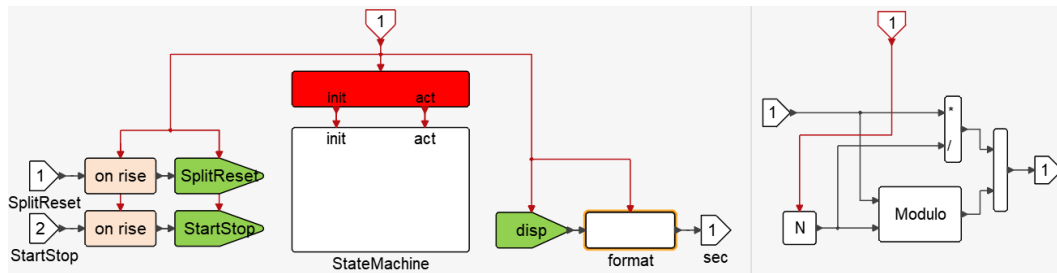start/stop : ->
split/reset : <-
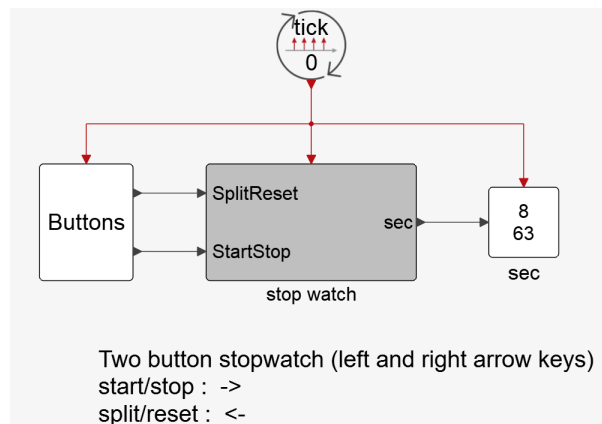
The standard operation of the stopwatch consists in using the start/stop button to start the stopwatch and then stop it. The split/reset button then resets the time to zero. If the start/stop button is applied again instead of the split/reset, the stopwatch restarts from the time it was stopped at. Its subsequent application stops the time again. This start-stop operation can be repeated indefinitely.

If after the start of the stopwatch, the split/reset button is used, the display freezes, indicating the lap time, but the watch continues advancing the time internally. Subsequent applications of the split/reset button display the following lap times. The start/stop button then stops the stopwatch, indicating the final time. The stopwatch can then be reset using the split/reset button.

The advancement of time can be modeled either using the **SM_Time** block, which provides the simulation time or counting the number of activations. The latter can be used if the run to completion option is set to false. For that to work, the block activations must be assumed to be periodic with a know period. Such a stopwatch implementation is available in the demos; the model's name is chrono.

External libraries such as Lapack may be required (and included automatically) when the model contains certain matrix operations.

---

[1]External libraries such as Lapack may be required (and included automatically) when the model contains certain matrix operations.

47 ASM library

The *StartStop* and *SplitReset* signals become true when the corresponding buttons are pressed. The *disp* signal contains the time to display in seconds. The format super block (content shown on the right) converts the value of time to two values: number of seconds and hundredths (or tenths depending on the value N) of seconds.

The state machine is realized by four states as shown below
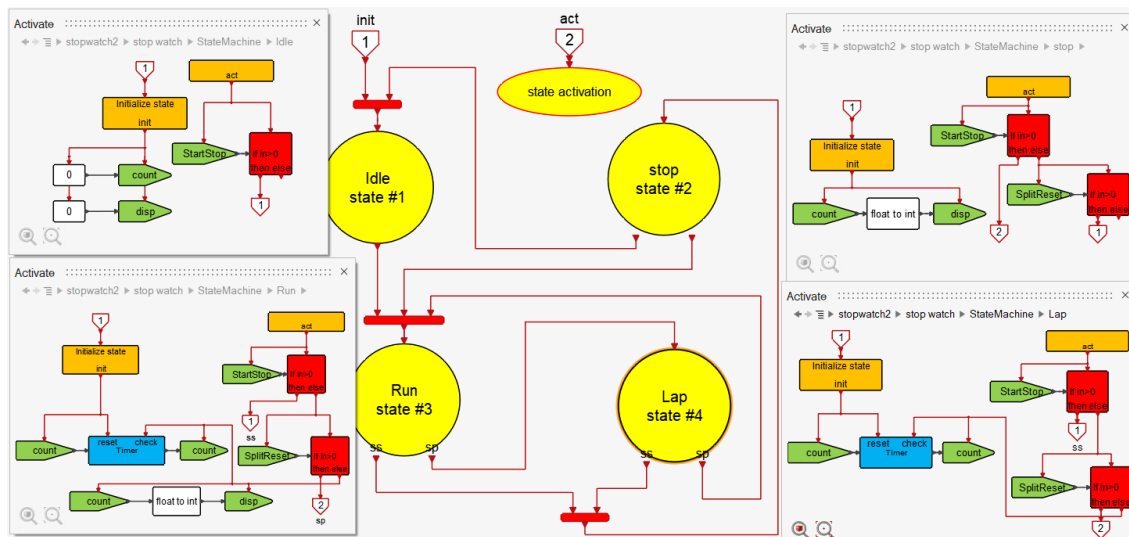
The "float to integer" super block converts the *count* signal, which has data type double representing the number of seconds, to an integer after multiplying it by N, to be used for display. The operations of different states are clear from their contents.

Inline code generator can be used to generate standalone code by selecting the "stop watch" block and invoking the OML function vssCompileToStandalone. This function generates a folder in the model temp directory which includes the main C code, body.c and the corresponding include file, body.h, associated with the content of the "stop watch" block. These files provide the basic functions used for all codes generated for different targets. In this case, their functions are called in the file main.c, a generic main function that produces an executable where the inputs are read on the keyboard via scanf and the outputs are printed in the shell. This main file, also generated by `vssCompileToStandalone`, shows how the body functions should be used; the content of main.c should be considered a prototype and needs to be adapted to the specific application.

In this case, the main.c is adapted by replacing the scanf instructions with non-blocking `GetAsyncKeyState` functions to obtain the code of the pressed key on the keyboard. The modified and original files are included in the demo folder for comparison. The vssCompileToStandalone function also generates makefiles to facilitate compilation and creation of the exe file under Windows with both MS Visual C compiler and TCC (embedded in **Twin Activate**). Running the main.exe file, continuously prints the

display of the stopwatch to the shell (seconds followed by tenths or hundredths of seconds). The left and right arrow keys on the keyboard are used to operate the stopwatch as in the simulation model:

```
0        0
0        0
0        0
0        0
0        1
0        2
0        3
0        4
0        5
0        6
0        7
0        8
0        9
0       10
0       11
0       12
```

Chapter 5

Pitfalls and common mistakes

Twin Activate is not a specialized tool for modeling exclusively state machines. The ASM library blocks provide the basic elements for the construction of state machines, but their usage must follow strict rules to avoid jeopardizing the integrity of the state machine being constructed. The predefined properties of the ASM library blocks introduced to ensure proper implementation of the state machines should not be modified. For example, the **Top_SM_Diagram** block has Atomic property. User should not modify this property[2]. Some of the ASM library blocks have invisible parameters for implementing the internal machinery of state machines; they should not be altered. State blocks do not have signal input and output ports. They should not be added by the user. And activation inputs and outputs must be reserved for use for state transitions, as has been explained throughout this document.

Even by following scrupulously these recommendations, just like programming in any language, the result is not always what is expected at the first try. Errors are encountered and must be fixed. The synchronization problems are the most common types of error encountered when constructing state machines. To understand these errors, and more importantly how to avoid them, a few typical situations are presented and analyzed here.

5.1 Algebraic loop errors

This error, which is not specific to state machine modeling, implies that the order of executions cannot be determined for a set of blocks. This means that the computation of a signal requires the value of the same signal. The following diagram contains an algebraic loop:
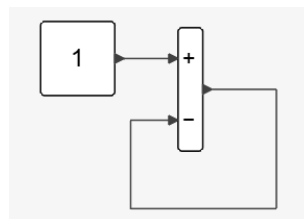

If the output of the sum is denoted X, then this diagram states that $X = 1 - X$. Even if in this simple

---

[2]Atomic option should not be removed unless the user knows exactly what he or she is doing. The Atomic property isolates the content of the block from its environment and is in general required for the proper operation of the run-to-completion functionality. It also avoids possible signal name conflicts with signals used outside the block. But in case the state machine is used to implement hybrid systems (continuous-time dynamics), it may be necessary to remove this option.

case the equation can be "solved" for X, **Twin Activate** does not attempt to solve numerically the algebraic equations (it does only if a LoopBreaker block is explicitly placed in the loop). The reason is that in most cases, algebraic loops correspond to modeling errors and need to be investigated, especially in state machines.

In discrete-time systems, loops are mainly used to compute the value of a signal as a function of its previous value:
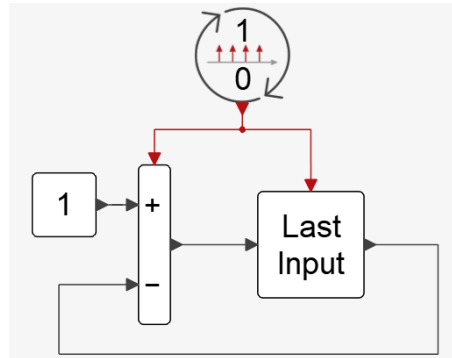
In this case there is no algebraic loop present in the model. The corresponding equation is $X(t_k) = 1 - X(t_{k-1})$. The LastInput block introduces a one-step delay in the loop, thus breaking it.

The presence of loops of (data) signals is not the only way to create algebraic loops in a model. For example, in the following model
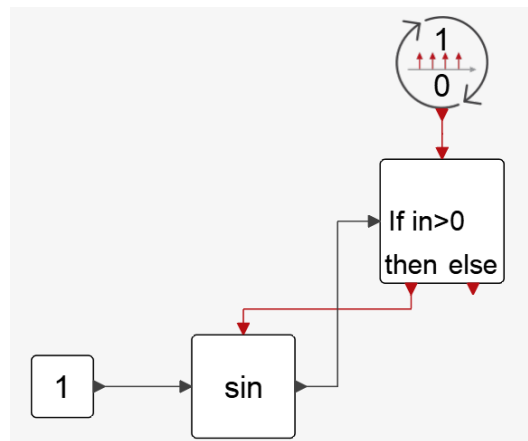
denoting the output signal of the sine block, X, the implemented equation is

if $X(t_k) > 0$ then $X(t_k) = \sin(1)$

This is another type of algebraic loop, but fundamentally the problem is the same: computation of $X(t_k)$ would require the value of $X(t_k)$.

In the case of state machines, such algebraic loops are easily created by mistake. And they can be difficult to isolate because the model is spread over multiple super blocks and most connections are not visible (they are established via set and get signals). So, it is important to be aware of common mistakes users can make that result in algebraic loops.

For an algebraic loop to be present, all the blocks of the loop must be synchronously activated. If the loop occurs fully inside a BlockState, then it is in general well isolated and easy to identify. The difficult loops are the ones that occur spread over multiple BlockState blocks. To understand how this can

happen, it is important to take a closer look at the event transition mechanism.

Consider the following two states and the transition from the left state to the right state:
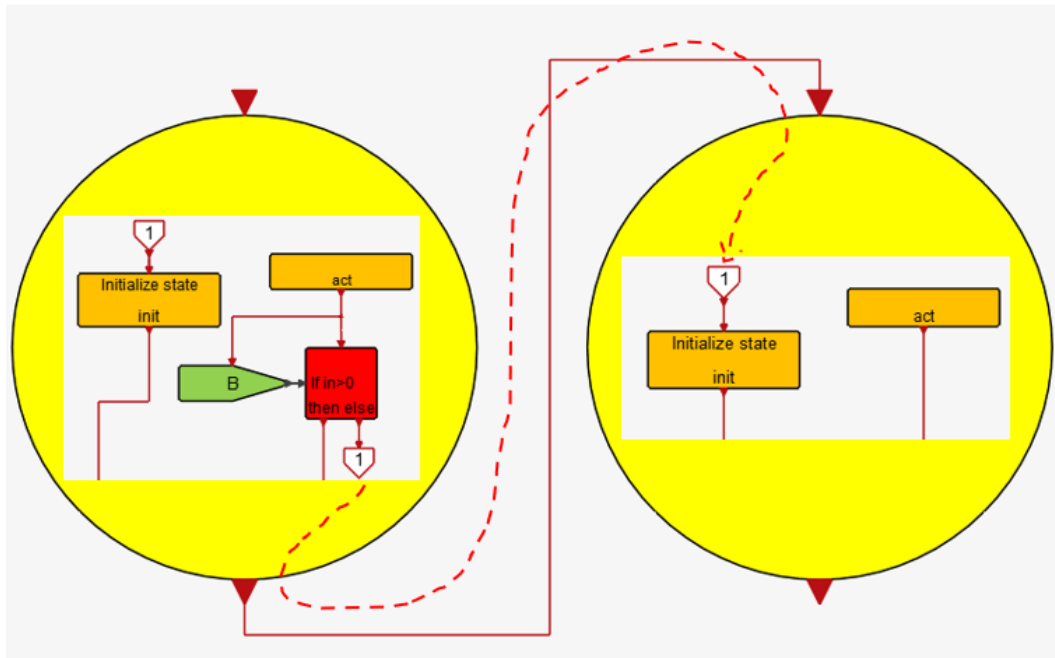
The left state exits when the state is activated, and the signal B is not > 0. The activation event goes through the else port of the **IfThenElse** block, leaves the left state and enters the right state, into its Initialize block, activating the right state.[1] This means that the last activation event in the left state corresponds to the initialization event in the right state. So, the exit event of a state is also the initialization event of the next state. This implies that the exit actions of the current state and the initialization actions of the following state are executed synchronously. This can lead to algebraic loops spread over the two states. For example, if the exit action includes

$B = C$

for some signals B and C, and the initialization action,

$C = B + 1$

then an algebraic loop is created.

Note also that if the same signal is set twice, in the exit action and in the initialization action, to two different values, the result is undetermined.[2] Since the event is interpreted as an event exiting the state and entering the next state to initialize it, one may expect that the exit actions precede the initialization actions, but since they are executed synchronously, depending on the data dependencies, an exit action may very well execute after an initialization action.

So, the important point to retain is that during a state transition, the exit event is the initialization event of the following state. The initialization makes the next state active, which means that subsequent activation signals are received on the act port of the **Action** block of the new state. In general, the initialization event and the activation events of a state are not synchronous. There is however one

---

[1]This description may be misleading because it gives the impression than the event reaches the right state after a delay. But in fact, it is the same event, and all blocks activated by it are synchronously activated.

[2]But no error is generated in this case because such a behavior is not strictly speaking an error. It is consistent with **Twin Activate** semantics.

exception: when a transition from a state to the same state occurs. This special case may, in some situations, lead to an algebraic loop if the state is not correctly constructed.

5.2   Infinite loops

With the option run-to-completion set to true, a state machine may get into an infinite loop where time does not advance. The run-to-completion option implies that if an activation causes a state transition, then it is repeated from the source (the activation of the **Top_SM_Diagram** block). This means that until the transitions continue to happen, the activations are repeated with time not advancing. This series of activations are not synchronous but occur at the same time. A valid model without the run-to-completion option may fall into an infinite loop with this option. Consider the following simple example: a state machine with one state where the state re-initializes itself at each activation (no other action).
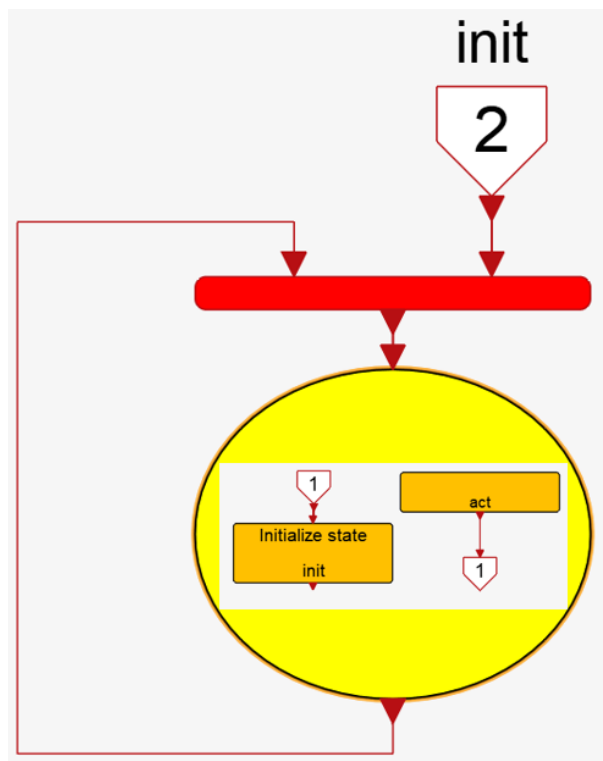
Without the run-to-completion option, a state transition occurs once at each activation of the **Top_SM_Diagram** block. With it, each transition leads to a new activation and a new transition, etc., resulting in an infinite loop at the first activation of the **Top_SM_Diagram** block. Note that there is a fundamental difference between the temporal dynamics of the state machines with and without the run-to-completion option. Without the option, state machines in **Top_SM_Diagram** block are activated once per activation of the block. So, for example if the block is activated by a periodic clock with known period, time delays can be realized by counting the number of events. Such a construction cannot be used with the option on. The state machines may be activated an unknown number of times at any activation of the block, so event counting cannot be used to measure a time period. Special timer blocks are provided in ASM to use for measuring time, and in particular to implement timers.

5.3   Unintended block activations inside inactive states

The fundamental property of a basic state machine is that only one state is active at any time, and the ASM blocks are designed to impose this property. But in some cases, this property may be violated by erroneous usage of some non ASM blocks. **Twin Activate** cannot detect such usages because they are not **Twin Activate** modeling errors.

One way a block can be activated inside an inactive state (**StateBlock**), is through activation inheritance. When an **Twin Activate** block is not explicitly activated, it inherits its activation through its input signals, or more specifically it inherits the activations of the block that produces these input signals. This happens whether the input signals are connected by links or through communication via set and get blocks.

To avoid unintentional activations through inheritance inside states, the ASM library provides activated set and get blocks (set and get blocks requiring explicit activations to operate). These blocks should systematically be used instead of standard set and get signal blocks to communicate signals between states and, between states and the block inputs and outputs. To prevent unintended activation via inheritance by link, the **StateBlock** blocks are void of signal input and output port.[1] This prevents a signal link to cross the boundaries of a state and carry its activation into an inactive block.

Another way a block can be activated inside an inactive state is by using blocks generating delayed asynchronous events, such as the EventDelay block. These blocks, when activated, program an event to be fired after a delay. But by the time the delay is over, the state may no longer be active.

Always active blocks should also be avoided in ASM based state machines. The ASM library is primary designed for discrete-time state machines. Some hybrid system construction can be carried out with ASM blocks, but there are many limitations and fine tuning of simulation parameters may be required. Additional extension will be made available in the future for properly modeling hybrid systems.

5.4   Improper state exits

Another common mistake is the generation of multiple synchronous exit events out of a state. A state should only be exited through one port at any time, otherwise it may initialize more than one state.

The exit out of a state is decided inside the state. The exit event is a normal activation event of the state (the last one), which the block decides to redirect to an output port. This is different from the common way state machines define exit conditions: they define exit conditions outside the state, on the transition links. This does not fundamentally change the operation of the state, but it imposes the responsibility of assuring that only at most one exit event can be produced on the state.

The uniqueness of the exit events can be imposed by the systematic use of a single **IfThenElse**, **SwitchCase** or **IfExpressions** block to produce all exit events. These block by construction and structurally redirect their input activation to only one of their outputs. The usage of a single block to generate all exit event is recommended even if it is not always necessary. Consider for example a state that may exit through two different ports: first port when a signal B is larger than 1, second when it is smaller than -1; otherwise, no exit occurs. The exit conditions can be implemented as follows

---

[1]Regular signals ports can be placed on the **StateBlock** block; this block, which is an Activate super block, cannot prohibit such construction. But the addition of such ports is strongly discouraged.
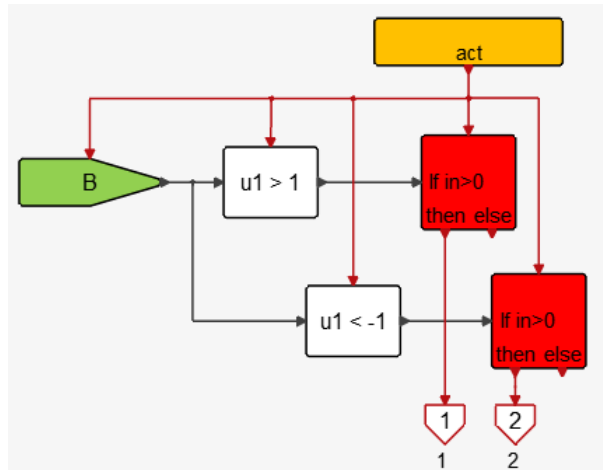
This construction is not incorrect because only one of the ports can receive an event at any time (B cannot be > 1 and < -1 at the same time). But this construction is fragile. If later a decision is made to parameterize the model and use parameters c and d in place of -1 and 1, wrong parameter values (for example exchanging them by mistake) can lead to two synchronous exit events. Such problems in a model are difficult to identify and may even go unnoticed. This construction is not optimal performance-wise either: even if $B > 1$, the test $B < -1$ is still done. The analogy with OML programming can be made with the following code[1]

if $B > 1$, exit = 1; end

if $B < -1$, exit = 2; end

A better code would be

if $B > 1$, exit = 1; elseif $B < -1$, exit = 2; end

The counterpart of this code would be the following **Twin Activate** implementation
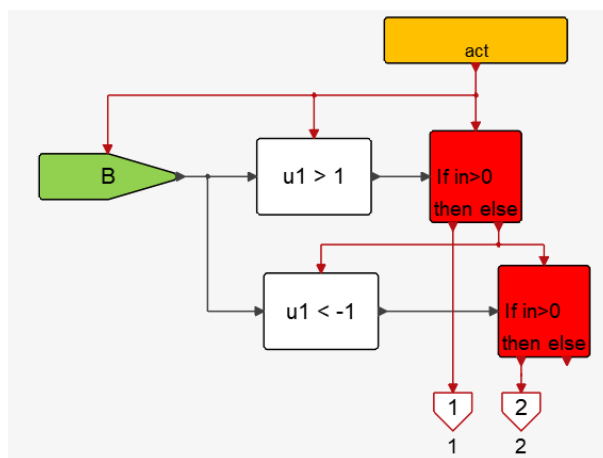
Not only this implementation is more efficient, but it is also structurally sounder and safer.

Alternatively, the **IfExpressions** block can be used with similar performance since this block is a super block including cascaded **IfThenElse** blocks disposed as in the previous construction:[2]

---

[1]Here it is assumed that the variable exit has already a value at this point, for example 0.

[2]The drawback is that the expressions are not visible on the diagram. They can be viewed only by opening the block GUI.
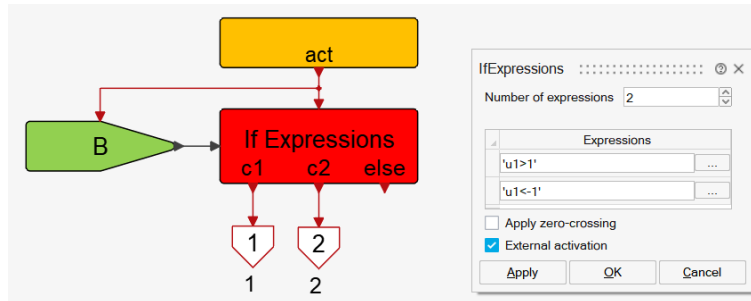
Note also that the use of the cascaded construction is particularly useful for implementing priorities when multiple exit conditions can be true simultaneously.

Another type of exit that should not be used is shown below
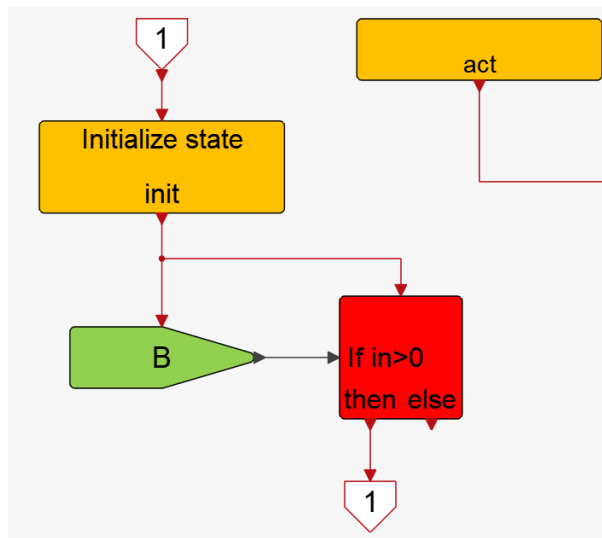
It may seem interesting to leave a state right-away at initialization when a condition is true, even before activating the state. But this could lead to unexpected behavior. The next state will be initialized synchronously with this state, so not only the initialization actions may be in conflict, but even the setting of the state is in conflict, so there is no guarantee as to which state becomes active following the double initialization.

The exit can be realized by the first activation of the block after initialization. This does not introduce a time delay if the run-to-completion mode is on (or only a single step delay if it is not).

5.5 Debug mode error messages

A number of error messages are generated by the ASM blocks. The debug mode checks for common mistakes such as improper numbering of the states (in particular using twice the same number) or not setting the max number of states correctly. These messages are usually self-explanatory.

Note however that the debug mode must be set to 'on' (or 'off', even though this not recommended): the line initializing the debug mode in **Top_SM_Diagram**:

```
SM_Debug('on');
```

should not be removed. This call to **SM_Debug** initializes the state of the debugger. If the state is not

57 ASM library

initialized, then the state corresponding to previously simulated model can be mixed up with that of the current model, resulting in incorrect error messages such as

Error: Multiple **StateActivation** blocks seem present in the same state machine

This error should only be generated if two **StateActivation** blocks are placed in the same state machine diagram by mistake.

Chapter 6

Example: Vehicle with adaptvie cruise control and automatic transmission

A very simplified automotive application is considered where the adaptive cruise control and automatic transmission mechanisms are modeled as parallel running state machines. A simple model of the vehicle is also used to test the controller behavior.
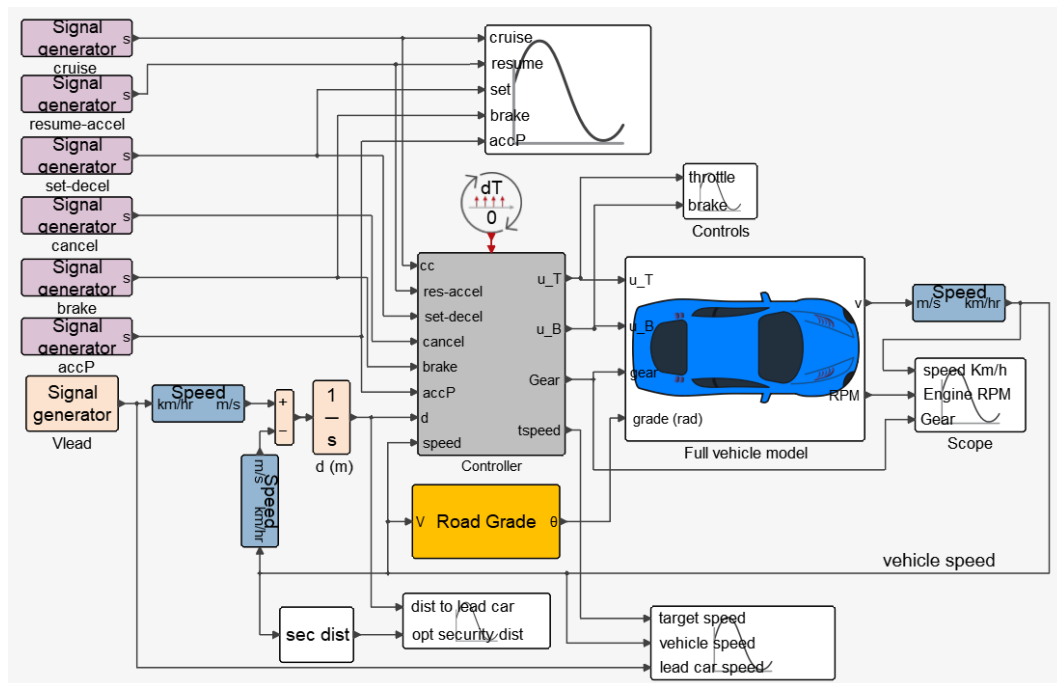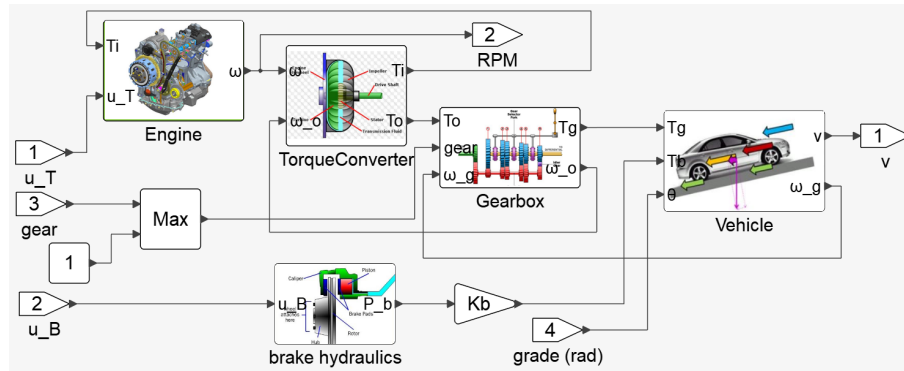
The content of the **Top_SM_Diagram** (controller) shows the separation between the cruise control funcitonality and the automatic transmission:
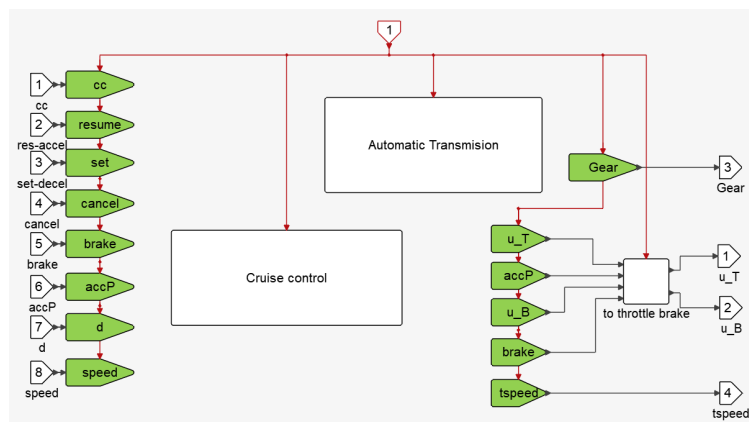
This is done by introducing an additional level of hierarchy for better visual organisation.

In this implementation, the run-to-completion option is not used, and the only exposed parameter is the clock period dt:
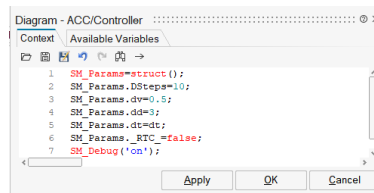
6.1 Adaptive cruise control (ACC)

The ACC controls the speed of the vehicle by acting both on the throttle and the brake. The objective of the controller is both to follow a target speed chosen by the conductor but also maintain the vehicle at a safe distance of the leading vehicle.

The ACC state machine not only implements the controller but also the user interactions for activating and disactivating it, setting the target speed, etc., operated via 4 buttons: Cruise, Set/Decel, Resume/Accel and Cancel. The ACC is turned on and off using the Cruise button. The ACC can be made active using the Set/Decel button. In that case the cruising speed (target speed) is set to the current speed. If the ACC is already active, the Set/Decel button can be used to reduce the target speed. The target speed is reduced by a value dv when the button is pressed. The Resume/Accel button in

this case can be used to increase the target speed by dv. Note however that if the accelerator pedal is depressed when the ACC is active, then the Set/Decel button sets the target speed to the current speed and the Resume/Accel button does nothing.

The following operations deactivate the ACC: depressing the brake pedal, using the Cancel button, and turning off the CC button. In the first two cases, the target speed is preserved in memory, and used if the ACC is reactivated using the Resume/Accel button.

The ACC control of the distance to the leading vehicle cannot be parameterized by the conductor. The following formula is used to evaluate an optimal security distance: $d_{\sec} = \tau v + d_0$ where v represents the vehicle's speed and the parameters τ and d_0 are fixed. The ACC tries to maintain the vehicle within this distance of the leading vehicle. In this simple model, the emregency situations, such hard braking of the car ahead, are not taken into account. The model of a real ACC is a lot more complex, in particular having to deal with tailure situations.
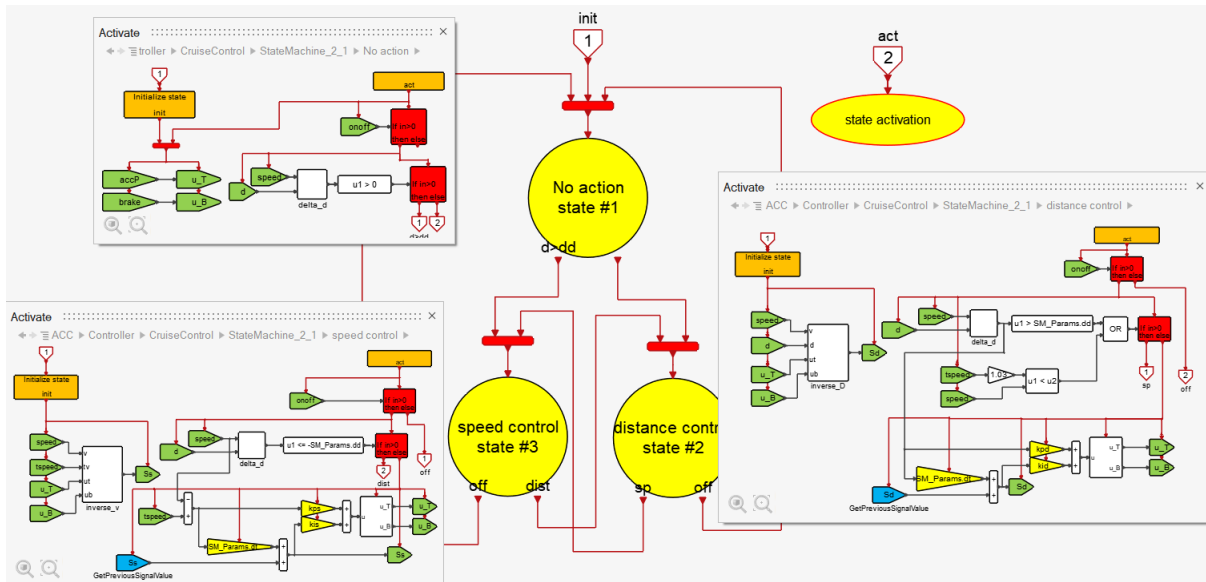
The ACC is implemented as two parallel state machines
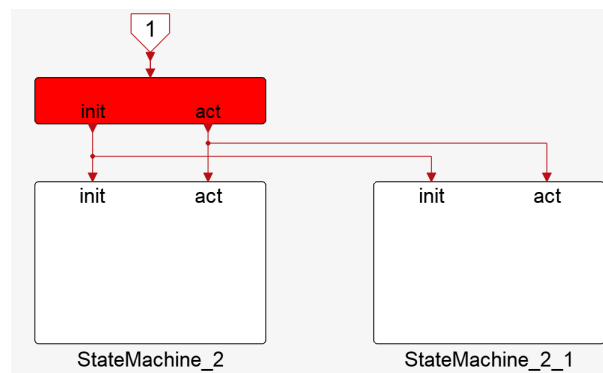
One handles the user interactions for setting the ACC parameters
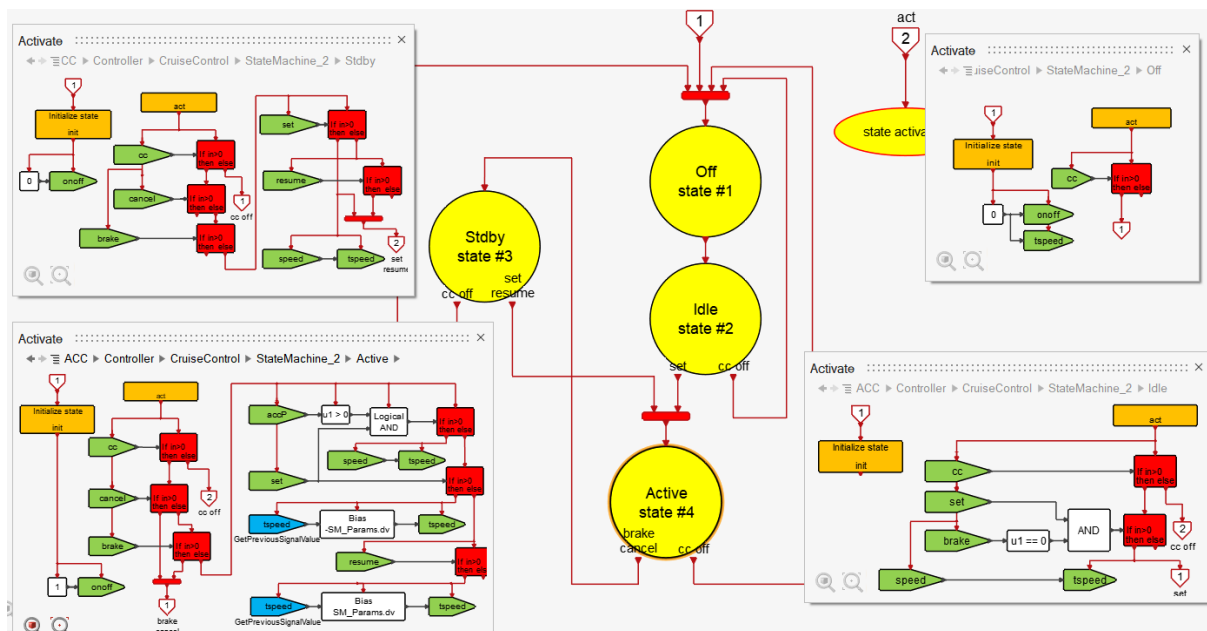
The other provides control signals for the throttle and the brake

When the ACC is active, the state machine can either be in the speed control state or the distance control state. Each controller is implemented by a PID. The states of the PIDs are initialized when the corresponding state is entered to guarantee smooth transitions.

6.2 Automatic transmission

The automatic transmission controller is also implemented using two parallel state machines
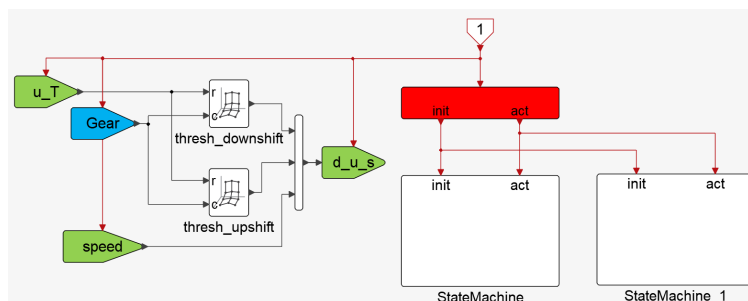

The gears are shifted based on thresholds provided by lookup tables; they depending on the vehicle speed and the throttle position. The decision to shift up or down however is not applied immediately. They are realized only after a period of time if the threshold crossing remains valid. If the threshold is crossed back during this period, the shift is abandoned:
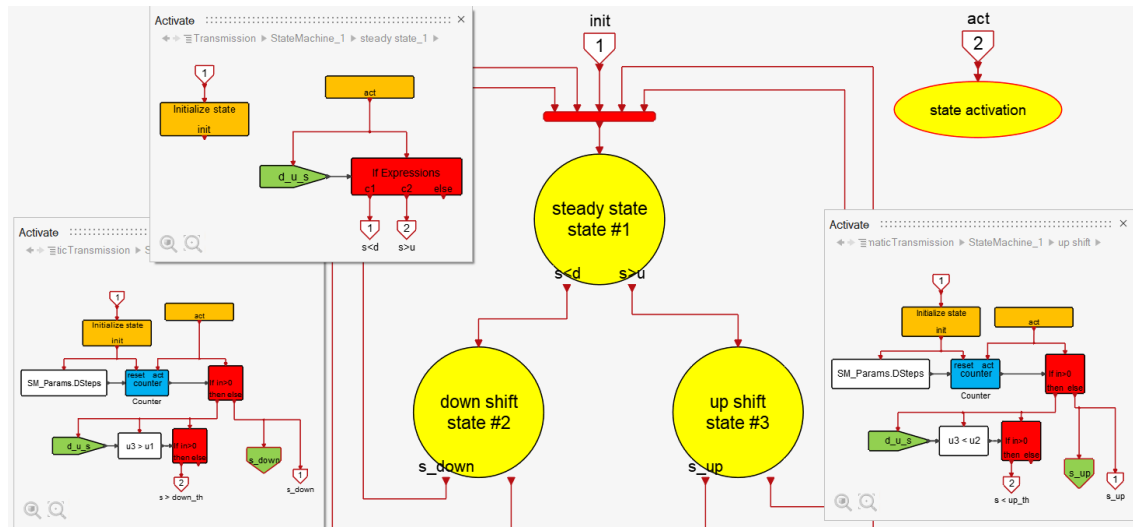
The shift up and down events changes the gear signal in the parallel state machine
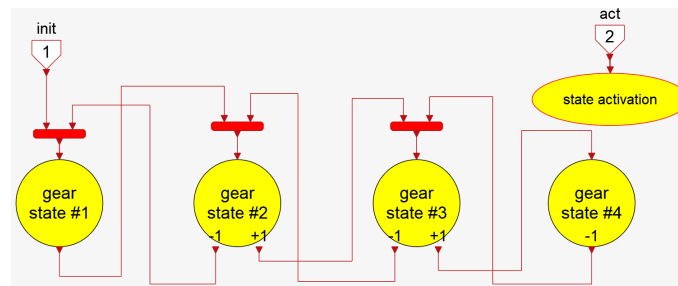
For example the content of the state corresponding to the second gear is:
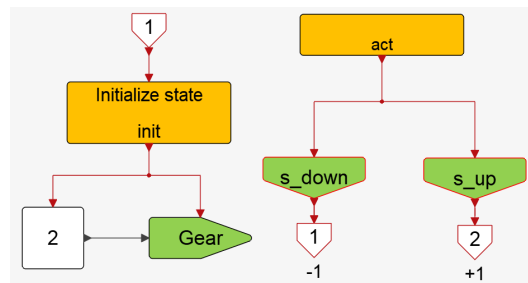


6.3 Simulation results

The above controllers are tested using a vehicle model presented originally in [3]. The resulting model, `ACC`, is available in the library demos.

The considered scenario is as follows: the vehicle starts from stand still. The accelerator pedal is pressed and the ACC is activated. Then at $t = 10$sec (speed close to 85Km/h), the set button is used to fix the target speed. From there, the vehicle is automatically controlled. The lead car being far ahead, the state machine enters the speed regulation state. The speed is regulated around 85km/h until its distance with the lead car falls below the security distance. Then it switches to the distance control state, etc.

The conductor brakes for a short period at $t = 140$sec. This action de-activates the ACC. The vehicle

63 ASM library

speed then drops until the resume button is pressed at $t = 180$sec. The target speed is thus reset to its original value (85km/h), and the vehicle speeds up to reach this value.
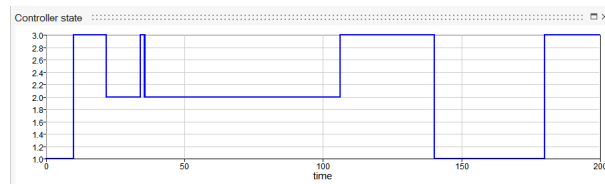
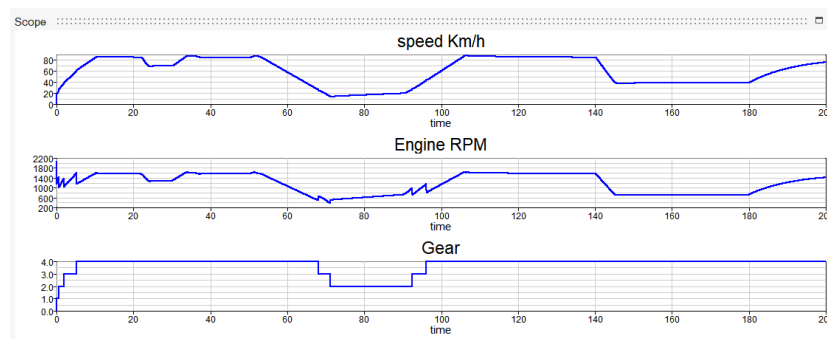The user input and sensor signals are shown below
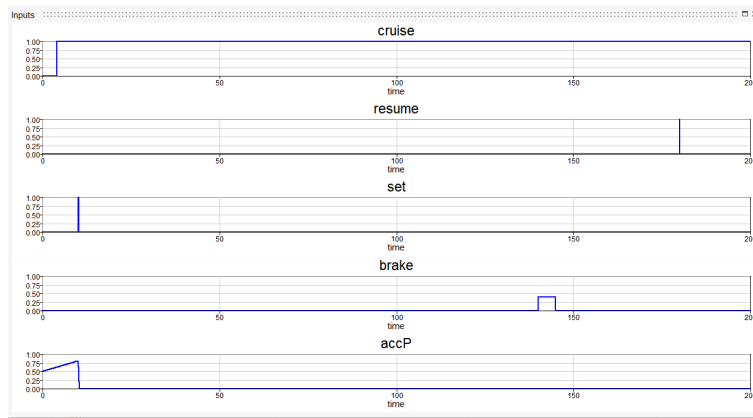
The vehicles speeds and relative distances are as follows
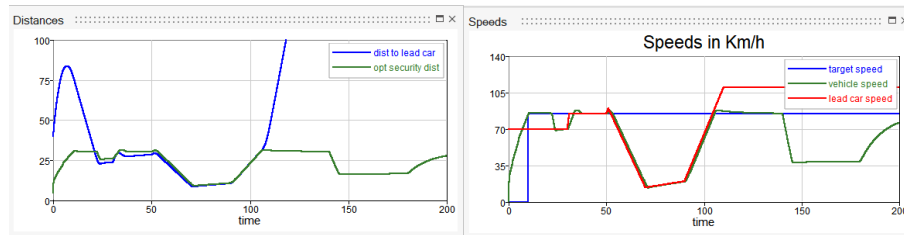
The changes in the controller states of the ACC can be seen below

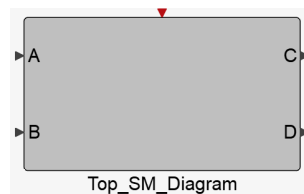The changes in the vehicle speed, engine RPM and gears are as follows

Chapter 7

Main Library Blocks

Additional information about the main ASM library blocks, and their usages, are provided in this section.

7.1   Top_SM_Diagram block

This (super) block provides a prototype of state machines. It is the starting point for the construction of new state machines. It contains a "typical" construction, which can be adapted to the specific problem at hand. Its content is shown below
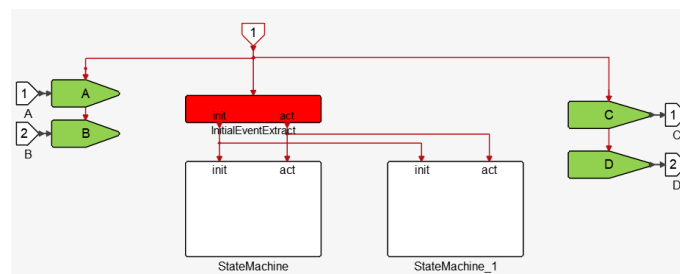
By default, two parallel state machines are assumed. Two inputs and outputs are also named for use inside the state machines. StateMachines are plain super blocks but contain state machine prototype blocks (to be adapted to the application).

Top_SM_Diagram is an Atomic super block. This feature is important for isolating its content, so that it behaves as an independent block, especially when the run-to-completion option is used. This also eliminates the risk of signal name conflicts between the inside and the outside of **Top_SM_Diagram**.

Parameters can be added to the mask of this block for parameterizing the State machines. The parameters should then be included in the **SM_Params** structure to be accessible everywhere in the state machine (in particular inside the states). This should be done in the Context of **Top_SM_Diagram**. For example, if a parameter A is to be added as parameter, then A should be added to the block mask, and its Context modified as follows:

```
SM_Params=struct;
SM_Params._RTC_=true;
SM_Debug('on');
SM_Params.A = A;
```

The first 3 lines are present in the Context by default (more default values may be added in the future). The last line is added to place the value of A in the structure **SM_Params**, which can be used anywhere inside the state machines.

Multiple **Top_SM_Diagram** blocks can be used in the same **Twin Activate** model. They can be connected through their ports, and run with the same clock, or not. There is no risk of unexpected interactions since they are atomic. Multiple blocks containing state machines actually represent another way of implementing parallel state machines. Note however that unlike parallel state machines inside the same **Top_SM_Diagram** block, state machines in different blocks can only communicate through signal ports.

7.2 StateBlock and StateActivation blocks

These two blocks go together. They are the basic components for defining a state machine, which is obtained by placing one **StateActivation**, and one or more **StateBlock** blocks in the same diagram. The **StateActivation** block dispatches its activation to one of the **StateBlock** blocks defined by the value of a hidden state variable. The value of this variable is set inside the states, when the states are initialized.

The (visible) parameters of the StateBlock are

The state number and name must be chosen to be unique (for each state machine). The number is in the range $1 \cdots NStates$, $NStates$ is the maximum number of states defined as parameter of the **StateActivation** block.

Ideally, $NStates$ should be set to the actual number of states used in the diagram; if it is chosen too

large, in some cases it may make the generated code more complex. It should of course not be set to a value less than the number of states. If it is, or if two states have given the same number in a diagram, an error is reported by the compiler (when debug mode is on).

Note that to make the model more readable, when a name of state to display is defined, the name of the block is automatically changed to the same name.

7.3   InitialEventExtract block
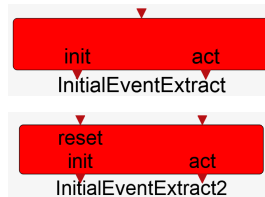
These blocks are used to separate the first event of an activation signal from the rest. The first event exists the init port and is usually used for initialization of the state machine. If the block is reset, then the first event following the reset is directed to the init port, and subsequent events to the act port, until possibly a new reset occurs.

This block has no parameter.

7.4   Initialize and Action blocks

These blocks are placed inside the **StateBlock**. The **Action** block provides the events dispatched by the **StateActivation** block corresponding to the state. The **Initialize** block must be activated by events entering the state (initializing the state). This block sets the hidden state variable so that the state containing this block becomes the active state. This implies in particular that all future events will be dispatched to this state and made available by the Action block, until another state becomes initialized.

A state contains only one **Action** block but may contain more than one **Initialize** block. Depending on the way the state is "entered", different initialization actions may be undertaken. The Init output of the **Initialize** block is just a replicate of its input activation and is provided in this form for convenience.

These blocks have no parameters.

7.5   Communication blocks

In **Twin Activate**, Set and Get blocks are used in models as alternative to link connections. In some situations, for example when two blocks to be connected are far apart, perhaps in different diagrams, it is more convenient to establish a virtual connection between their ports instead of creating a link, possibly having to traverse multiple super block players. The association between the Set and Get blocks is made using a common signal name.

67 ASM library
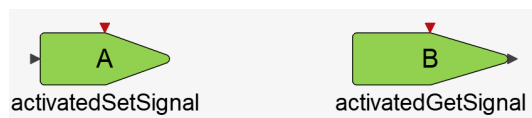
Regular signal

The blocks for communicating regular signal blocks in the ASM library are based on **SetSignal** and **GetSignal** blocks, but they must be activated to operate. The **ActivatedSetSignal** block sets the value of the corresponding named signal (state machine variable) to its input value only when it is activated. Similarly, the **ActivatedGetSignal** provides the value variable only when it is activated. These blocks should be used to communicate variables between different states and state machines.
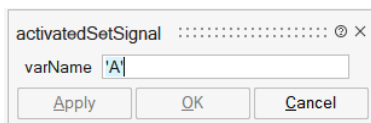

The block parameters are the name of the signals, i.e., the name of the state machine variables. These names have global scope. Note however that since the **Top_SM_Diagram** block is Atomic, the global scope is limited to the interior of this block. There is no risk of name clash with signals defined outside this block or inside other blocks of the kind.

The use of the **SetSignal** and **GetSignal** blocks of the Base library should be avoided in state machines. These blocks, just as regular links, can result in the activation of the block receiving the signal by inheritance. This may activate blocks inside a non-active state, breaking the fundamental property that in a state machine, only one state can be active at any time. More generally, it is a good practice to explicitly control the activation of all the blocks used inside a StateBlock.

Previous value of regular signals
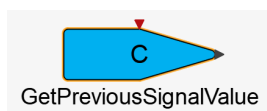
The **GetPreviousSignalValue** block, when activated by an event, provides the value of the named signal set previously by a different or the previous instance of the same event. This block defines an extended state; it uses a LastInput block inside to implement it.
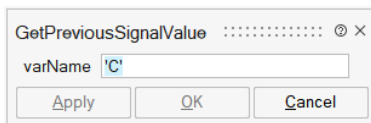

The explicit use of the "previous" operator is required to avoid "algebraic" loops in certain situations. Unlike imperative programming languages, such as OML, where for example a variable can be incremented using a code like

$C = C + 1;$

in declarative synchronous languages, such an operation is not allowed. The reason is that both C's refer to the value of the variable C at the time of activation of the instruction. So, this instruction needs the value of C to determine the value of C, which is a contradiction.

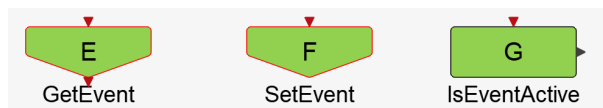The **Twin Activate** version of this instruction:
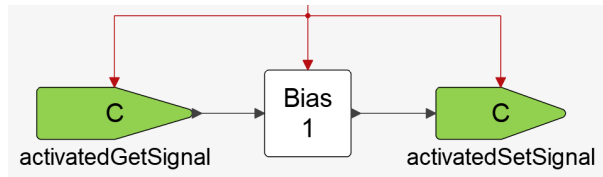
is flagged as an algebraic loop error by the compiler. A correct implementation can be as follows
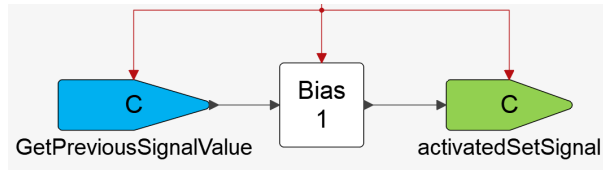

This is to be compared for example with the Modelica statement

$$C := pre(C) + 1;$$

7.6   Event signal

GetEvent and **SetEvent** blocks are similarly based on Base library blocks **GetActivationSignal** and **SetActivationSignal** blocks, and are used to communicate activation signals (events). The **GetActivationSignal** provides an event only if the named event occurs and the block is activated, synchronously. The **IsEventActive** block is a slight variation on this block, presenting the result as a regular signal: when activated, it outputs 1 if the named event is activated, 0 if not.
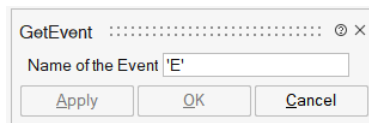

The only block parameter of this block is the name of the event. It has global scope.

## 7.7   Time and Timer blocks
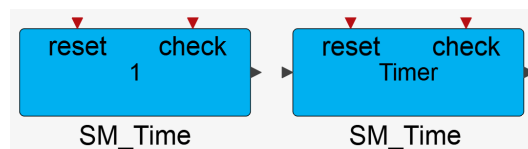
The **SM_Time** block provides a time output running according to the simulation time. The time value can be set using a reset event to either a block parameter or the value of the input. This is a block option. Another option determines if the time runs in the direction of simulation time (forward) or backward:

69 ASM library
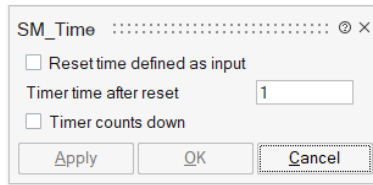
The output of the block provides the time only when the block is activated through its check activation port.

This time block can be used in particular to create timers. However, for convenience, several timer blocks are also provided in the library:
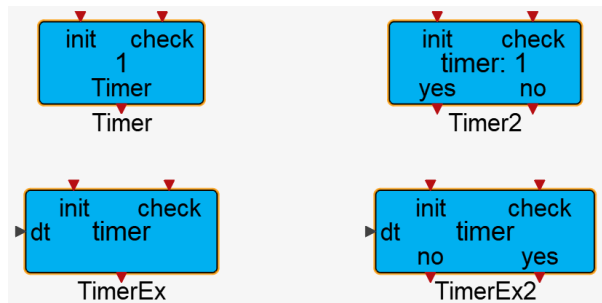
They all work on the same principle: an event on the init activation port starts the timer. An event on its check activation port is then directed to the output if the specified time-period, called timeout, is over. So, the event on the check port "checks" if the timeout period has run out or not.

The timeout can either be defined as a block parameter or via an input to the block. The timer may also provide a second output activation port, the complement of the first one, providing an event if the timeout period is not over when the block is checked.

The Timer and Timer2 blocks take as parameter the timeout period. The TimerEx and TimerEx2 blocks do not have any parameters.

7.8   Counter block

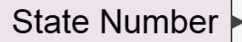This block can be used to count the number of occurrences of an event:
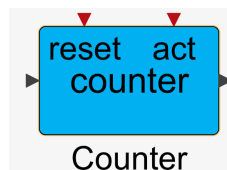
Note that if act is activated by a state activation event, and the run-to-completion option is true, the evolution of the counter value may not be predictible as a function of time.

The **Counter** can count both up and down depending on the sign of the step parameter.

7.9   StateNumber block

This block, once placed in a state machine, provides the number of the state which is active. Mainly used for debugging, this block is activated by inheritance; it does not require to be explicitly activated.

State Number ▶

This block has no parameter.

7.10   Animate block

This block, once placed in a state machine, provides animation. The active block is indicated by changing its background color. Note that for the animation to be visible, the simulation may have to be considerably slowed down. For the block to operate, the **SM_Debug** mode must be on.

SM Animation

The block parameter is the number of states to be animated; it can be overestimated, but it is limited to 31.

Part II

Continuous-time state machines

Chapter 8

Introduction

State machines considered so far were activated at discrete times by events from an external source. All the state machine activities were limited to these activations; the state machine itself could not generate new activations.[2] This in particular meant that the **Top_SM_Diagram** block was only activated through its input activation port. The activations were limited to events (discrete-time activations) but did not have to be periodic.

The state machines considered here go beyond this limitation and allow the state machine to exhibit continuous-time dynamics. There exist different types of state machines with continuous-time dynamics. They will be discussed in the next chapter.

The continuous-time dynamics extension presented here is still experimental. The blocks should be considered in beta stage. The framework allows modeling complex hybrid systems, which are known to be difficult to simulate.

---

[2]The run-to-completion option creates additional activations but they are associated with, and occur at the same time, as external activations.

Chapter 9

Library blocks for continuous-time state machines

In the case of a continuous-time state machine, the construction is similar to the construction of state machines considered previously but different blocks should be used in some cases. ASM blocks used specifically for constructing continuous-time state machines are available in the sub-palette **ContinuouslyActive** of the ASM palette.

9.1   ContinuouslyActive sub-palette

The following blocks are dedicated to constructing state machines with continuous-time dynamics.
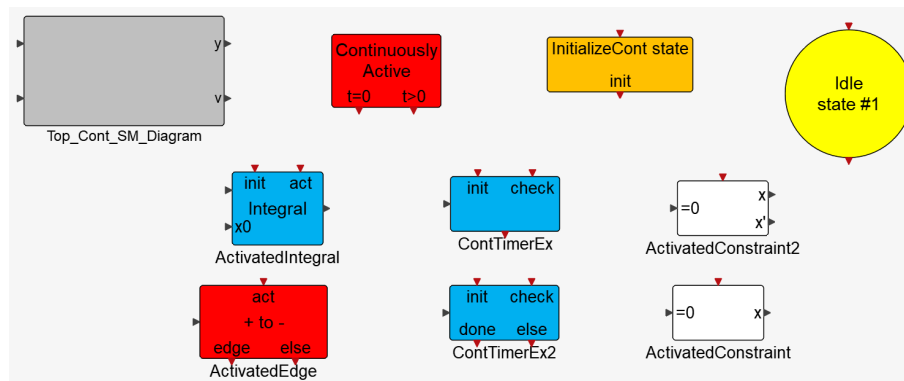
Some of the blocks replace similar blocks used for the construction of models in the discrete-time case. Others are specific to the usage in the continuous-time, for example the constraint blocks and zero-crossing blocks.

9.2   Construction of continuous-time state machines

As in the discrete-time case, a top state machine diagram block is provided to be used as a prototype for the construction of a new state machine. The **Top_Cont_SM_Diagram** contains the basic elements needed in the construction of a continuous-time state machine diagram. This block, unlike its discrete-time counterpart, is not declared Atomic by default. The Atomic property, needed in the discrete time,

in particular in case where the run-to-completion option was true, is not required in this case. In some cases, especially for hybrid systems where the state machine itself contains continuous-time states, its usage may even prevent the simulation of the model.

Note that since the **Top_Cont_SM_Diagram** super block is not necessarily atomic, the signal names used by set and get signals inside the super block are no longer isolated from the rest of the model. So care must be taken to insure the same signal names are not used inside and outside unintentionally.

The other difference between **Top_SM_Diagram** and **Top_Cont_SM_Diagram** is that the latter does not have an input activation port. The reason is that in continuous time operation, the block is not activated by external events. It contains its own always active source. This can be seen by examining its content shown below
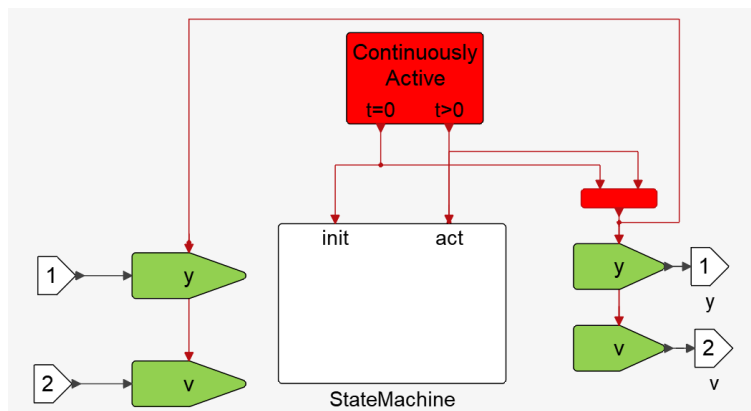

The **ContActivation** (Continuously Active) block provides an always active activation signal similar to the activation signal provided by the **AlwaysActive** block but separated in two: the initial activation signal and the rest of the activation signals.

The state machine provides a level of hierarchy, in particular for the construction of parallel state machines, as in the discrete time case:
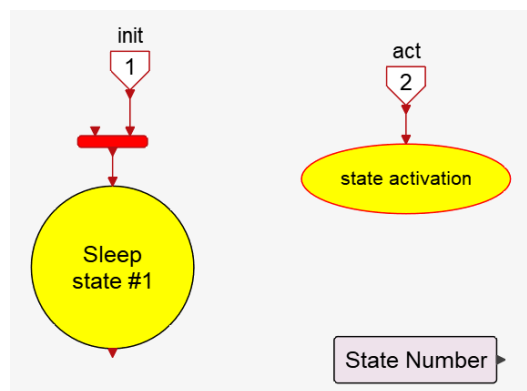
The content of the **StateBlock** however in this case is slightly different: it uses a different initialization block:
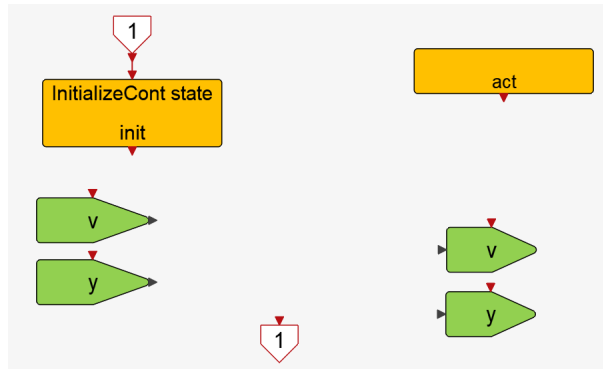
The **InitializeCont** block is the continuous-time counterpart of the **Initialize** block. The **Action** and the set and get blocks are the same as in the discrete-time case.

The usage of other blocks in this sub-palette will be illustrated in the next chapter via examples.

Chapter 10

Different types of continuous-time state machines

Any state machine not strictly activated by external events is referred to here as a continuous-time state machine. This includes many different types of state machines.

In some cases, the state machine still operates at event times, but the events are generated by the state machine itself. So, what makes the state machine continuous is the possibility to generate internally events at any time. Consider for example a Poisson point process where events are generated at random times.

In other cases, the state machine is activated over periods of time (or even always activated). Consider for example a state machine that reacts to the zero-crossing of an input signal. If the input signal is a continuous-time signal, the state machine must operate in continuous time to detect and isolate the time of crossing.

The most complex situation is when the state machine itself contains continuous-time states. Such models can be used to model hybrid systems. In different states, the system exhibits different dynamical behavior. Even the size of the state and the number of equations of the system may differ from one state to the other.

The above cases are discussed in the following sections using examples.

10.1   Modeling asynchronous events

Events occuring over time, such as in queuing processes, can be seen as asynchronous events. For example, the events corresponding to clients entering and leaving check-out lines in a grocery store are in general not considered synchronous. The events are considered random in time, following for example the exponential law (Poisson process). Such processes can be modeled using the continuous-time state machines presented here.

Consider a single queue, for example a single check-out counter, where the arrivals correspond to a Poisson process and the service time is fixed. This system can be modeled using two parallel running state machines, modeling respectively the arrivals and the departures:
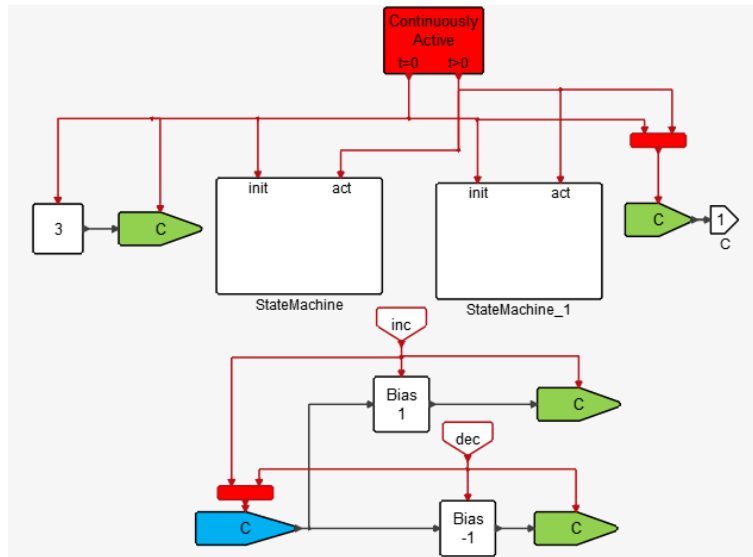
An extended state (signal C) is used to hold the state of the queue, i.e., the number of people waiting. This state is incremented and decremented by the transitions within the two state machines. The incrementing is done by the arrival event, generated by the first state machine shown below
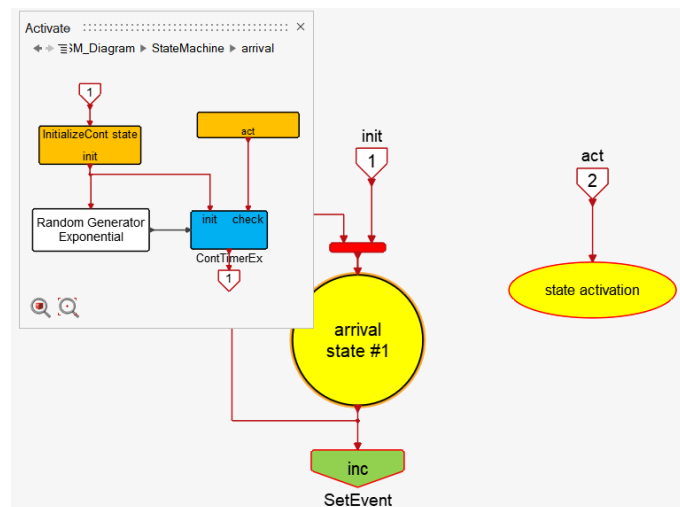
The model uses a single state, which implements a random delay before transitioning. Note the use of the **ContTimerEx** block which generates an event at the end of a time-out period. The time-out value is provided by the input signal at block initialization. So, in this case, the time-out period is a random number.

The **ContTimerEx** event is generated as soon as the period is over. This is in contrast with the discrete time timer block operation where the event is a redirection of the following discrete event, activating the **Top_SM_Diagram** block.

The state machine implementing the departure process is more complex because a departure cannot happen if the queue is empty. So, the state machine is implemented with two states representing respectively the empty and the non-empty situations:

ASM library 82
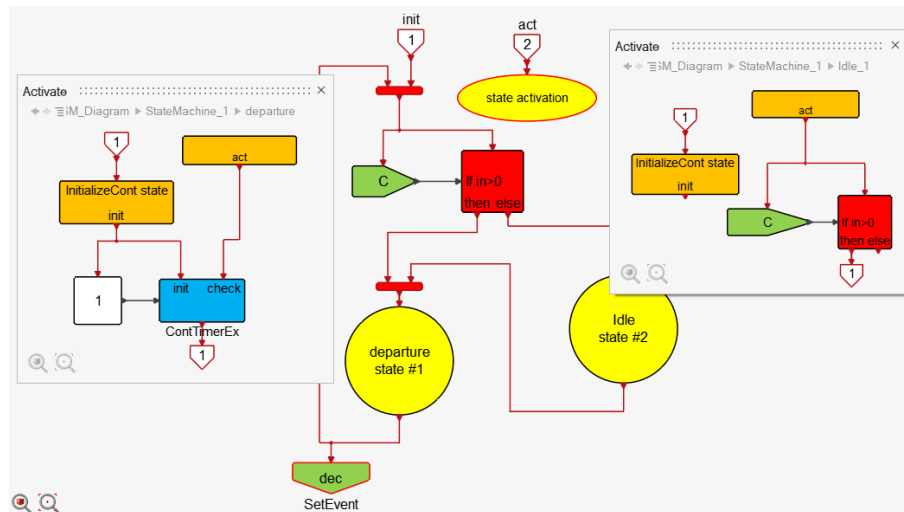
The process time here is assumed constant equal to 1. Other models for the process time can be implemented by simply using other inputs to the timer block instead of a constant. The process time can be made random, deterministic but time varying, or even made dependent on the model states, for example on the number of clients waiting in line.[1]

A typical simulation result, with initial queue state set to 3, is shown below:
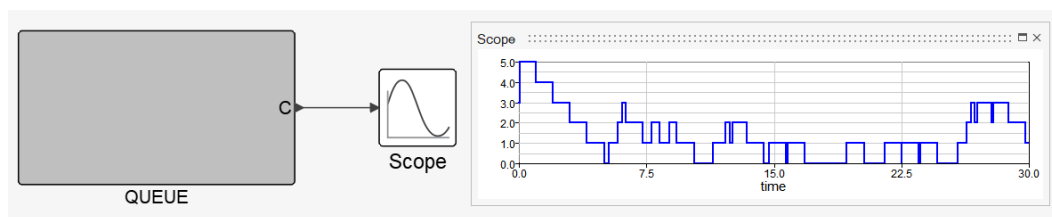
## 10.2   Always-active state machines

In the previous example, the state machine was activated at discrete time instants, but the model had to be continuous-time because the activation times were a priori arbitrary. In the following example, the state machine is always active because it generates events by detecting the zero-crossings of a continuous-time input signal.

The example is the StateCharts model already considered in Section 2.2. The top diagram of the state machine is implemented using a single state machine:

---

[1]Consider for example the cashier working faster when the queue becomes longer.

83 ASM library
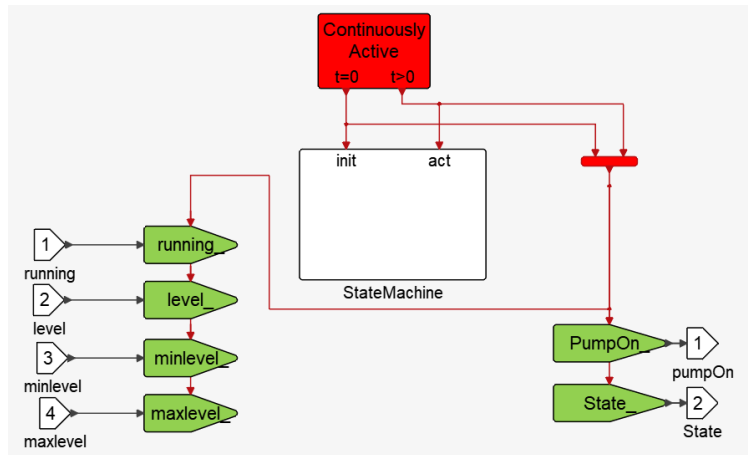
The state machine uses 3 states, as in the discrete-time implementation:
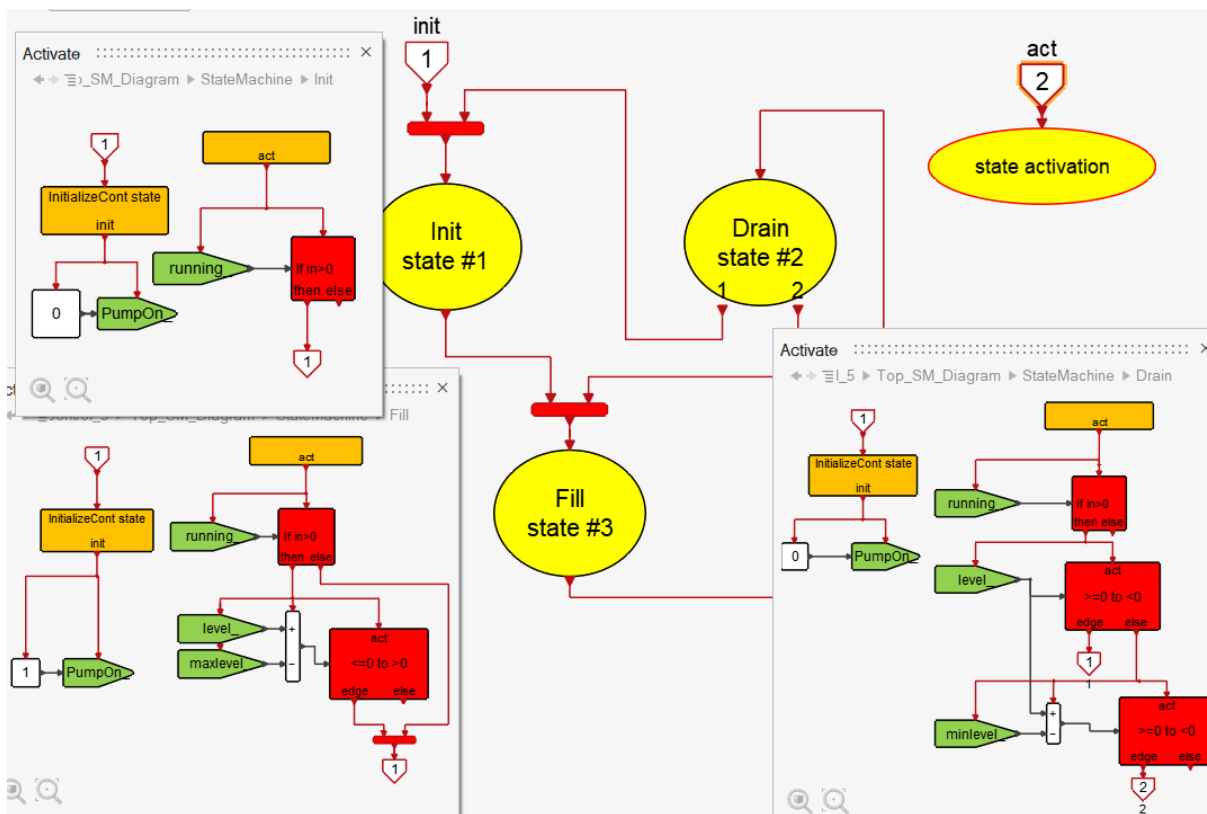
In this implementation the inequality tests in the original StateCharts model are implemented with zero-crossing tests. The **ActivatedEdge** blocks are used to detect the crossings of the threshold temperatures. The simulation can now be performed using variable step-size solvers.

10.3 Hybrid systems

Hybrid systems are dynamical systems obtained by mixing smooth continuous-time dynamical systems governed by differential equations with discrete events. Events interact with the continuous part for example by creating jumps in the state variables, or even altering completely the set of active differential

equations. And, the continuous-time dynamics interacts with the discrete part by for example generating zero-crossing events [9].

Continuous-time state machines can be used to model hybrid systems. Two examples are presented here. The first one is a classical bouncing ball model. The second model represents the dynamics of a moving vehicle with its wheels not being permanently in contact with the ground.

Bouncing ball

This model is a new formulation of a state machine model representing the dynamics of a ball bouncing on the ground originally presented in [3]. The ball height $y(t)$ satisfies the following differential equation

$$\ddot{y}(t) = -g. \tag{10.1}$$

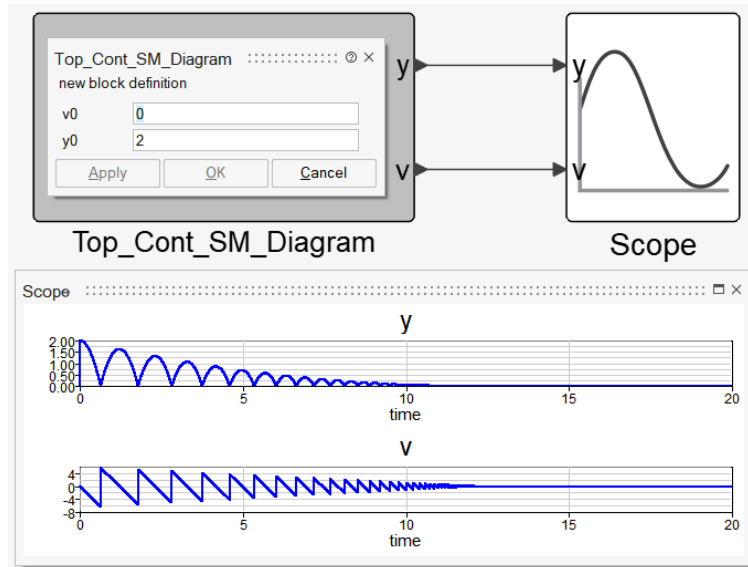When the ball hits the ground (at $y = 0$), the ball bounces back up losing 10% of its speed, i.e.,

$$\dot{y}(\tau) = -0.9\dot{y}(\tau^-),$$

where τ is a time of impact. The state of the ball is fully specified in terms of the ball's height y and speed $v = \dot{y}$.

In theory, the ball bounces an infinite number of times. The bounces become smaller as time goes on, and at some point, the model becomes non-physical. The assumption is made here that the ball comes to rest on the ground as soon as the speed by which it hits the ground falls below a treshold. So, the state machine used has two states
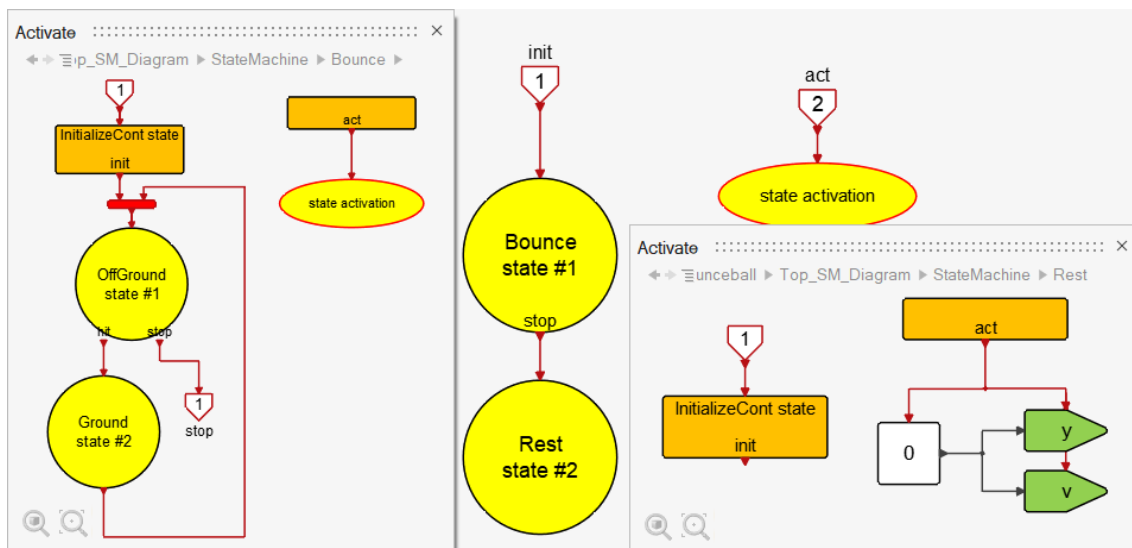

one representing the bouncing state, and the other the resting state. The system can transition from the bounce state to the rest state, but not the other way around. The bounce state contains another state machine corresponding the case where the ball is in the air, and the case when the ball hits the ground:

85 ASM library

In the OffGround state, **ActivatedIntegral** blocks are used to solve for the speed v and position y. The contact with the ground is detected by an **ActivatedEdge** block. This block generates an event at contact causing an exit from the state. Two possible exits are possible depending on whether the speed of the ball at contact is below a threshold or not.

The Ground state is a simple transition state: the speed of the ball is changed and the state is exited immediately.

The initial height and speed of ball are used as parameters of the **Top_Cont_SM_Diagram**. The initialization is done in the top diagram:

Note that the integrator blocks are re-initialized at initial time with these values so their initial conditions are irrelevant; they are left at their default zero values.

The simulation result is shown below

As expected the ball bounces over a period of time and then rests on the ground.

Vehicle with wheels on and off the ground

This system was originally introduced in [2]. It was modeled and simulated without using state machines. A state machine formulation was presented in [3]. The model used here is a new implementation of this latter model using the ASM library blocks.
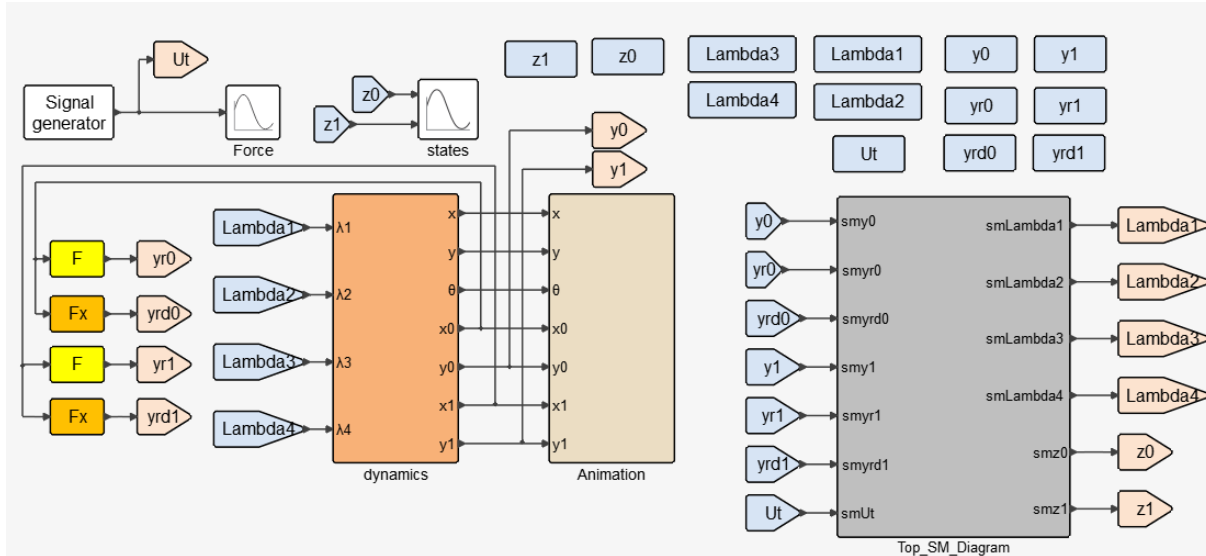

The model represents the dynamics of a 2-wheel vehicle moving straight (no steering) on a non-flat surface. The wheels can be in contact with the ground or be off the ground. So, each wheel can be in two different states (on or off the ground). Only parts of the continuous-time dynamics of the overall system depends on these states, so part of the dynamical system is modeled outside of the state machines, and part of it, inside.

Only the state machine part of the system is discussed here.[1] The equations of the full model can be

---

[1]The demo model `car_hybrid_sm2` contains the full model.

87 ASM library

found in [3] (Chapter 3), and their details are not essential in presenting the implementation of the state machines discussed here. They include the equations of motion for a simplified two-wheel vehicle. The wheels are connected to the body of the vehicle by two springer dampers. All the vehicle's mass is supposed to be concentrated in the body. The vehicle is subject to the force of gravity. It is also subject to contact forces with the ground if a wheel is on the ground (preventing the wheel from penetrating in the ground). Addtionaly, for the rear wheel, the contact force may include a force produced by the vehicle, representing acceleration and braking.
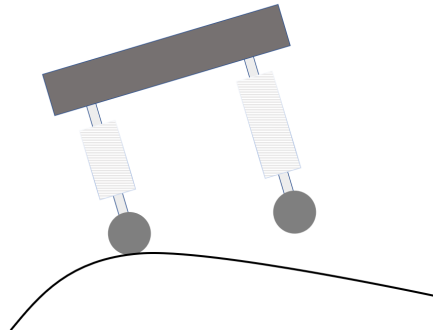

The system equations, when the wheels are on the ground, are expressed as a system of DAE (Differential algebraic equations). The algebraic equations correspond to the constraints that the wheel contact points remain on the curve representing the ground profile. The corresponding algebraic variables are the contact forces. A wheel goes off the ground when the corresponding contact force becomes negative (the ground does not hold down the vehicle), in which case the system equations change: the constraint corresponding to the wheel is removed and the contact force becomes zero.

The change in the state of a wheel (on and off the ground) is thus detected as follows: if the wheel is on the ground, the contact force is monitored and a state change is declared when the contact force crosses 0; when off the ground, the position of wheel is compared with the ground profile and a state change is declared when a zero-crossing occurs in their difference.

The state machines in this case represent the states of the rear and front wheels. Each wheel can either be in contact with the ground, or not. So, the state diagram contains two parallel state machines, each having two states
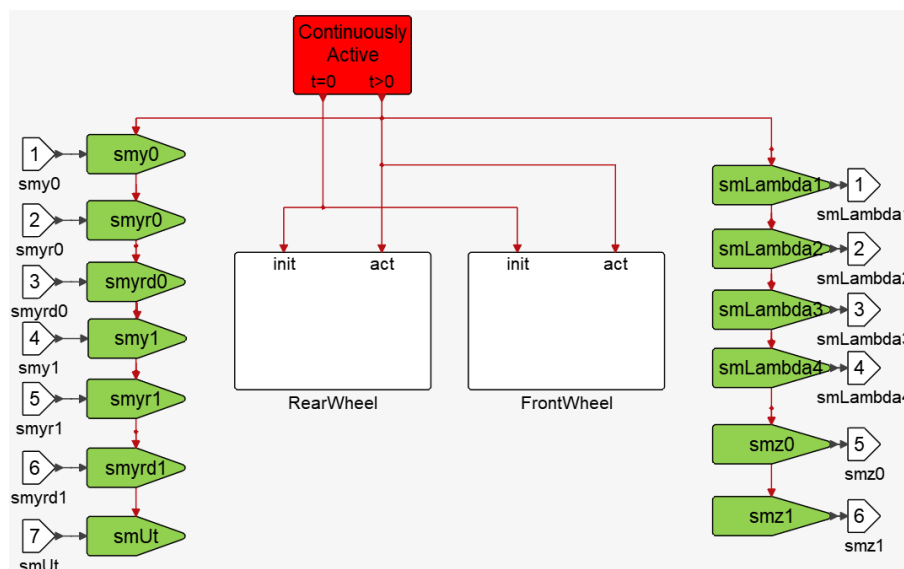
The state machine associated with the real wheel is shown below
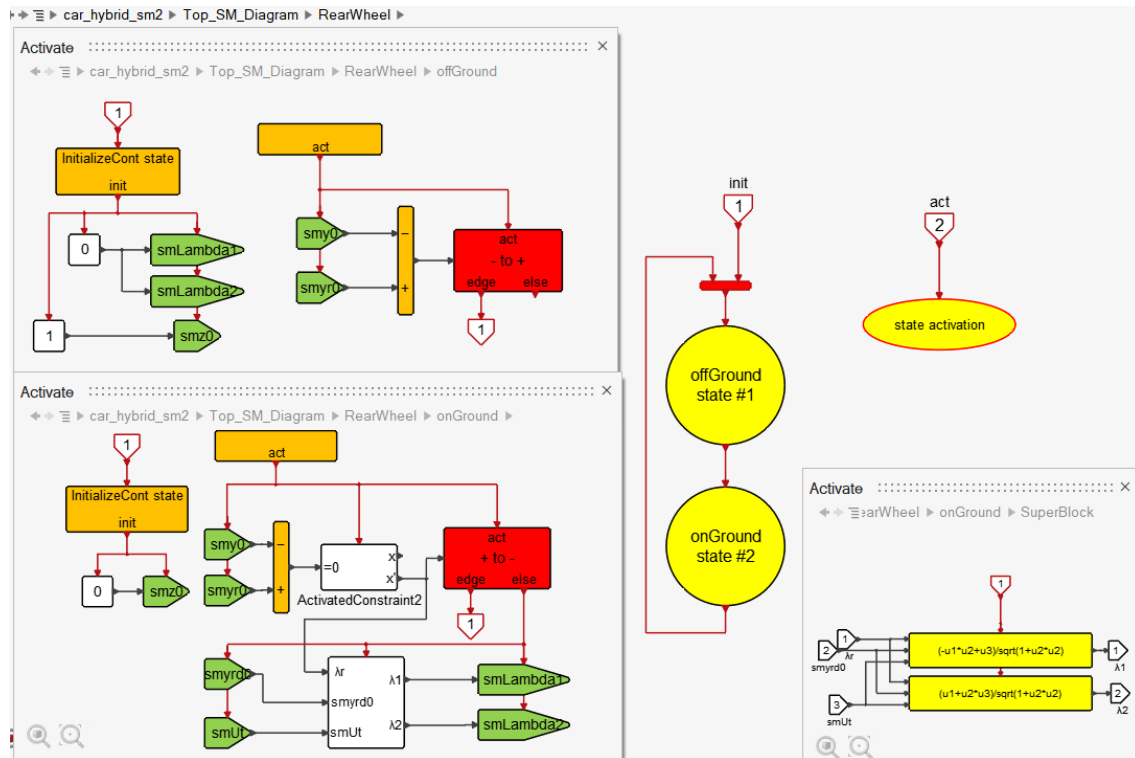
The complete model of the system needs to be considered to fully understand the above diagrams but note simply that when a wheel is off the ground, there is no contact force with the ground. This force represented by a variable λ (obtained from a Lagrangian formulation of the mechanical problem), is then identically zero when the wheel is off the ground, and needs to be calculated when it is on the ground. The calculation requires solving a nonlinear system, which is done here using a constraint block.

The front wheel state machine is similar. It is the following
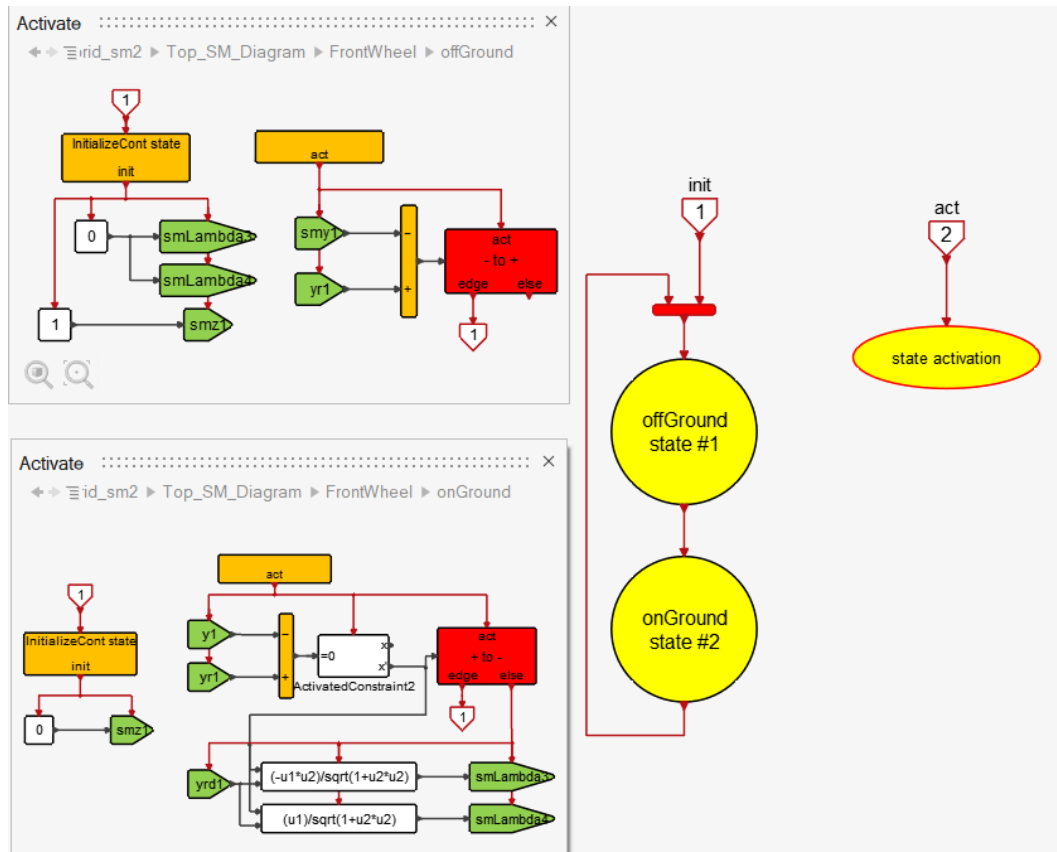
The difference with the rear wheel state machine is that the rear wheel is subject to torque from the vehicle engine, which can be used to accelerate or brake.

The simulation result is best seen through an animation using the **Anim2D** block:
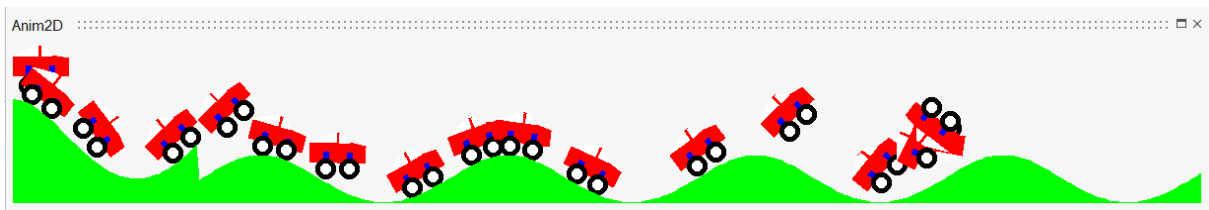
The vehicle is dropped on top of a hill. Zero, one or both wheels are touching the ground at any time as it rolls down the hill. Towards the end, the vehicle accelerates leading to a high jump and a disastrous ending.

Chapter 11

Examples

11.1   Sticky masses

The sticky masses (or balls) model presented here is a re-implementation of a model presented in [3], which was a variation of the sticky masses model presented in [6], Chapter 4.

Two masses are placed on a frictionless surface. They are each attached to a spring with the opposite side of the spring fixed on the surface. The masses only move on the segment connecting the attachement points to the surface, as shown below
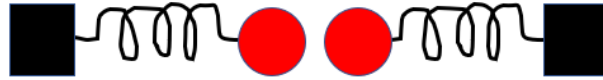
If the masses come into contact, they are attracted by a stickness force and may get attached. In that case they remain stuck until the spring forces pulling them apart exceeds the stickness force. The stickness force is supposed to decay exponentially.

Denoting the positions of the masses x_1 and x_2, and the springs' coefficients k_1 and k_2, the spring forces F_1 and F_2 are

$$F_1 = -k_1(x_1 - a)$$
$$F_2 = -k_2(x_2 - b).$$

So when the masses are separate, their movements follow the following equations

$$\ddot{x}_1 = -k_1(x_1 - a)/m_1$$
$$\ddot{x}_2 = -k_2(x_2 - b)/m_2$$

where m_1 and m_2 represent their masses. The masses of the springs are neglected.

When the two masses are attached following a collision (when $x_1 = x_2$), they can be considered as a single object with mass $m_1 + m_2$. So its equation of motion is

$$\ddot{x}_3 = -(k_1(x_3 - a) + k_2(x_3 - b))/(m_1 + m_2).$$

where $x_3 = x_1 = x_2$. In this case the force of contact F between the two masses is

$$F = (m_2 F_1 - m_1 F_2)/(m_1 + m_2).$$

The force pulling apart the two masses is $-2F$, and separation occurs if this force exceeds the stickness force s:
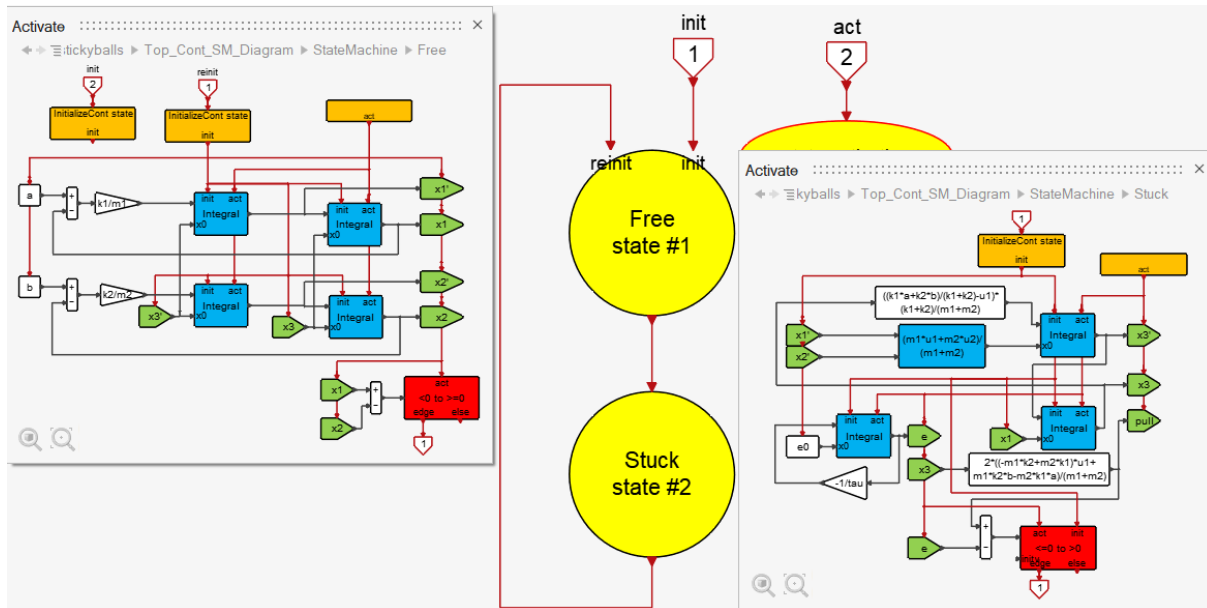
$$2\frac{(m_2 k_1 - m_1 k_2)x_3 - (m_2 k_1 a - m_1 k_2 b)}{(m_1 + m_2)} > s.$$

The stickness force is supposed to satisfy the following equation

$$\dot{s} = -s/\tau$$

The variable s is re-initialized to $s_{\max}$ every time the masses come into contact.

The **Twin Activate** model for this system uses a state machine with two states: two masses separate, and two masses attached. The contents of the states implement the equations given above
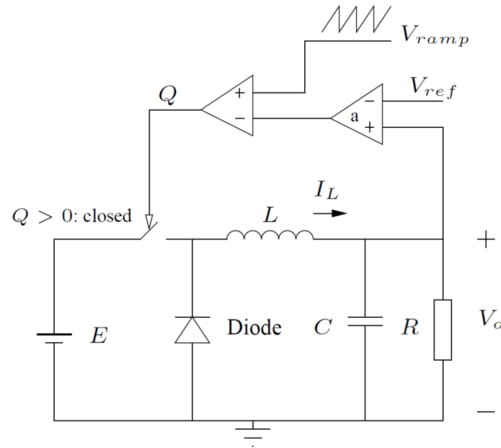

The simulation result (curves representing x_1 and x_2) shows that after collision, the masses may remain stuck for a period of time or detach instantly:
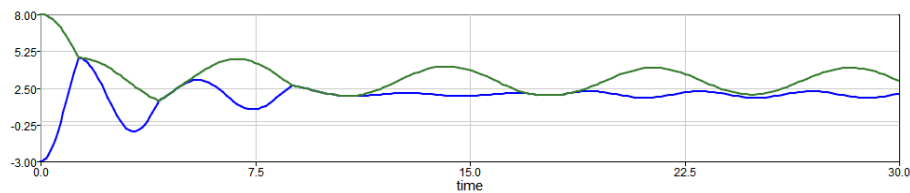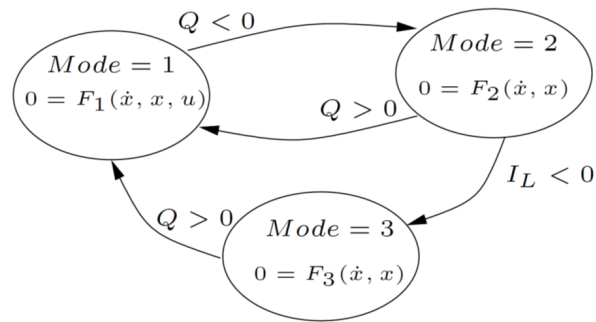

11.2   DC/DC Buck Converter

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 [5]. The DC/DC buck converter circuit behavior

can be formulated as a bybrid system

where the state $x = (V_o, I_L)$ eveloves in 3 different modes (states):

$$0 = F_1(\dot{x}, x, u) = \begin{cases} RC\dot{x}_1 + x_1 - Rx_2 \\ L\dot{x}_2 + x_1 - E \end{cases}$$
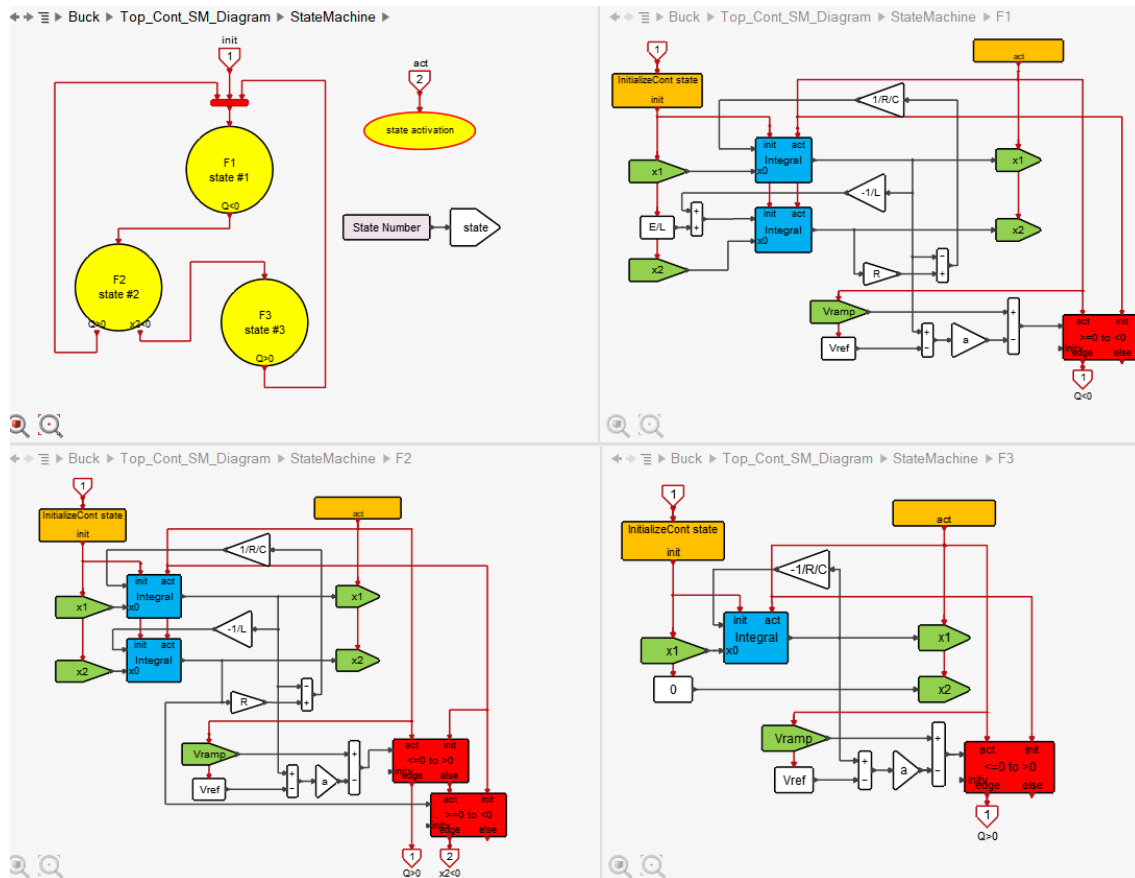
$$0 = F_2(\dot{x}, x) = \begin{cases} RC\dot{x}_1 + x_1 - Rx_2 \\ L\dot{x}_2 + x_1 \end{cases}$$

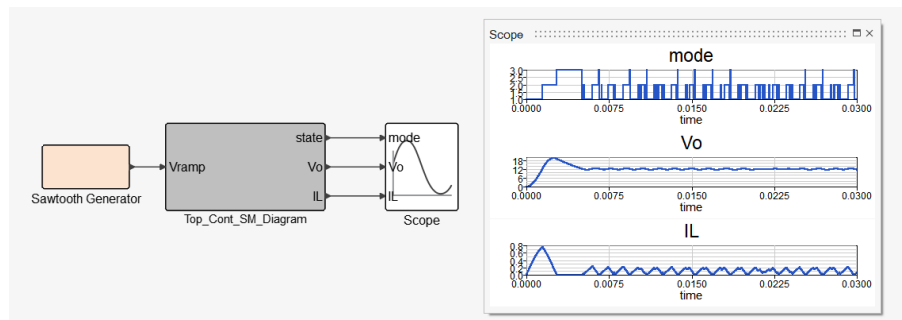$$0 = F_3(\dot{x}, x) = \begin{cases} RC\dot{x}_1 + x_1 \\ x_2 \end{cases}$$

with

$$Q = V_{ramp} - (x_1 - V_{ref})a$$

This formulation is presented in [1] where it is modeled using the **Automat** block. This hyrbid system here is modeled as a state machine:
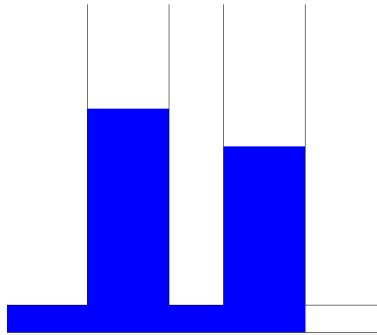
93 ASM library

The simulation result for parameter values $L = 20mH$, $C = 47\mu F$, $R = 110\Omega$, $a = 8.4$, $V_U = 8.2V$, $V_L = 3.8v$, $V_{\text{ref}} = 11.3v$, $E = 20v$, $T = 400\mu S$, is as follows


11.3   Two-tank level control

The state machine presented here is a reformulation of an **Twin Activate** demo where the level of liquid is conrolled in two communicating tanks.
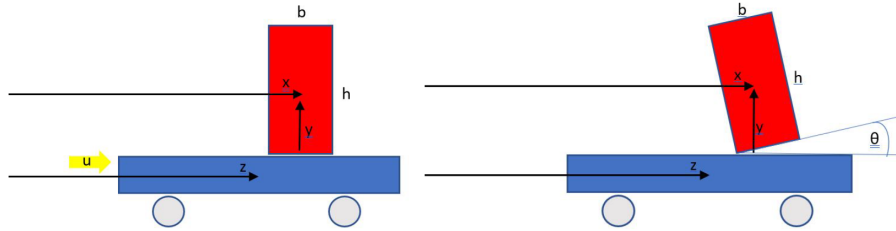
The control consists of turning on and off a pump, which can inject liquid in the first tank with a constant flow. The second tank is emptied through a valve, placed at the bottom of the tak, which is opened and closed at random times. The pipe connecting the two tanks is also placed at the bottom of the tanks.

The level control strategy is implemented using a state machine. This strategy, to control the levels h_1 and h_2 in the two tanks, uses three parameters $h1_{max}$, $h2_{up}$ ad $h2_{down}$ ad operates as follows: the pump is off if $h1 \geq h1_{max}$; otherwise, the pump is turned on if h_2 drops below $h2_{down}$, and it is turned off if h_2 goes above $h2\_textup$.

The controller is implemented by a hierarchical state machine. At the first level, the level of the first tank $h1$ is used to
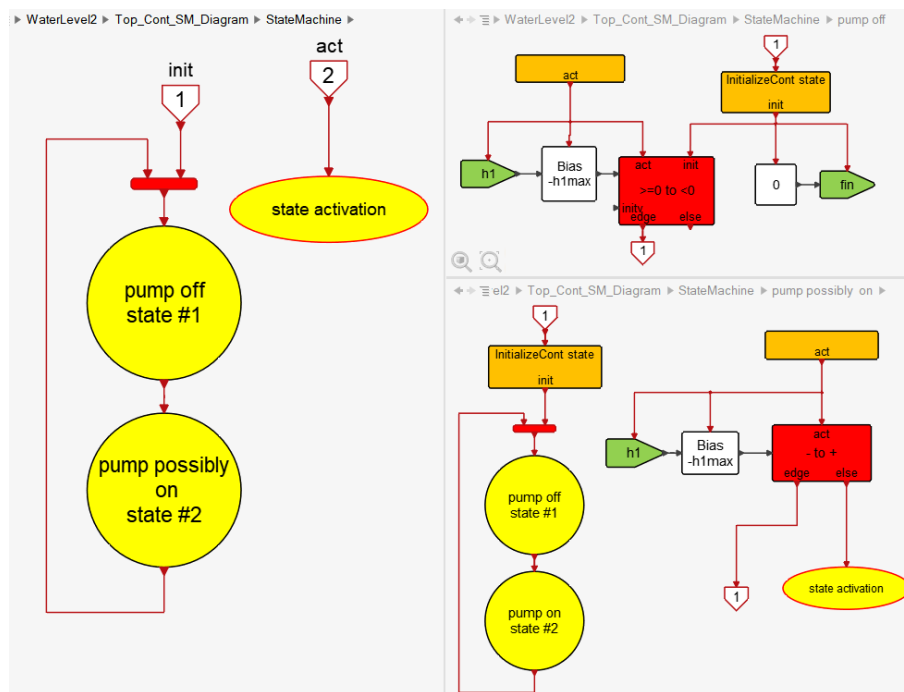
switch between two states: pump off, and pump possibly on. In the second state, the pump can be on and off depending on the level $h2$. This is modeled using another state machine with two states

95 ASM library

The controller is implemented as a **Top_Cont_SM_Diagram**:

11.4 Block on a cart

This example is a re-implementation, using the ASM library blocks, of an **Twin Activate** demo. It corresponds to a mechanical system consisting of a rigid block placed on top of a cart, moving horizontally, subject to a control force. The block has height h, width b and mass m. The cart has mass M, and is subject to force $u(t)$ as shown below

On the left figure, the block is flat on its base and moves along with the cart or slides on the cart surface, due to friction between the base of the block and the surface of the cart.

If the cart accelerates fast enough, the block may tilt with only a corner staying in touch with the cart. The tilt angle is denoted θ. See the right hand figure. In this configuration, it is assumed that the friction is high enough so that no sliding occurs. Typically after tilting, the block drops and bounces against the surface of the cart leading to a rocking motion. At every bounce, the block loses energy, so angular velocity $\dot{\theta}$. For small enough $\dot{\theta}$, the block is assumed to stop flat on the surface.

What makes this system hybrid is that the block may move along, slide or rock on top of the cart. When the block is flat on the cart, its equations of motion are different from when it is tilted with a corner touching the cart's surface. When the block is flat, viscous friction combined with Coulomb friction and static friction is assumed between the block and the cart surface. Such systems are also often modeled as hybrid systems.

This system can be in three states:

- A_1: the block flat on the cart ($\theta = 0$),

- A_2: the block tilting to the left ($\theta > 0$), and

- A_3: the block tilting to the right ($\theta < 0$).

The first state ($\theta = 0$), has two substates:

- B_1: the block moving with the cart, and

- B_2: the block sliding on the surface of the cart.

System equations

Let z denote the horizontal position of the central of gravity of the cart. x the position of the block, and y its vertical position. Let d be the displacement between the cart and the block ($d = x - z$ when $\theta = 0$) and let

$$R = \sqrt{h^2 + b^2}/2$$

and

$$\alpha = \arctan(h/b).$$

Then the equations of motion in different states can be expressed as follows.

State A_2 ($\theta > 0$)

The equations in this state can be obtained similarly to the previous case:

$$\begin{aligned} x &= z + d + R\cos(\theta + \alpha) - b/2 \\ y &= R\sin(\theta + \alpha). \end{aligned}$$

97 ASM library

The system's kinetic energy can then be expressed as

$$T = \frac{1}{2}(M\dot{z}^2 + m(\dot{x}^2 + \dot{y}^2) + J\dot{\theta}^2)$$

and the potential energy

$$V = mgy.$$

The equations of motion can be obtained as follows

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{z}} - \frac{\partial L}{\partial z} = u$$

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

where $L = T - V$, leading to

$$\begin{pmatrix} M+m & -mR\sin(\theta+\alpha) \\ -mR\sin(\theta+\alpha) & J+mR^2 \end{pmatrix} \begin{pmatrix} \ddot{z} \\ \ddot{\theta} \end{pmatrix} = \begin{pmatrix} u + mR\cos(\theta+\alpha)\dot{\theta}^2 \\ -mgR\cos(\theta+\alpha) \end{pmatrix}. \tag{11.1}$$

State A_3 ($\theta < 0$)

In this state,

$$x = z + d - R\cos(-\theta+\alpha) + b/2$$

$$y = R\sin(-\theta+\alpha)$$

and

$$\begin{pmatrix} M+m & -mR\sin(-\theta+\alpha) \\ -mR\sin(-\theta+\alpha) & J+mR^2 \end{pmatrix} \begin{pmatrix} \ddot{z} \\ \ddot{\theta} \end{pmatrix} = \begin{pmatrix} u + mR\cos(-\theta+\alpha)\dot{\theta}^2 \\ mgR\cos(-\theta+\alpha) \end{pmatrix}. \tag{11.2}$$

State A_1 ($\theta = 0$)

In this state, the block movement is subject to the force of friction F between the block base and the cart surface. The cart may be sliding or not, and the equations of motion are

$$m\ddot{x} = F \tag{11.3}$$

$$M\ddot{z} = u - F. \tag{11.4}$$

The displacement between the cart and the block is then obtained as

$$d = x - z.$$

The two situations, non-sliding and sliding, are represented as two substates.

Substate B_1 (block not sliding) In this case,

$$\dot{x} = \dot{z} \tag{11.5}$$

which thanks to (11.3,11.4) implies that

$$F = \frac{m}{M+m}u.$$

The system stays in this state as long as

$$\left| \frac{m}{M+m} u \right| \le \mu_s F_n$$

where μ_s is the static coefficient of friction and

$$F_n = mg,$$

the normal force, unless the system moves to states A_2 or A_3 before.

For the system to move to state A_2, i.e., the block to tilt left, $\ddot{\theta}$ in (11.1), for zero θ and $\dot{\theta}$, must be positive. This condition can be expressed as

$$u > (M+m)gb/h. \tag{11.6}$$

For the system to move to state A_3, i.e., the block to tilt right, $\ddot{\theta}$ in (11.2), for zero θ and $\dot{\theta}$, must be negative. This condition can be expressed as

$$u < -(M+m)gb/h. \tag{11.7}$$

Substate B_2 (block sliding) When the block is sliding ($\dot{x} \ne \dot{z}$), the friction force is defined as

$$F = -\mu_d F_n \text{sign}(\dot{x} - \dot{z}) - \beta(\dot{x} - \dot{z})$$

where μ_d is the dynamic coefficient of friction and β, the viscous friction coefficient. The system leaves this state if condition (11.5) is encountered, or the system moves to states A_2 or A_3, i.e., if conditions (11.6) or (11.7) are satisfied.

Re-initialization of state variables

When passing from one state or substate to another, the system equations change and their state variables must be re-initialized. In most transitions, the variables z, $\dot{z}$, x, $\dot{x}$, θ and $\dot{\theta}$ are preserved, except when the block rocks (bounces against the cart) where at each bounce

$$\dot{\theta}^+ = r\dot{\theta}^-$$

and if $\dot{\theta}^-$ is smaller than a given threshold, $\dot{\theta}^+ = 0$. And when the system moves from B_2 to A_2 or A_3. In substate B_2, the block is sliding, i.e., $\dot{x}$ is not equal to $\dot{z}$, but once in A_2 and A_3, no sliding occurs which means that at the entry, since θ and $\dot{\theta}$ are zero, $\dot{x}$ must be equal to $\dot{z}$. This happening instantaneously amounts to having a collision where the conservation of momentum implies

$$\dot{z}+ = \dot{x}^+ = (M\dot{z}^- + m\dot{x}^+)/(M+m).$$

Implementation in Twin Activate

The implementation in **Twin Activate** is done by creating a hierarchical state machine, in particular to represent the states A_1, A_2, A_3, and the A_1 substates B_1 and B_2:

99 ASM library
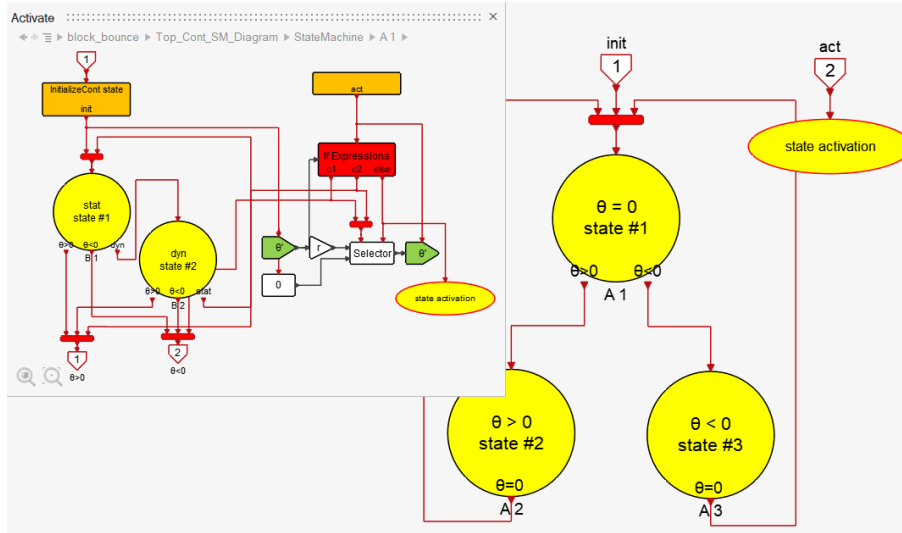
The contents of the states implement the system equations in different situations. Note that A_1 can may exit before any activation of the substates B_1 and B_2. This happens if the magnitude of $\dot\theta$ is above a threshold. In this cases the block bounces back instead of lying flat.

The contents of the substates B_1 and B_2 are shown below
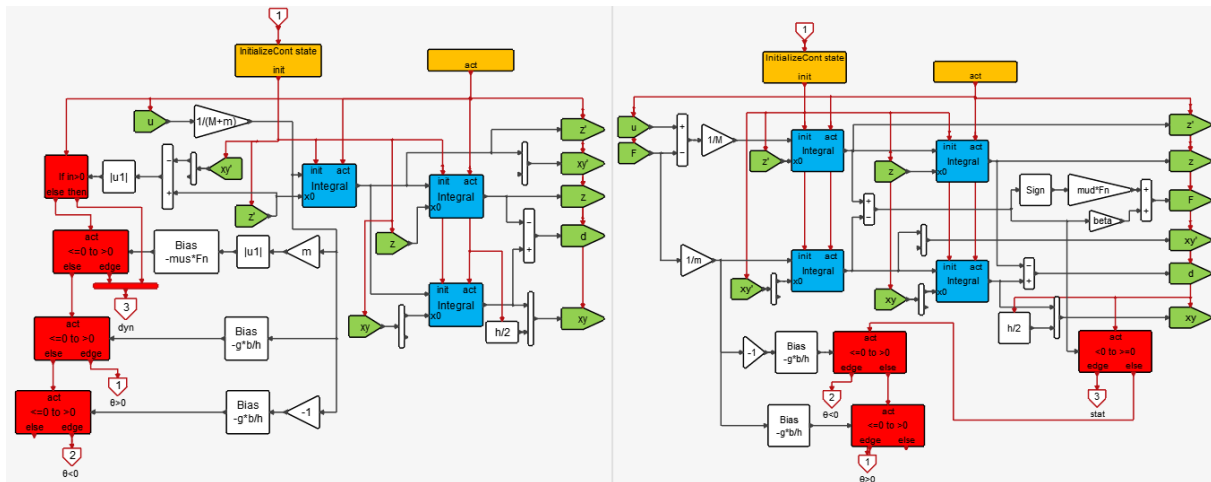
The **ActivatedIntegral** blocks are used to implement the differential equations of the system and their (re-)initializations.

The states A_2 and A_3 contain the following diagrams
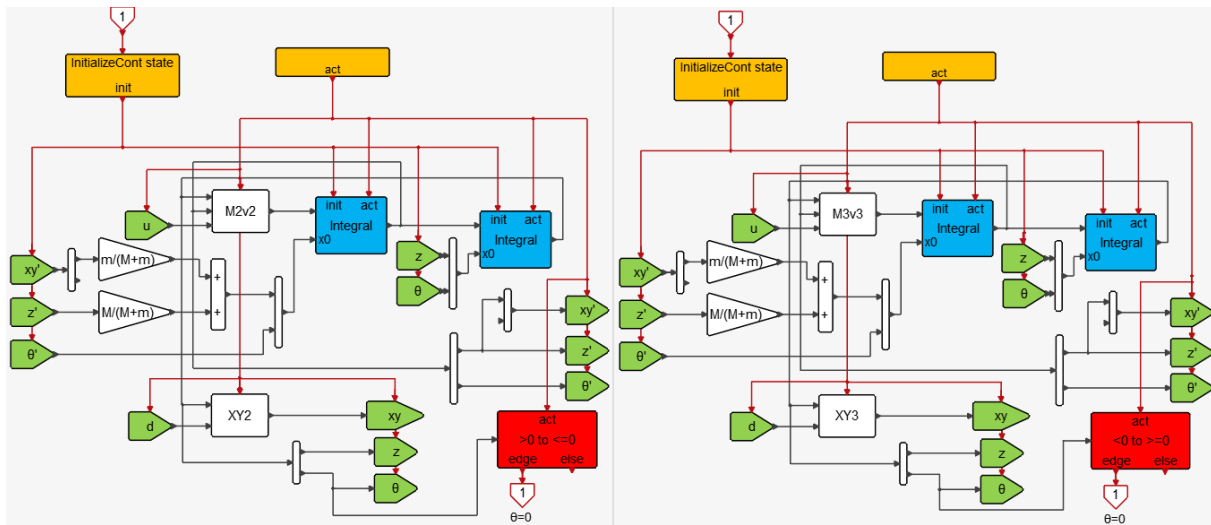
The expressions used in **MatrixExpression** blocks are parameterized and defined the super block contexts as follows
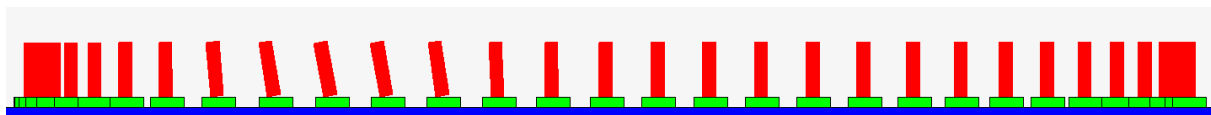
```
M2v2='[M+m,-m*R*sin(a+u1(2));-m*R*sin(a+u1(2)),J+m*R^2]\[u3+m*R*cos(a+u1(2))*u2(2)^2;-m*g*R*cos(a+u1(2))]';
M3v3='[M+m,-m*R*sin(a-u1(2));-m*R*sin(a-u1(2)),J+m*R^2]\[u3+m*R*cos(a-u1(2))*u2(2)^2;m*g*R*cos(a-u1(2))]';
XY2='[u1(1)+u2+R*cos(a+u1(2))-b/2;R*sin(a+u1(2))]';
XY3='[u1(1)+u2-R*cos(a-u1(2))+b/2;R*sin(a-u1(2))]';
```
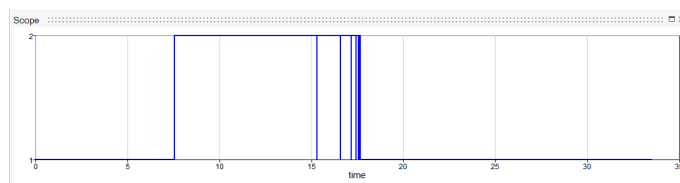
The simulation result for the following input force
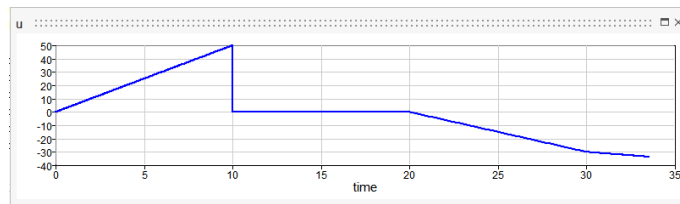
can be seen via the animation block:

Initially the cart is pushed to the right, accelerating it. Then it is pushed to the left, slowing it down. Note that the block rocks and slides on the cart. The Changes in the states are shown below

showing the states of the top state machine switching between A_1 and A_2.

101 ASM library

Chapter 12

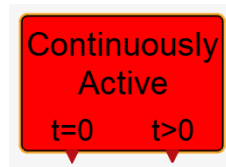# Continuous-time specific blocks

12.1   InitializeCont block

This block is the continuous-time counterpart of the **Initialize** block. Its operation is similar and should be used the same way for capturing the state initialization event.
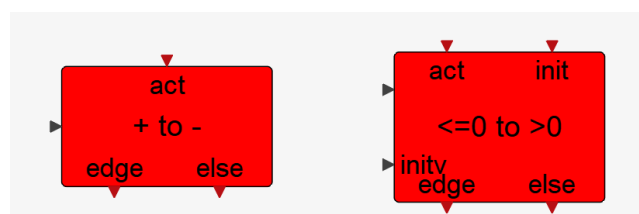
This block has no parameter.

12.2   ContActivation block

This block should be used inside the continuous-time state machine to activate the states. It provides the intial event separately.
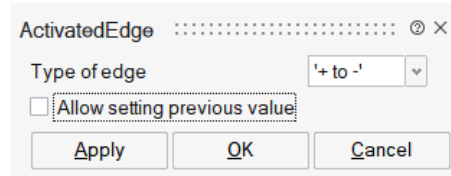
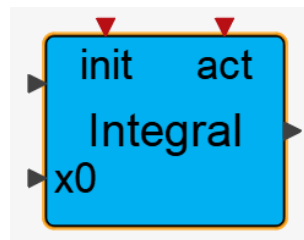This block has no parameter.

12.3   ActivatedEdge block

This block is used for detecting zero crossings. Optionally, the "previous" input value can be set to have full control over the first activation of the block. For example if a state must exit if the input of the block is > 0, then simply detecting a zero-crossing to initiate an exit may not be enough: when entering the state, the value of the input may already be > 0. This could happen in particular at initial time. By setting the "previous" input value to 0, a zero-crossing is detected and the state is exited.

The block parameters are the choice of the type of zero-crossing (same as for the **EdgeTrigger** block), and the option to set the previous value of the input:
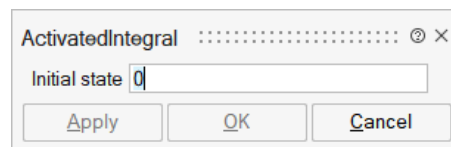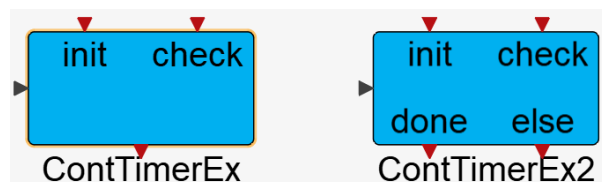

12.4   ActivatedIntegral block

This block is an activated integration block with reset.

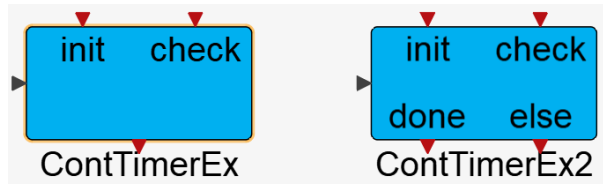The block parameter is the initial state of the integrator:

12.5   Timer blocks

The timer blocks **ContTimerEx** and **ContTimerEx** 2 generate an event when the time period set at initialization and provided by the regular input runs out. The **ContTimerEx** 2 block generates an activation signal via its second output activation port. This signal is the redirection of the input activation of the block when crossing does not occur.

These blocks have no parameter.

ASM library 104

12.6   constraint blocks

The constraint blocks **ActivatedConstraint** and **ActivatedConstraint2** are based on the **Constraint** block. But they impose the constraint that their inputs be zero only when they are activated.

The block parameter is the initial guess value of the state, and additionally in the case of **Activated-Constraint2**, the guess value of its derivative:
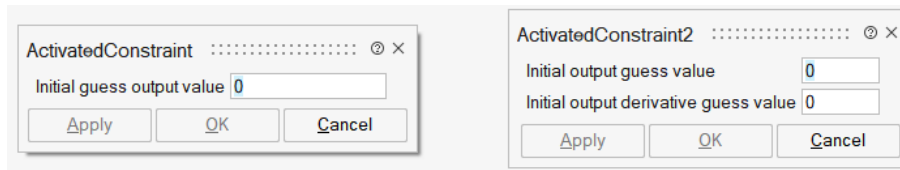
Bibliography

[1] Altair Engineering Inc. *Altair Activate, Extended Definitions*, 2021. available by download.

[2] Stephen L. Campbell, Jean-Philippe Chancelier, and Ramine Nikoukhah. *Modeling and Simulation in Scilab/Scicos with ScicosLab 4.4*. Springer, New York, New York, 2009.

[3] Stephen L. Campbell and Ramine Nikoukhah. *Modeling and Simulation with Compose and Activate*. Springer International Publishing, Springer Nature Switzerland AG, 2018.

[4] Charles André. Semantics of synccharts, 2003. URL: https://www.i3s.unice.fr/~andre/SyncCharts/.

[5] M. di Bernardo, F. Garofalo, L. Glielmo, and F. Vasca. Switching, bifurcations, and chaos in dc/dc converters. *IEEE Transactions on Circuits and Systems*, 45:133–141, 1998.

[6] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems: A Cyber-physical Systems Approach*. LeeSeshia.org, 2011.

[7] Wikipedia contributors. Esterel technologies, 2021. [Online; accessed 30-August-2021]. URL: https://en.wikipedia.org/wiki/Esterel_Technologies.

[8] Wikipedia contributors. Finite-state machine, 2021. [Online; accessed 30-August-2021]. URL: https://en.wikipedia.org/wiki/Finite-state_machine.

[9] Wikipedia contributors. Hybrid system, 2021. [Online; accessed 30-August-2021]. URL: https://en.wikipedia.org/wiki/Hybrid_system.

[10] Wikipedia contributors. State diagram, 2021. [Online; accessed 30-August-2021]. URL: https://en.wikipedia.org/wiki/State_diagram.

[11] Wikipedia contributors. Stateflow, 2021. [Online; accessed 30-August-2021]. URL: https://en.wikipedia.org/wiki/Stateflow.

[12] Wikipedia contributors. Uml state machine, 2021. [Online; accessed 30-August-2021]. URL: https://en.wikipedia.org/wiki/UML_state_machine.