M. Otter, K.-E. Årzén, I. Dressler
DLR Oberpfaffenhofen, Germany; Lund Institute of Technology, Sweden

StateGraph-A Modelica Library for Hierarchical State Machines
pp. 569-578

StateGraph – A Modelica Library for Hierarchical State Machines

Martin Otter[1], Karl-Erik Årzén[2], and Isolde Dressler[2]
[1]DLR Institute of Robotics and Mechatronics, Oberpfaffenhofen, Germany, Martin.Otter@dlr.de
[2]Lund Institute of Technology, Lund, Sweden, {karlerik, Isolde.Dressler}@control.lth.se

Abstract

The new library Modelica.StateGraph is a free Modelica package providing components to model discrete event and reactive systems in a convenient way. It has a similar modeling power as Statecharts, but avoids some deficiencies of Statecharts by using elements of JGrafchart and by using Modelica as an "action" language. An overview of the StateGraph library is given, the available components and an application example. The implementation of the library in Modelica is sketched, especially the needed extension to Modelica that will be available in release 2.2 of the Modelica language.

1 Introduction

This section shortly discusses discrete event formalisms and the relationship to the StateGraph library.

Grafcet [3], or the industrial alias Sequential Function Charts (SFC), is a state-transition based computational model that has been widely accepted in the industrial automation industry for representing sequential control logic. It is defined in the standards IEC 848 and IEC 61131-3. States are represented by steps to which actions can be associated, and the steps are interconnected by transitions with associated Boolean conditions or event expressions. The activity in a Grafcet diagram flows downwards from the top of the diagram. It supports alternative branches, parallel branches, and repetition. Hierarchies are supported in the form of macro steps.

Although Grafcet has the same formal power of expression as an ordinary state machine, it is cumbersome to use for representing larger state-machine oriented models. For these applications the Statecharts formalism is better suited [6]. Statecharts use a syntax that is similar to ordinary state machines and supports hierarchical states through the concept of superstates, a considerably more powerful concept than the macro steps of Grafcet.

Grafchart is the name of a graphical language aimed at supervisory control applications developed at Lund University [1]. It combines the function chart formalism of Grafcet with the hierarchical states of Statecharts. It also supports parameterized function chart procedures. Through this the best concepts from both Grafcet and Statecharts are combined. JGrafchart is the name of a Java implementation of Grafchart [2]. It is a combined graphical editor and run-time system, and can be viewed as a soft-PLC. It is also possible to use JGrafchart only as a graphical editor generating executable code. In [4] code generation from JGrafchart to Modelica is presented. Code generation has also been provided to C and Java.

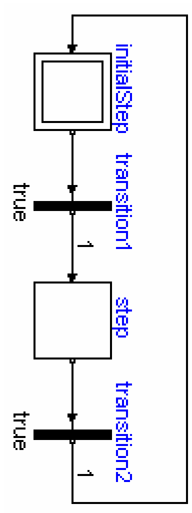
The StateGraph library is based on a subset of JGrafchart. Besides minor modifications to arrive at a suitable Modelica implementation, the essential difference is to use Modelica as an "action" language. The "single assignment rule" of Modelica makes it completely different to the action languages used in the formalisms from above. It will be shown that this has significant advantages.

2 Users View

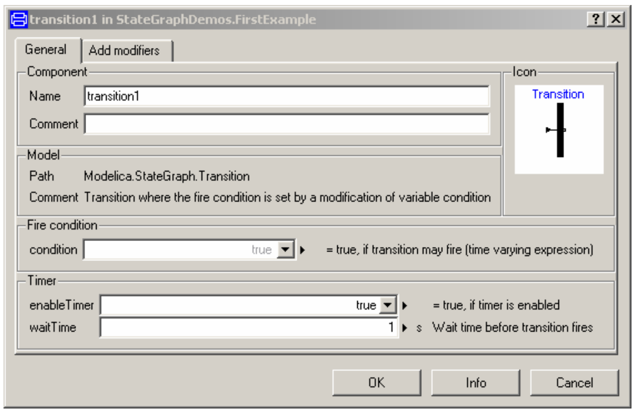
In this section the components of the StateGraph library are introduced by examples to show how it can be used in applications.

2.1 Steps and Transitions

The basic elements of StateGraphs are **steps** and **transitions** as shown in the next figure. **Steps** represent the possible states a StateGraph can have. If a step is active the Boolean variable **active** of the step is **true**. If it is deactivated, **active** = **false**. At the initial time, all ordinary steps are deactivated. The **InitialStep** objects are steps that are activated at the initial time. They are characterized by a double box (see next figure at the left).



Transitions are used to change the state of a StateGraph. When the step connected to the input of a transition is active, the step connected to the output of this transition is deactivated and when the transition condition becomes true, then the transition fires. This means that the step connected to the input to the transition is deactivated and the step connected to the output of the transition is activated. The transition **condition** is defined via the parameter menu of the transition object. Clicking on object "transition1" in the above figure, results in the following menu:



transition1 in StateGraphDemos.FirstExample

General | Add modifiers

Component
Name: transition1
Comment:

Icon: Transition

Model
Path: Modelica.StateGraph.Transition
Comment: Transition where the fire condition is set by a modification of variable condition

Fire condition
condition: true — = true, if transition may fire (time varying expression)

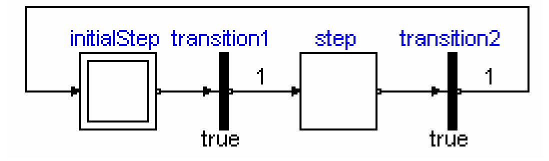
Timer
enableTimer: true — = true, if timer is enabled
waitTime: 1 s — Wait time before transition fires

OK | Info | Cancel

In the input field "**condition**", any type of time varying Boolean expression can be given (in Modelica notation, this is a modification of the time varying variable **condition**). Whenever this condition is true, the transition can fire. Additionally, it is possible to activate a timer, via **enableTimer** (see menu above) and provide a **waitTime**. In this case the firing of the transition is delayed according to the defined waitTime. The transition only fires if the condition remains true during the waitTime. The transition condition and the waitTime are displayed in the transition icon.

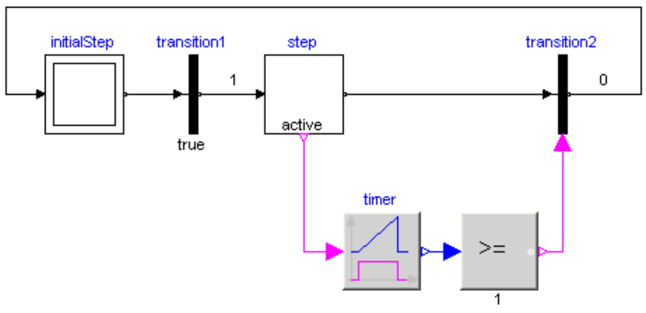
In the above example, the simulation starts at **initialStep**. After 1 second, **transition1** fires and **step1** becomes active. After another second **transition2** fires and **initialStep** becomes again active. After a further second **step1** becomes active, and so on.

In Grafchart, Grafcet and SFC the network of steps and transitions is drawn from top to bottom. In StateGraphs, no particular direction is defined, since Modelica models do not depend on the placement of components and connection lines. Usually, it is more practical to define the network from left to right, since it is easier to read the labels on the icons. The example from above has then the following layout:

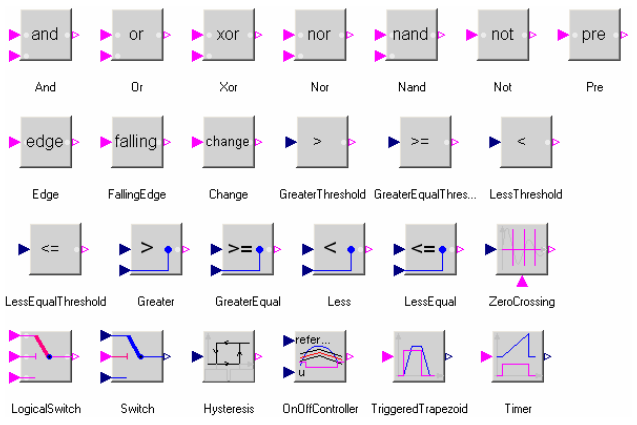


2.2 Conditions and Actions

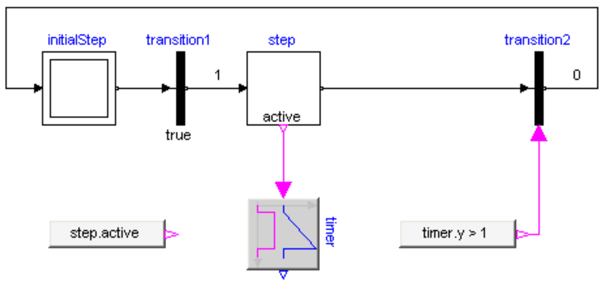
With the block **TransitionWithSignal**, the firing condition can be provided as Boolean input signal, instead as entry in the menu of the transition with block Transition, see example in the next figure:



Component "step" is an instance of "StepWithSignal" that is a usual step where the active flag is available as Boolean output signal. To this output, component "Timer" from library "Modelica.Blocks.-Logical" is connected. It measures the time from the time instant where the Boolean input (i.e., the active flag of the step) became true up to the current time instant. The timer is connected to a comparison component. The output is true, once the timer reaches 1 second. This signal is used as condition input of the transition. As a result, "transition2" fires, once step "step" has been active for 1 second. Of course, any other Modelica block with a Boolean output signal can be connected to the condition input as well, especially blocks of the Modelica.Blocks.-Logical library, see next figure. The Logical library will be extended in the future. It is also easy for a user to define his own, specialized logical blocks.



Instead of using logical blocks, via the Modelica.Blocks.Sources.SetBoolean component any type of logical expression can be defined in textual form, as shown in the next figure:



With the block "**SetBoolean**", a time varying expression can be provided as modification to the output signal **y** (see block with icon text "timer.y > 1" in the figure above). The output signal can be in turn connected to the condition input of a TransitionWithSignal block.

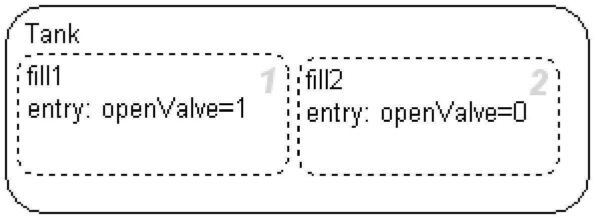
The "**SetBoolean**" block can also be used to compute a Boolean signal depending on the active step. In the figure above, the output of the block with the icon text "step.active" is true, when "step" is active, otherwise it is false (note, the icon text of "SetBoolean" displays the modification of the output signal "y"). This signal can then be used to compute desired **actions** in the physical systems model. For example, if a **valve** shall be open, when the StateGraph is in "step1" or in "step4", a "SetBoolean" block may be connected to the valve model using the following condition:

```
      step1.active or step2.active
```

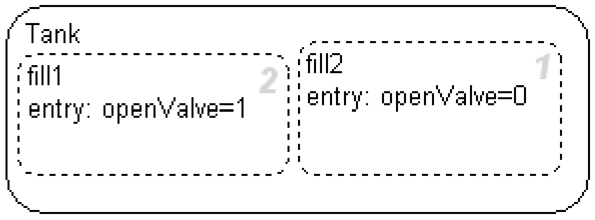
Via the Modelica operators **edge**(..) and **change**(..), conditions depending on rising and falling edges of Boolean expressions can be used when needed.

In Grafchart, Grafcet, SFC and Statecharts, **actions** are formulated **within a step**. Such actions are distinguished as **entry**, **normal**, **exit** and **abort** actions. For example, a valve might be opened by an entry action of a step and might be closed by an exit action of the same step. In StateGraphs this is **not possible** due to Modelicas "single assignment rule" that requires that every variable is defined by exactly one equation. Instead, the approach explained above is used. For example, via the "SetBoolean" component, the valve variable is set to true when the StateGraph is in particular steps.

This feature of a StateGraph is very useful, since it allows a Modelica translator to **guarantee** that a given StateGraph has always **deterministic** behaviour without conflicts. In the other methodologies this is much more cumbersome. As an example, in the next figure a critical situation in Stateflow is shown (Mathworks Stateflow is similar to a StateGraph but has, e.g., a slightly different visual appearance, and is integrated in Mathworks Simulink):



The two substates "fill1" and "fill2" are executed in parallel. In both states the variable "openValve" is set as entry action. The question is whether openValve will have value 0 or 1 after execution of the steps. Stateflow changes this non-deterministic behaviour to a formally deterministic one by defining an execution sequence of the states that depends on their graphical position. The light number on the right of the states shows in which order the states are executed. In the figure above this means that "openValve=0" after leaving the two states. If the second state "fill2" is changed a little bit graphically



"openValve=1" after "fill1" and "fill2" have been executed. This is a dangerous situation because (a) slight changes in the placement of states might change the simulation result and (b) if the parallel execution of actions depends on the evaluation order, errors are very difficult to detect.

Note, similar problems occur in other StateGraph variants, SFC, Grafcet and Graphcharts: Variables are changed according to an evaluation sequence of the simulator. It seems not possible to provide an easy-to-grasp rule about evaluation order of actions that are executed in parallel. Therefore, either the simulator just uses an internal evaluation order, or non-obvious rules are present as in Stateflow that do not solve the underlying problem.

In a StateGraph, such a situation is detected by the translator resulting in an error, since there are two equations to compute one variable. The user is forced to reformulate the network by explicitly defining priorities. For example, if "fill1" and "fill2" are steps that are executed in parallel, there might be a "SetBoolean" block that defines:

```
  openValve =
      if fill1.active then 1 else
      if fill2.active then 0 else 2
```

Therefore step fill1 has a higher priority as step fill2.

In a Stategraph or Graphchart it is difficult to modularize a sub chart if the used actions reference variables in an outer scope: Assume, for example, that a state machine “control” has the following hierarchy:

```
        control.superstate1.step1
```

Within "step1" a Statechart would, e.g., access variable "control.openValve", say as "entry action: openValve = true". This typical usage has the drawback that it is difficult to use the hierarchical state "superstate1" as component in another context, because "step1" references a particular name outside of this component.

In a StateGraph, there would be typically a "SetBoolean" component in the "control" component stating:

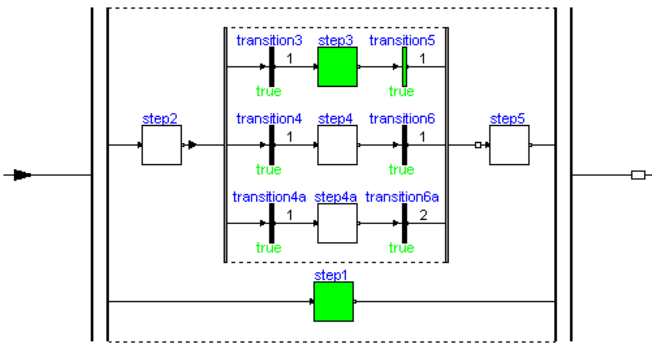
```
 openValve = superstate1.step1.active;
```

As a result, the "superstate1" component can be used in another context, because it does not depend on the environment where it is used.

The disadvantage of the StateGraph approach is that the user might not be able to formulate the network directly as desired. For example, in order to fill a tank usually several actions are necessary, e.g., to close one valve and to open another one. In a SFC all actions to “fill a tank” would be defined as actions to a “fill_a_tank” step and this might be more convenient for the user. For example, copying or deleting a “fill_a_tank” step would require only a change at one place in a SFC whereas it would require changes at several places in a StateGraph.

2.3 Parallel and Alternative Execution

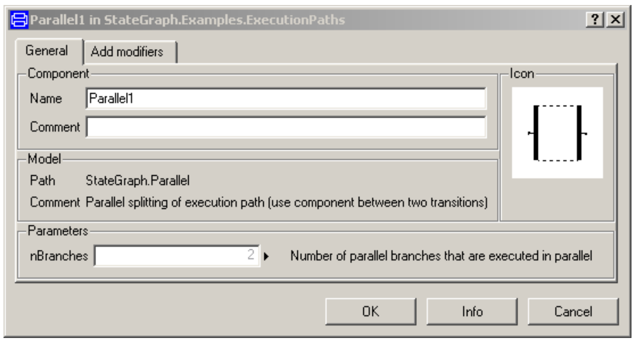
Parallel activities can be defined by component StateGraph.**Parallel** and alternative activities can be defined by component StateGraph.**Alternative**. An example for both components is given in the next figure. Here, the branch from "step2" to "step5" is executed in parallel to "step1". A transition



connected to the output of a parallel branch component can only fire if the final steps in all parallel branches are active simultaneously. The figure above is a screen-shot from the animation of the StateGraph: Whenever a step is active or a transition can fire, the corresponding component is marked in **green** color.

The three branches within "step2" to "step5" are executed alternatively, depending which transition condition of "transition3", "transition4", "transition4a" fires first. Since all three transitions fire after 1 second, they are all candidates for the active branch. If two or more transitions would fire at the same time instant, a priority selection is made: The alternative and parallel components have a vector of connectors. Every branch has to be connected to exactly one entry of the connector vector. The entry with the lowest number has the highest priority.

Parallel, Alternative and Step components have vectors of connectors. The dimensions of these vectors are set in the corresponding parameter menu. E.g. in a "Parallel" component:


Parallel1 in StateGraph.Examples.ExecutionPaths

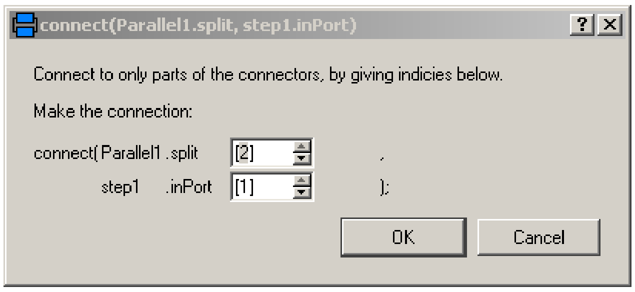
General | Add modifiers

Component
Name: Parallel1
Comment:

Model
Path: StateGraph.Parallel
Comment: Parallel splitting of execution path (use component between two transitions)

Parameters
nBranches: 2 — Number of parallel branches that are executed in parallel

Currently in the Modelica tool Dymola the following menu pops up when a branch is connected to a vector of components in order to define the vector index to


connect(Parallel1.split, step1.inPort)

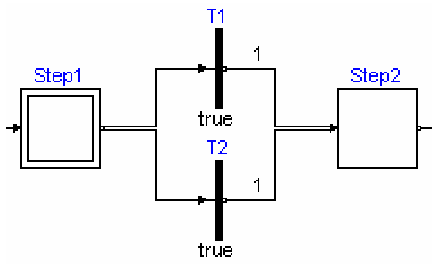
Connect to only parts of the connectors, by giving indicies below.

Make the connection:

connect(Parallel1 .split [2] ,
step1 .inPort [1]);

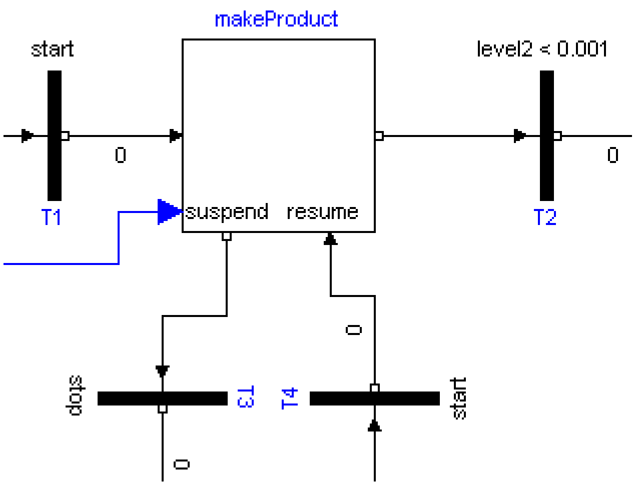
which the component shall be connected. There are discussions to improve the Modelica language to handle such situations more conveniently.

Note, alternative branches can also be defined without the “Alternative” component by just connecting several transitions to the outputs of the same step as shown in the next figure:



2.4 Composite Steps

A StateGraph can be hierarchically structured by using a component that inherits from StateGraph.**PartialCompositeStep**. An example is given in the next figure:



The CompositeStep component contains a local StateGraph that is entered by one or more input transitions and that is left by one or more output transitions. Also, other needed signals may enter a CompositeStep. The CompositeStep has similiar properties as a "usual" step: The CompositeStep is **active** once at least one step within the CompositeStep is active. Variable **active** defines the state of the CompositeStep.

Additionally, a CompositeStep has a **suspend** port. Whenever the transition connected to this port fires, the CompositeStep is left at once. When leaving the CompositeStep via the suspend port, the internal state of the CompositeStep is saved, i.e., the active flags of all steps within the CompositeStep. The CompositeStep might be entered via its **resume** port. In this case the internal state from the suspend transition is reconstructed and the CompositeStep continues the execution that it had before the suspend transition fired (this corresponds to the history ports of Statecharts or JGrafcharts).

A CompositeStep may contain other CompositeSteps. At every level, a CompositeStep and all of its content can be left via its suspend ports (actually, there is a vector of suspend connectors, i.e., a CompositeStep might be aborted due to different transitions).

The CompositeStep can be used in the same way as a superstate in Statecharts. In a superstate it is possible to enter the state in different ways ending up in different internal states. This can be modeled in a StateGraph or a Graphchart by having multiple input transitions, each leading to a different internal step. In a superstate it is possible to exit a superstate in different ways depending on which internal state that is active. This is modeled in a StateGraph or Graphchart by associating different output transitions to the different internal steps. In a superstate it is, finally, also possible to exit the state independently from which internal state that is active. This is achieved with the suspend port here. The conditions connected to the transitions attached to the suspend port can also be conditioned by the status of the internal steps of the CompositeStep. In this way it is possible to suspend the step if a certain condition holds and unless a certain internal step is active. The history arcs in Statecharts correspond to the resume port. Superstates with parallel subparts, so called XOR superstates, can be modeled using parallel constructs inside the CompositeStep.

In addition to using CompositeSteps for modeling hierarchical states they can also be used to simply aggregate a part of a larger StateGraph. This can be useful to improve the structure

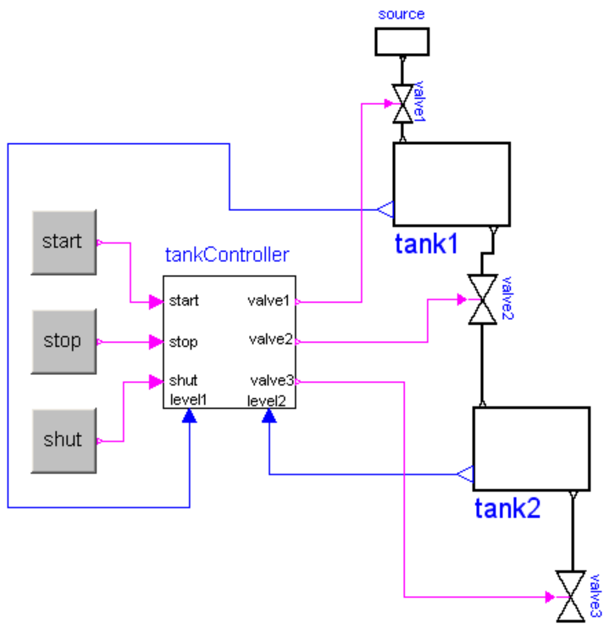
2.5 Execution Model

The execution model of a StateGraph follows from its Modelica implementation: Given the states of all steps, i.e., whether a step is active or not active, the equations of all steps, transitions, transition conditions, actions etc. are sorted resulting in an execution sequence to compute essentially the new values of the steps. If conflicts occur, e.g., if there are more equations as variables, of if there are algebraic loops between Boolean variables, an error occurs. Once all equations have been processed, the **active** variables of all steps are updated to the newly calculated values. Afterwards, the equations are again evaluated. The iteration stops, once no step changes its state anymore, i.e., once no transition fires anymore. Then, simulation continuous until a new event is triggered, i.e., when a relation changes its value.

With the Modelica "sampled(..)" operator, a StateGraph might also be executed within a discrete controller that is called at regular time instants. In a future version of the StateGraph library, this might be more directly supported.

3 Example of a Tank Controller

In this section a more realistic, still simple, application example is given, to demonstrate various features of the StateGraph library. This example shows the control of a two tank system from [4]. In the following figure the top level of the model is shown.



This model is available as Modelica.StateGraph.-Examples.ControlledTanks. In the right part of the figure, two tanks are shown. At the top part, a large fluid source is present from which fluid can be filled in **tank1** when **valve1** is open. Tank1 can be emptied via **valve2** that is located in the bottom of tank2 and fills a second **tank2** which in turn is emptied via **valve3**. The actual levels of the tanks are measured and are provided as signals **level1** and **level2** to the **tankController**.

The **tankController** is controlled by three buttons, **start**, **stop** and **shut** (for shutdown) that are mutually exclusive. This means that whenever one button is pressed (i.e., its state is **true**) then the other two buttons are not pressed (i.e., their states are **false**). The buttons could be implemented as dynamic elements that react when clicking on them. In the example, they are implemented with logical tables, i.e., block Modelica.StateGraph.Temporary.RadioButton, in order that the result of the simulation is reproducible.

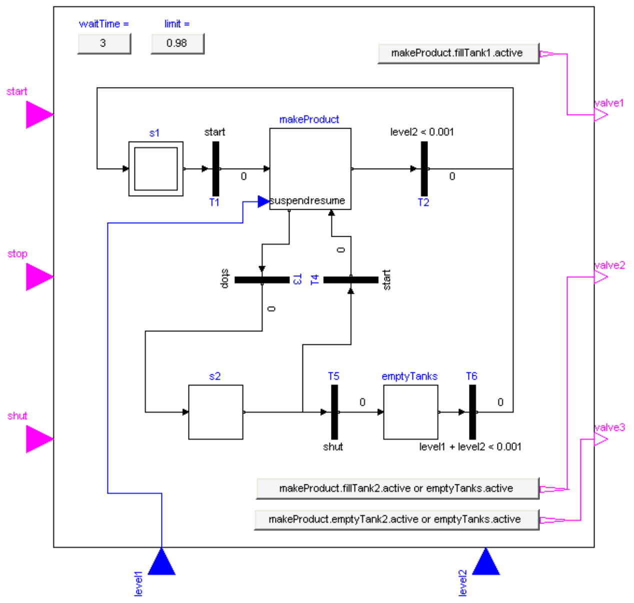
When button **start** is pressed, the "normal" operation to fill and to empty the two tanks is processed:

1. Valve 1 is opened and tank 1 is filled.
2. When tank 1 reaches its fill level limit, valve 1 is closed.
3. After a waiting time, valve 2 is opened and the fluid flows from tank 1 into tank 2.
4. When tank 1 is empty, valve 2 is closed.
5. After a waiting time, valve 3 is opened and the fluid flows out of tank 2
6. When tank 2 is empty, valve 3 is closed

The above "normal" process can be influenced by the following buttons:

- Button **start** starts the above process. When this button is pressed after a "stop" or "shut" operation, the process operation continues.
- Button **stop** stops the above process by closing all valves. Then, the controller waits for further input (either "start" or "shut" operation).
- Button **shut** is used to shutdown the process, by emptying at once both tanks. When this is achieved, the process goes back to its start configuration. Clicking on "start", restarts the process.

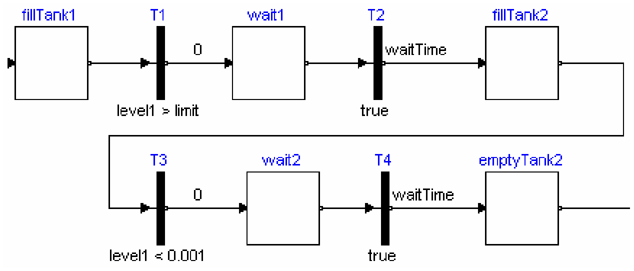
The implementation of the **tankController** is shown in the next figure. When the "**start**" button is pressed, the stateGraph is within the CompositeStep "**makeProduct**". During normal operation this CompositeStep is only left, once tank2 is empty. Afterwards, the CompositeStep is at once re-entered. When the "**stop**" button is pressed, the "makeProduct" CompositeStep is at once terminated via the "**suspend**" port and the stateGraph waits in step "**s2**" for further commands. When the "**start**" button is pressed, the CompositeStep is re-entered via its **resume** port and the "normal" operation continues at the state where it was aborted by the suspend transition. If the "**shut**" button is pressed, the stateGraph waits in the "**emptyTanks**" step, until both tanks are empty and then waits at the initial step "**s1**" for further input.



The opening and closing of valves is **not** directly defined in the StateGraph. Instead via the "**setValveX**" components, the Boolean state of the valves are determined. For example, the output y of "setValve2" is computed as:

```
y = makeProduct.fillTank2.active
    or emptyTanks.active
```

i.e., valve2 is open, when step "makeProduct.fillTank2 or when step "emptyTanks" is active. Otherwise, valve2 is closed. The main part of the composite step “makeProduct” is shown in the next figure. Step “fillTank1” is left, once the highest level



for the tank is reached (level1 > limit). The StateGraph remains in step “wait1” during the defined “waitTime”. Afterwards, step “fillTank2” remains active until tank1 is empty (level1 < 0.001). After a waiting phase, the “emptyTank2” step is entered.

4 Implementation

In this section the implementation of the most important parts of the library is sketched.

4.1 Steps and Transitions

Steps and transitions are implemented according to the method described in [7][5] to define Petri nets with an equation based language.

A transition has one inPort and one outPort connector and is basically defined by the following equations (if no timer is present):

```
fire = condition and
          inPort.available and not
          outPort.occupied;
inPort.reset = fire;
outPort.set  = fire;
```

Note, that the inPort connector of a transition consists of the Boolean variables “available” and “reset” and the outPort connector consists of the Boolean variables “occupied” and “set”. The above equation states that “fire = true”, if (1) the firing condition is true, (2) the inPort step is active and (3) the outPort step is not active. The “fire” value is reported to the two steps to which the transition is connected.

A step has a vector of input and a vector of output connectors. It is basically defined as:

```
   active = pre(newActive);
newActive = anyTrue(inPort.set) or active
            and not anyTrue(outPort.reset)
```

The function “anyTrue(..)” returns **true**, if any element of the input vector is **true**. The step becomes active in the next iteration when one of the transitions connected to the inPort connectors fires (set = true if a transition fires). The step remains active if it was active and no transition connected to the outPort connectors fires (reset = true, if a transition fires).

A step reports its active flag to the transition connected to its first outPort by the equation:

```
outPort[1].available = active;
```

In order to make sure that only one of the transitions connected to the outPorts can fire, the active flag is hidden to the second outPort transition if the first transition decides to fire and sends a reset condition:

```
outPort[2].available =
       outPort[1].available and not
       outPort[1].reset;
```

The general case can be written in Modelica as

```
for i in 1:size(outPort,1) loop
  outPort[i].available =
     if i == 1 then active else
       outPort[i-1].available and not
       outPort[i-1].reset;
end for;
```

A step needs to signal to its inPort transitions whether it is possible to activate it or whether it is about to become active via transitions with higher priorities. This is described as

```
for i in 1:size(inPort,1) loop
  inPort[i].occupied =
    if i == 1 then active else
      inPort[i-1].occupied or
      inPort[i-1].set;
end for;
```

The inPort and outPort connectors contain appropriate “input” and “output” prefixes of the connector variables, in order that steps can only be connected to transitions and vice versa. Furthermore, the annotation “Hide = **true**” is set on all connector variables, in order that these variables do not show up in the plot browser, because these are internal variables that are of no interest for the user of the StateGraph library.

In a parameter menu of a component usually only variables are displayed that are declared as parameters. In the parameter menu of a transition, additionally the time varying variable “condition” is displayed as shown in section 2.1. This is implemented by adding the annotation “Dialog” to the variable declaration:

```
Boolean condition annotation(Dialog);
```

Usually, the “Dialog” annotation has additional subentries, such as “group” or “tab”. However, if no subentries are present, this annotation just means to include the variable in the parameter menu.

In a JGraphchart there is a timer associated with every step by providing the time difference between the actual time and the time when the step became active via variable “t”. In a StateGraph no time variable is associated with a step, but an optional timer is provided in a transition and via the connector "active" of a step a timer from the Logical library can be attached to the step. This provides similar functionality as for a JGraphchart. One reason for this change was to improve the efficiency. For example, in a transition the following code fragment to define a timer is present:

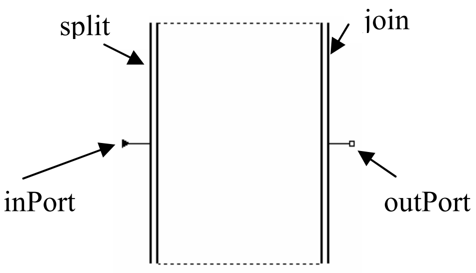
```
if enableTimer then
  when enableFire then
    t_start = time;
  end when;
  t_dummy = time - t_start;
  t = if enableFire then t_dummy else 0;
  fire = enableFire and
         time >= t_start + waitTime
else
  ...
end if;
```

A Modelica translator triggers an event when time reaches "t_start + waitTime". Since "t_start" is a variable that is set in the same scope in a when clause and "waitTime" is a parameter, a Modelica translator can easily trigger a **time event**.

The situation is different, if the when clause "**when** enableFire **then** t_start = time; **end when**" is present within a step and the relation "time >= t_start + waitTime" is present in another component, e.g., in a "condition" of a transition. A Modelica translator will then usually trigger a **state event** because in the scope of the relation it is not known that "t_start" can change its value only at event instants.

4.2 Parallel and Alternative Execution

The parallel component has the following icon



and consists of 4 connectors. The "inPort" and outPort" connectors allow only a connection to transitions. The "split" and "join" connectors are vectors of connectors that are drawn in a quite "lenghty" format to resemble the usual visual layout of parallel execution in SFC. They allow only a connection to steps. After dragging this icon in a model, it is usually enlarged until the desired elements can be placed between the "split" and the "join" connectors.

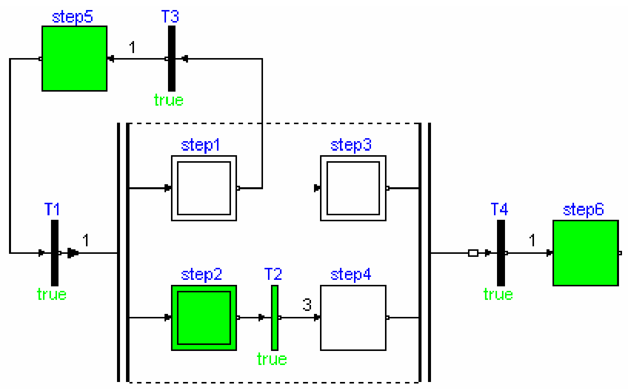
Besides appropriate "assert" statements to guarantee the desired connection structure, the Parallel component consists of the following equations only:

```
n = size(split,1);
split.set  = fill(inPort.set, n);
join.reset = fill(outPort.reset,n);
inPort.occupied  =anyTrue(split.occupied);
outPort.available=allTrue(join.available);
```

The second and third equation report the "set" and "reset" flags of the inPort and outPort connectors to the "split" and "join" connectors. The two last equations perform the synchronization of the parallel branches: Via function "anyTrue(..)" it is defined that the input transition can only fire if none of the steps connected directly to the "split" connector array is active. Via function "allTrue(..)" it is defined that the output transition can only fire if all steps connected directly to the "join" connector array are active.

The implementation of the "Alternative" component is performed in a similar way.

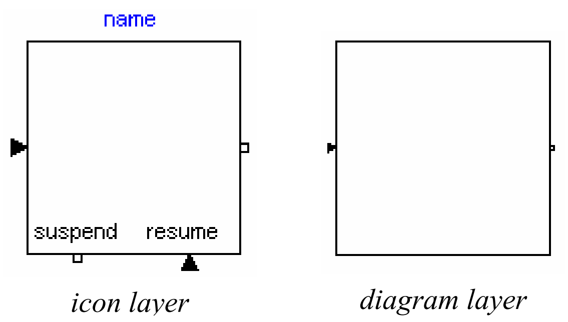
Both the Parallel and the Alternative component have the (slight) disadvantage that they can be misused. For example, in a Parallel Component it is possible to connect from a step in the parallel branches to a transition that is connected to a step outside of the Parallel component, see the example in the next figure:



It would be desirable to prevent such types of networks in a StateGraph. However, it seems not possible to formulate a corresponding restriction with the Modelica language. There are currently Modelica scripting functions under development that allow to traverse a Modelica model and extract information about the model. It might be that such functionality will allow to detect such undesirable networks. These types of function charts are also known as *unsafe* or *unreachable*. In commercial SFC editors it is common that the editor makes it impossible to enter these types of charts, rather than including these global constraints in the language itself.

4.3 Composite Steps

A composite step is a model that extends from PartialCompositeStep. The icon and diagram layer of this superclass is shown in the next figure:



There is one default "inPort" and "outPort" connector on the left and right side. More connectors to enter and leave a composite step may be added. In the icon layer a vector of "suspend" and a vector of "resume" connectors is present. These connectors are not visible in the diagram layer and therefore it is in the graphical editor not possible to connect a component in a composite step to them. The "suspend" and "resume" connector instances are not visible in the diagram layer of a composite step, because the underlying connector classes have an empty diagram layer.

A composite step is active, if at least one step in the composite step is active, and a composite step is deactivated, and also all steps in the composite step, if a transition fires that is connected to one of the "suspend" connectors. This means a communication channel between a composite step and all steps within a composite step is necessary. This is implemented by having a connector

```
connector CompositeStepStatePort
  Boolean suspend;
  Boolean resume;
  flow Real activeSteps;
end CompositeStepStatePort;
```

and use an inner definition of this connector in PartialCompositeStep:

```
inner CompositeStepStatePort root;
   ...
activeSteps = -integer(root.activeSteps);
root.suspend = anyTrue(suspend.reset);
root.resume  = anyTrue(resume.set);
newActive = activeSteps > 0 and not
            anyTrue(suspend.reset) or
            anyTrue(resume.set);
active    = pre(newActive);
```

Via flow variable "activeSteps in the inner root connector, the number of active steps is reported from the steps to the composite step. The composite step is active if this number is greater than zero and no transition at the suspend connector fires ("anyTrue(suspend.reset)") or a transition at one of the "resume" connectors fires. The information about the "suspend" and "resume" connector settings are reported to the steps inside the composite step again via the inner root connector.

In a step, a corresponding "outer" declaration of connector "root" is present and the code of section 4.1 of a step is slightly changed to:

```
protected
  outer CompositeStepStatePort root;
  CompositeStepStatePort localRoot;
equation
  connect(localRoot, root);

  localRoot.activeSteps =
                 if active then 1 else 0;
  active    = pre(newActive);
  newActive =
     if localRoot.resume then oldActive
     else (anyTrue(inPort.set) or
           active and not
           anyTrue(outPort.reset))
          and not root.suspend;

  when localRoot.suspend then
    oldActive = active;
  end when;
```

Via outer flow variable activeSteps, the active setting is reported to the composite step. Additionally, a memory is introduced via variable "oldActive" to remember the current value of the "active" flag when the composite step is terminated via its "suspend" port ("**when** localRoot.suspend **then** ..."). The assignment to "newActive" is slightly changed to include the transitions via the "suspend" and "resume" connectors in the composite step.

A composite step may contain not only steps but other composite steps. The implementation above does not handle this case. In fact, with the Modelica language version 2.1 it is not possible to provide a proper implementation. Therefore, an extension was needed that is defined in the coming version 2.2 of the Modelica language (it is already supported in Dymola):

In a composite step a construct of the following form would be needed:

```
// wrong Modelica code
inner CompositeStepStatePort root;
outer CompositeStepStatePort root;
```

where the "inner root" connector is used in all steps inside the current composite step and the "outer root" connector refers to the composite step outside of the current scope in order to have a communication channel to the outside scope. However, this is wrong Modelica code because there are two declarations with the same name. Note, the names must be the same, because in a step a communication channel to the “nearest” composite step is needed and the name used in the “outer” declaration of a step must be identical to the name used in the “inner” declaration of a composite step.

In the Modelica language version 2.2 the following extension was introduced:

```
// Modelica 2.2 code
inner outer CompositeStepStatePort root(..)
```

to define actually a new “inner” variable “root” and at the same time reference an “outer” variable “root”. **References** to “root” inside the current scope, references the “**outer**” variable. **Modifications** to “root” are not allowed for “outer” variables and therefore apply to the “**inner**” variable. In other words, inside a composite step the “outer root” is accessed by variable “root” and settings for the “inner root” have to be performed via a modification in the declaration of “root”.

The previous code fragments must be slightly modified to include the new “inner outer” declaration, and to handle the case of composite steps that are inside and/or outside the current one.

5 Summary

The free Modelica.StateGraph library offers new features to conveniently define discrete event and reactive systems in Modelica models. Since Modelica is used as an action language, a Modelica translator can guarantee that a StateGraph has deterministic behaviour. StateGraph models can be combined with components of any other Modelica library and can therefore be very easily used to control a continuous plant.

StateGraph is based on Grafchart, which contains several features that not, so far, have been implemented in StateGraph. Some of these features, such as function chart procedures, assume support for dispatching at run-time, which does not match well with the philosophy of Modelica. Other features such as lists could very well be included in StateGraph.

It is also planned to improve the graphical handling of StateGraphs in the future and to add more functionality especially also to the Modelica.Blocks.-Logical library that is often used in a StateGraph. Improvement suggestions and contributions are welcome.

References

[1] Årzen K.-E. (1996): Grafchart: **A Graphical Language for Sequential Supervisory Control Applications**. IFAC'96, Preprints 13th World Congress of IFAC, San Francisco.

[2] Årzen K.-E., Olsson R., and Akesson J. (2002): **Grafchart for Procedural Operator Support Tasks**. Proceedings of the 15th IFAC World Congress, Barcelona, Spain.

[3] David R., and Alla H. (1992): **Petri Nets and Grafcet: Tools for modeling discrete event systems**. Prentice Hall.

[4] Dressler I. (2004): **Code Generation from JGrafchart to Modelica**. Master Thesis, ISRN LUTFD2/TFRT-5726-SE, Department of Automatic Control, Lund Institute of Technology, Sweden.

[5] Elmqvist H., Mattsson S.E., and Otter M. (2001): **Object-Oriented and Hybrid Modeling in Modelica**. Journal Européen des systèmes automatisés, vol. 35, no. 1, pp. 1-22.

[6] Harel D. (1987): **Statecharts: A Visual Formalism for Complex Systems**. Science of Computer Programming, Vol. 8.

[7] Mosterman P.J., Otter M., and Elmqvist H. (1998): **Modeling Petri Nets as Local Constraint Equations for Hybrid Systems Using Modelica**. In Proceedings of SCS Summer Simulation Conference, pp. 314-319, Reno, Nevada, July 1998.