# A New Formalism for Modeling of Reactive and Hybrid Systems

Martin Otter[1], Martin Malmheden[2], Hilding Elmqvist[2], Sven Erik Mattsson[2], Charlotta Johnsson[3]
[1]German Aerospace Centre (DLR), Institute for Robotics and Mechatronics, Germany
[2] Dassault Systèmes, Lund, Sweden (Dynasim)
[3] Department of Automatic Control, Lund University, Sweden
Martin.Otter@dlr.de, Martin.Malmheden@3ds.com, Hilding.Elmqvist@3ds.com,
SvenErik.Mattsson@3ds.com, Charlotta.Johnsson@control.lth.se

## Abstract

A new Modelica library is presented that is used to model safe hierarchical state machines in combination with any Modelica model, e.g., controllers, logical blocks, and physical systems described by differential-algebraic equations. It has been designed to simplify usage, improve safety aspects and to harmonize with the design of the new Modelica_EmbeddedSystems library. Furthermore, new blocks are introduced to define actions in a visual way, and not textually. The library is inspired by Statecharts, Sequential Function Charts, Safe State Machines (SSM) and Mode-Automata. It has been designed so that only small extensions to Modelica 3.1 are needed. The algorithms are sketched that are used to guarantee consistent graphs that give a limited number of event iterations. Furthermore, it is shown how a symbolic verifier can be used to guarantee additional properties of state machines.

*Keywords: ModeGraph; Statechart, Sequential Function Charts, Mode-Automata, Safe State Machines; NuSMV; reactive systems, hybrid systems.*

## 1 Introduction

In this article the open source Modelica_StateGraph2 library is presented. This is version 2 of the existing Modelica.StateGraph library *(Otter et. al. 2005)*. It is planned to replace Modelica.StateGraph in one of the next releases of the Modelica Standard Library with Modelica_StateGraph2. Besides the basic Step and Transition mechanism, all other parts have been redesigned and significantly improved based on the experience with the experimental ModeGraph library *(Malmheden et. al. 2008)*. Note, below the name "StateGraph" is often used as abbreviation for the full name "Modelica_StateGraph2".

The StateGraph library is inspired by Statecharts *(Harel 1987)*, Sequential Function Charts (SFC), Safe State Machines (SSM) *(André 2003)*, and Mode-Automata *(Maraninchi and Rémond 2002*). The primary purpose of the library is to provide support for modeling of reactive and of hybrid systems and to verify certain properties of such systems.

Reactive systems react to stimuli from their environment, see, e.g. *(Benveniste et. al. 2003)*. In combination with the Modelica_EmbeddedSystems library *(Elmqvist et. al. 2009)*, the StateGraph library can be used to model such systems and it will be possible to use StateGraph models in production code of embedded systems.

Hybrid systems combine closely continuous-time models and discrete event systems, see, e.g. *(Lynch 2002)*. The StateGraph library is implemented with the Modelica language and therefore every Modelica model, i.e., models consisting of differential, algebraic and discrete equations, as well as functions, can be conveniently and naturally combined with state diagrams constructed with the StateGraph library.

## 2 Using Modelica_StateGraph2

In this section an overview is given of how to use the library by several small examples.

### 2.1 StateGraph Elements

A StateGraph graph is constructed by three elements: Step, Transition, and Parallel that will now be discussed in some detail.

**Step**
A Step is the graphical representation of a state and is said to be either active or not active. A StateGraph



DOI: 10.3384/ecp09430108

model is comprised of one or more steps that may or may not change their states during execution. A StateGraph model must have one initial step. An initial step is defined by setting parameter initialStep at one step to true. The initial step is visualized by a small arrow pointing to this step, see Step s1 in **Figure 1**.

**Transition**

To define a possible change of states, a Transition is connected to the output of the preceding Step and to the input of the succeeding Step, see, e.g., **Figure 1**, where Transition t1 defines the transition from Step s1 to Step s2. Note: A Transition has exactly one preceding and one succeeding Step. A Transition is said to be enabled if the preceding step is active. An enabled transition is said to be fireable when the Boolean condition defined in the parameter menu of the transition is evaluated to true. This condition is also called "Transition condition" and is displayed in the icon of the Transition. When parameter "use_conditionPort" is set, the Transition condition is alternatively defined by a Boolean signal that is connected to the enabled "conditionPort". A fireable transition will fire immediately. In **Figure 1**, t1 fires when s1 is active and time is greater than one.

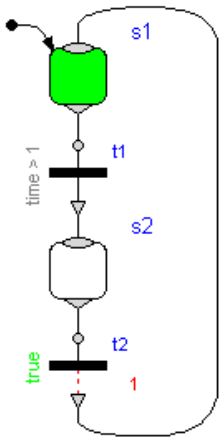



**Figure 1:** Model with two steps and two transitions

The firing of a transition can optionally also be delayed for a certain period of time. See, e.g., t2 in **Figure 1**, that is delayed for one second before it may fire, given that the condition remains true and the preceding Step remains active during the entire delay time. The evolution of a graph over time can be visualized by diagram animation: Active steps and Boolean variables that are true are marked in green here, see, e.g., **Figure 1**.

**Parallel**

Subgraphs can be aggregated into superstates by using the Parallel component. This component acts both as a composite step (having just one branch) and as a step that has parallel branches. The Parallel component, often referred to as "p" in the following figures, allows the user to place any StateGraph element inside it, especially Steps, Transitions, and Parallel components.

A Parallel component has always an entry port, see **Figure 2**, and it may have optionally an exit port. All branches in a Parallel Component must start at the entry port and at least one must terminate at the exit port, provided the exit port is enabled via parameter "use_outPort". If a Parallel component shall be entered from the outside via a Transition, parameter "use_inPort" must be set to true, to enable an input port. If a Parallel Component shall be left via a transition to an outside step, parameter "use_outPort" must be set to true, to enable the output and the exit port. A Parallel component may be used as initial step, by setting parameter initialStep to true. This property is again visualized by a small arrow pointing to the Parallel component, see **Figure 2**.

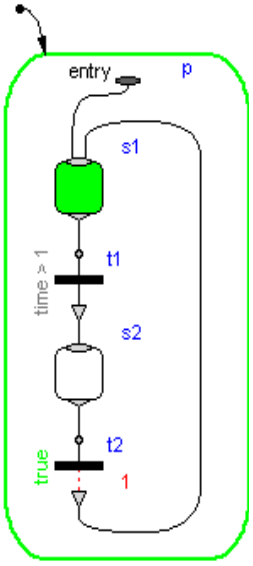



**Figure 2**: A Parallel component with a small sub-system.

A Parallel component may be suspended and subsequently resumed. In **Figure 3**, Transition T6 fires whenever the input signal u is true, suspending the Parallel component p and the enclosed Steps s2, s3,

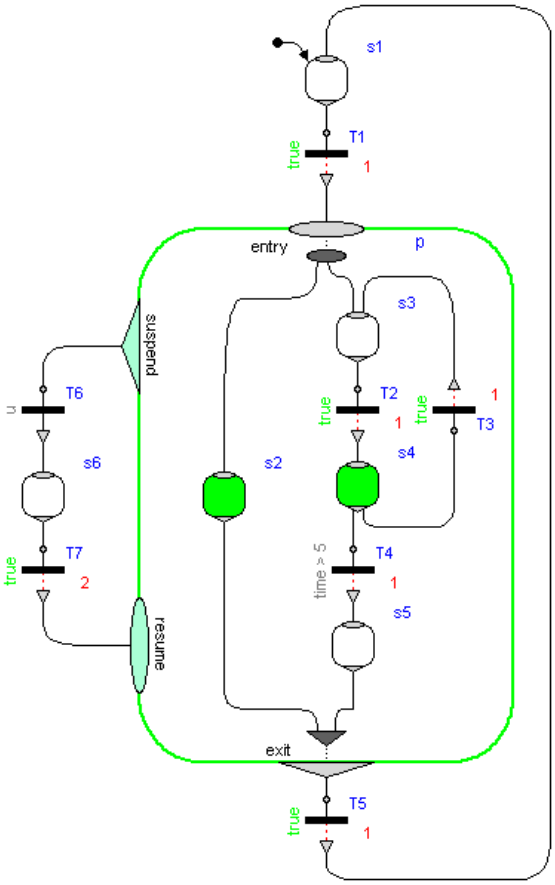



**Figure 3:** Parallel component with 2 parallel branches that is suspended whenever the input u is true.

s4 and s5 for two seconds. When Transition T7 fires, p is re-activated in the same state as when it was suspended.

As mentioned before, inPorts and outPorts of a Parallel component are optional and can be set by the user. If the parallel component has an inPort, then the entry port constitutes the connection between the Transition connected to the inPort and the first Steps to be activated in the Parallel component. If the Parallel component is configured to have an outPort, an exit port shows up on the bottom of the Parallel component, see **Figure 3**.

The Parallel component allows the entry port to branch out into several parallel paths. These branches are not allowed to interact, see **Figure 3**. When all Steps connected to the exit port are active, the Parallel component is said to be available and may exit when the Transition connected to the outPort fires. In **Figure 3** Transition T5 fires when both Step s2 and s5 have been active together for one second and thereby deactivates the Parallel component p. Note, in Statecharts parallel branches must be synchronized via transition conditions, which is inconvenient. In SFC, all branches are synchronized. In StateGraph, only branches that are connected to the exit port are synchronized, which is more flexible as the SFC approach.

No component contained within the Parallel component may be connected to any other component "outside" of the Parallel component. This rule is used to protect the user from making mistakes that could lead to unexpected results and states of the graph that are not well-defined. Consider for example the graph in **Figure 4Figure** at T=7. Especially, note that the Parallel component p is never properly terminated through either an outPort or a suspend port. If the graph would be allowed to execute, the consequence would be an increasing number of active Steps. Such a situation is reported as an error. The details about the algorithm to accomplish this are given in appendix A2.

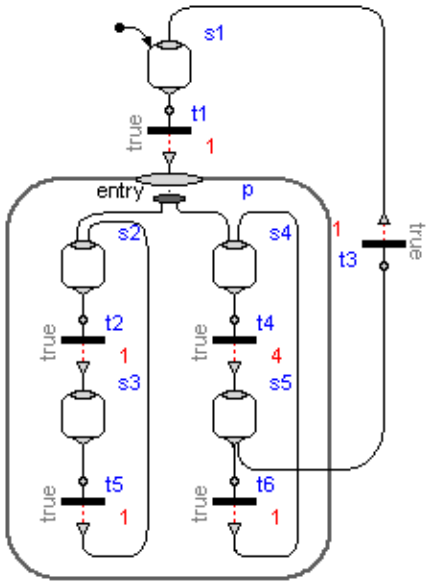



**Figure 4**: Wrong graph since components in the Parallel component may not be connected to "outside" ones.

In order to graphically organize large graphs in different levels of hierarchy and with encapsulation of variables, StateGraph also contains a component PartialParallel. It is similar to the normal Parallel component but introduces a new hierarchy once the user inherits from it. A number of large subsystems can thus be abstracted into composite steps to improve organization and overview of the subsystems. **Figure 5** shows a component built from a PartialParallel component. As the diagram and the icon layer of the PartialParallel component does not need to be the same size, the user can benefit from collecting large subsystems in smaller closed Parallel components to improve overview and modularization of the full system.

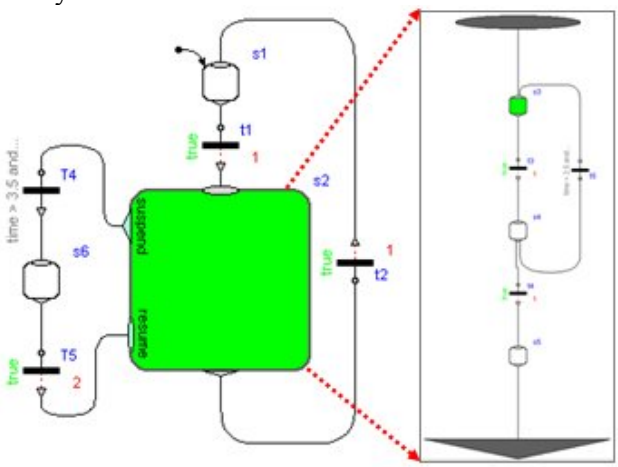



**Figure 5:** Composite derived from PartialParallel component and its subsystem.

## 2.2 Graphical Action Blocks

An important practical aspect of state machines is the ability to assign values and expressions to variables depending on the state of the machine. In StateGraph, a number of graphical components, see **Figure 6**, have been added to facilitate usage in a safe and intuitive way. Since these are just input/output blocks and will also be useful in another context, it is planned to add them to the Modelica Standard Library under "Modelica.Blocks". Some of these blocks will be explained in this section.

There are a number of standard blocks with common operations/displays of the three basic types (Boolean, Integer, Real) using vector input connectors which enables them to be connected to an arbitrary number of sources. Resizing a vector port and

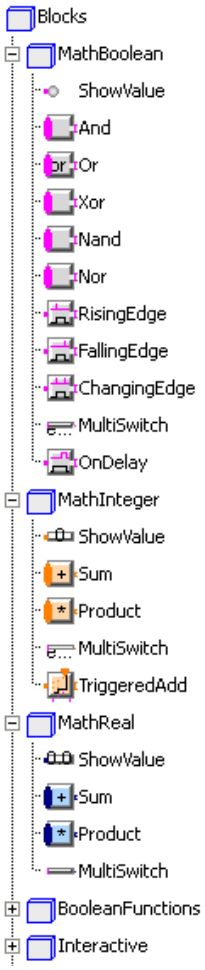



**Figure 6:** Blocks to define actions graphically.

connecting to the next free element is performed automatically when connecting to the connector, see Appendix A4. So this is much more convenient than with the Modelica.Blocks.

Logical, Modelica.StateGraph or ModeGraph libraries. A vector of input connectors is visualized as an ellipse, see, e.g., the violet connector on the left side of the "**and**" block in the figure to the right where "`y = u[1] and u[2]and ...`".


A MultiSwitch block selects one of n expressions depending on an array of Boolean inputs. The index of the first input in the Boolean array that is true defines the index of the expression in the expression array to be used as the scalar output y. In **Figure 7**, the MultiSwitch component will output the value y = 1 if Step s1 is active, and will output y = 2 if s2 is active as the expression array is defined as {1,2}. If none of the Boolean array inputs is true, the "**else**" value will be used instead that is defined in the parameter menu of the MultiSwitch component and is displayed below the icon. Consider **Figure 7** when Step s3 is active – this will set the output of component "multiSwitch" to the "else" value "3". Alternatively, in the parameter menu of the MultiSwitch component it can be defined to keep its previous value, i.e. y = **pre**(y). If this option would be selected for **Figure 7**, then multiSwitch.y = 2 when Step s3 is active.

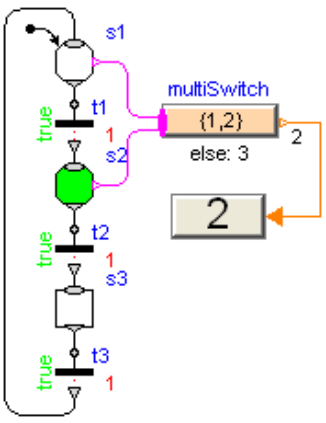



**Figure 7:** Example of MultiSwitch component for Integer numbers that depends on different steps.

The MultiSwitch block is inspired by "Modes" from Mode Automata *(Maraninchi and Rémond 2002)*: Variable multiSwitch.y has always a unique value, and this value depends on the expressions that are associated with mutually exclusive active steps. The advantages of MultiSwitch are that (1) the definition is performed in a purely graphical way, (2) it can also be used for mutually non-exclusive active steps[1], and (3) it can be implemented in Modelica in a very simple way. The drawback is that the expressions in the MultiSwitch block might no longer be so easy associated with Steps, compared to the alternative where the expressions are defined directly in the respective Steps. This latter approach would, however, require non-trivial extensions to the Modelica language.

The RisingEdge, FallingEdge and ChangingEdge components can be used to generate "pulse" signals depending on the rising, falling or changing values of Boolean signals. An example is shown in **Figure 8** where the Boolean indicator lamp is turned on when Step s2 becomes active and is turned off when Transition t3 fires and Step s3 becomes inactive. Two variants are shown to utilize the "rising" property of a Boolean signal: The Boolean connectors at steps and transitions can be activated via parameters "use_activePort" and "use_firePort", respectively. If s2 becomes active, rising = true and therefore multiSwitch.y = true. If transition t3 fires, t3.firePort=true and therefore multiSwitch.y = false.

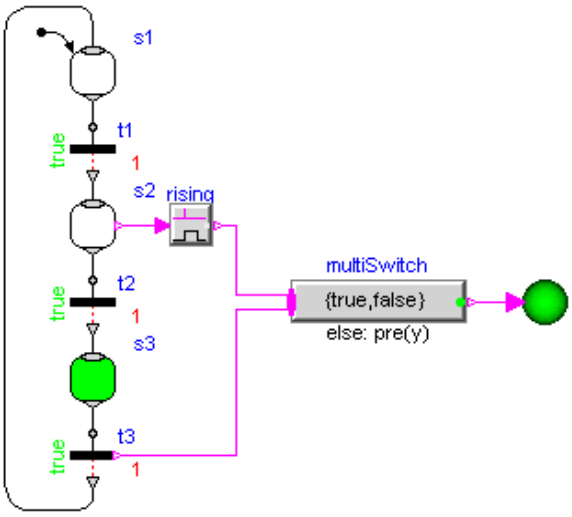



**Figure 8:** Two variants to control a Boolean indicator by a MultiSwitch component.

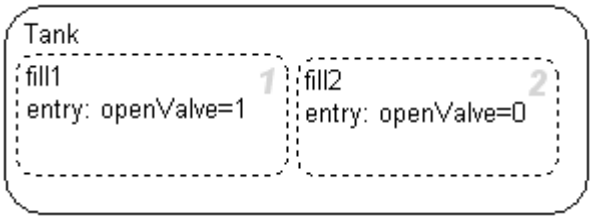
[1] If an MultiSwitch block is connected to steps of different branches of a Parallel component, a priority is present: If several inputs are true, then the one has highest priority that is connected to the lowest index of the vector of input connectors (= connection line "closest" to the icon name).

### 2.3 Safe StateGraph models

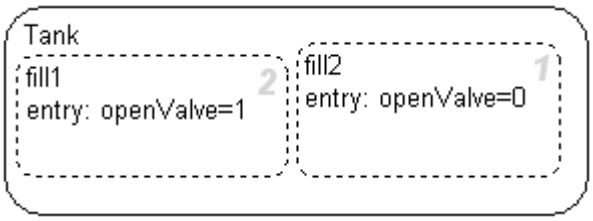
In this section it will be discussed in which sense "StateGraph" models are "safe".

**Only valid graph structures are accepted**
Contrary to Modelica.StateGraph (version 1 of the library which is distributed with the Modelica Standard Library since 2004), only valid graph structures are accepted for the Modelica_StateGraph2 library. For example, the model of **Figure 4** leads to an error. In order that this was possible, Modelica 3.1 had to be enhanced slightly. Details are given in section A2.

**One variable is defined by one equation**
In all state machine formalisms problems are present when assignments to the same variables are performed in branches that are executed in parallel. As an example, in the next figure such a situation in Stateflow *(StateFlow 2009)* is shown:




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 behavior to a formally deterministic one by defining an execution sequence of the states that depends on their graphical position. The grey 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 critical situation because (a) slight changes in the graphical positioning of states might change the simulation result and (b) if the parallel execution of actions depends on the evaluation order, errors are difficult to detect.

In StateGraph such a situation is not possible. The reason is that StateGraph is implemented in Modelica and a very basic feature of Modelica is that every declared unknown variable must be defined by exactly one equation. This is sometimes called "single assignment rule". It is therefore not possible to assign the same variable twice in a model. The above situation would be described in StateGraph instead with a MultiSwitch action block "openValve" as shown in **Figure 9**. Here, everything is well defined: There are two input connections to the openValve block. If both become true at the same time instant, the connection with the "lowest" index (i.e., the upper signal in the figure) has highest priority. Therefore, openValve gets the value true, once the Parallel component is entered.

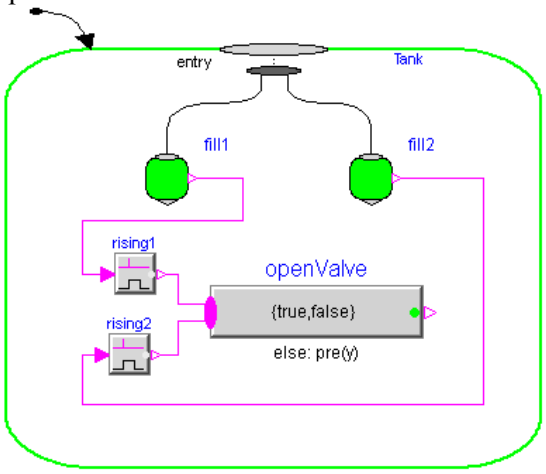



**Figure 9:** Assignment of variables with forced priority due to Modelicas single assignment rule.

**Upper bound on number of model evaluations**
At an event instant, an event iteration occurs, due to the Modelica semantics (= whenever a new event occurs, the model is re-evaluated). This means that Transitions keep firing along a connected graph, as long as the firing conditions are true. The question therefore arises, whether infinite event looping is possible? A simple example of this kind is shown in **Figure 10**. Here, all Transition conditions are true and therefore all Transitions fire forever. This is no valid StateGraph model and will result in an error.

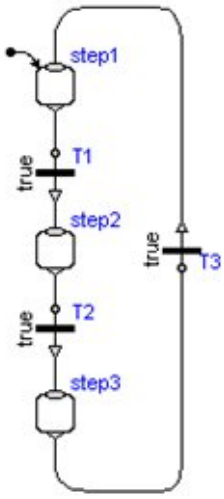



**Figure 10:** Wrong graph that gives rise to infinite looping.

In order to avoid a situation as in **Figure 10**, it is required that a StateGraph model has at least one delayed Transition per loop, see Appendix 0. This means that one of T1, T2, or T3, must be a delayed Transition, otherwise an error occurs. Since event iteration stops at a delayed Transition, infinite event looping cannot occur. This also means that at one time instant every Transition can fire at most once and therefore the

number of model evaluations at an event instant is bounded by the number of Transition components.

It is still possible that infinite event looping occurs due to model errors in other parts of the model. For example, if a user introduces an equation of the form "J = pre(J) + 1" outside of a when-clause, event iteration does not stop. Although this situation is not completely satisfactory, it helps already a lot if a tool points out potential problems of a StateGraph model, in case delayed transitions are missing.

# 3 Application Examples

In this section some involved application examples are shown to demonstrate the usage of the StateGraph library. These and other examples are available in the library under "Examples.Applications".

## 3.1 Harels wristwatch

When presenting the Statecharts formalism in *(Harel 1987)*, David Harel identified and described the behavior of his Citizen Quartz Multi-Alarm III wristwatch (see schematic figure to the right) using the new visual formalism as a case study to proof his new formalism to be flexible enough to describe the intricate structure of the wristwatch behavior in a comprehensible and clean way. As the wristwatch example serves as a challenging benchmark for the capabilities of a graphical formalism, it has been included as an application example in the StateGraph library to demonstrate that the library is flexible enough to realize this example in a good way. It also serves as a template for other human interfaces. For example, an automotive cruise control has several switches and some of them have different levels. There are different influences if in cruise mode or not.


The wristwatch display is comprised of a number of different display modes showing the current time (displayed in either 12h or 24h mode), time setting (also in either 12h/24h mode), date/date setting (day, month, day of week, year etc.), alarm setting, chime setting, and a stopwatch display. The stopwatch can be turned on, off, stalled when running to show lap time and reset when stopped. The chime functionality is triggered each time the clock reaches a whole hour that makes the chime beep for two seconds. Furthermore, the wristwatch has two concurrently running alarms that sound when the time hits their respective configured time, display back-light for improved illumination, alarm test functionality and low battery warning.

The wristwatch is operated by four buttons A, B, C and D. Button A switches between the different modes where time and date can be set, alarms and chime can be set and turned on/off and the stopwatch can be run, paused and reset. When in a time-, date-, alarm- or chime-setting mode, button C can be used to flip through between different quantities that the current time/alarm/chime-setting can be incremented with the currently chosen quantity using button D.

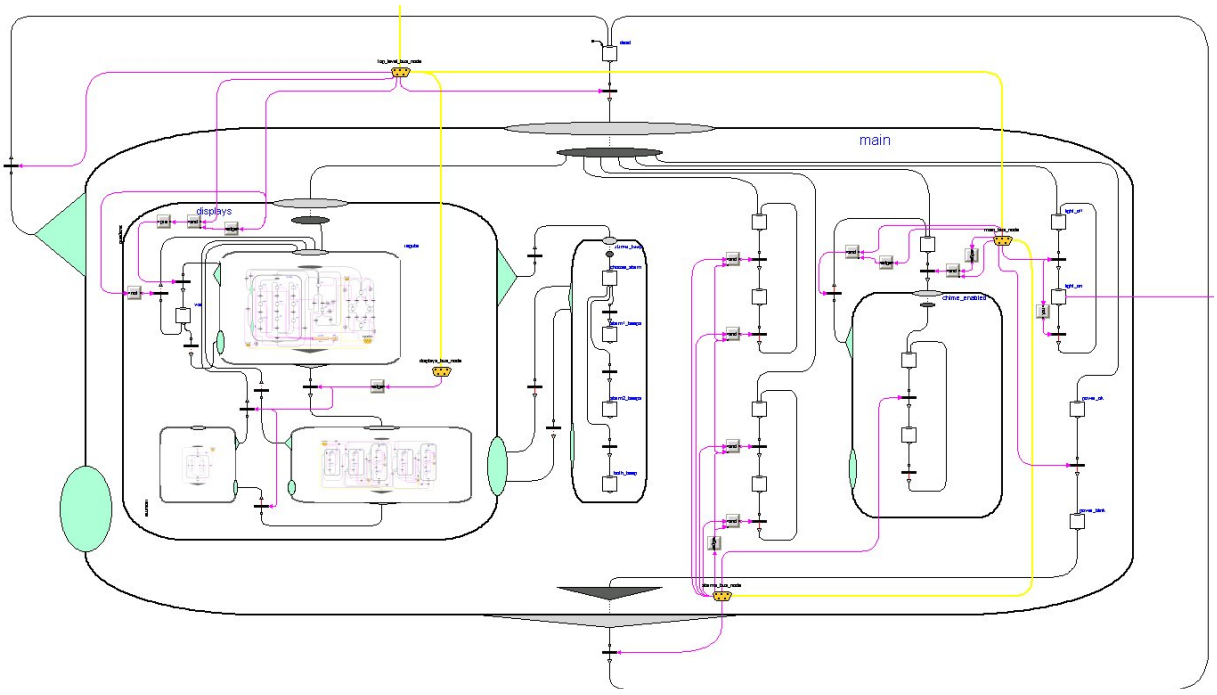



**Figure 11:** Top level of the StateGraph that defines Harels wristwatch.

When updating the time or an alarm time, button B can be used to immediately return to displaying either time or the current setting of the current alarm.

There are in total six concurrently running subsystems (Main – containing all the display and setting behavior, Alarm 1 Status, Alarm 2 Status, Chime Status, Back-Light and Power Status) that independently of each other react to the user input and the current time. There is also interaction between the Main subsystem and the Alarms/Chime Status to make it possible to concurrently guard the status of each functionality depending on the current time but also provide means to update their setting using only the given four buttons. The Main view of the StateGraph implementation of the wristwatch can be seen in **Figure 11**.

### 3.2 Controlled tank system

As another application example, the control of a tank system is present in the StateGraph library. This example is based on a similar system from *(Dressler 2004)*, which in turn is based on an example model of Karl Erik Årzén from the JGraphCharts manual. The top level view is shown in **Figure 12**: On the right side a two-tank system is present which is modeled with the Modelica.Fluid library *(Franke et. al. 2009)*: It consists of an infinite reservoir of water, "reservoir", that flows via two tanks, "tank1,

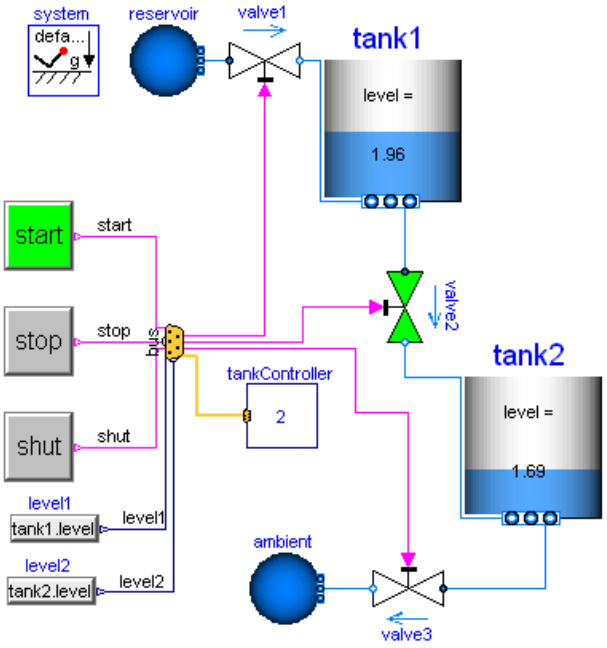



**Figure 12:** Two tank system controlled by 3 buttons.

tank2", to the environment, "ambient". The flow can be controlled by three valves, "valve1, valve2, valve3". There are three buttons, "start", "stop", "shut", to control the operation. The actual level of a tank is measured in an ideal way by accessing variables tank1.level and tank2.level. All variables are communicated via an ideal bus "bus" to the tank controller. The basic operation is to fill and empty the two tanks:

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" operation can be influenced by three 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 immediately. Then, the controller waits for further input (either "start" or "shut").
- Button "shut" is used to shutdown the process, by emptying both tanks at once. When this is achieved, the process goes back to its start configuration. Clicking on "start", restarts the process.

The tank controller is hierarchically modeled with two Parallel components and some logical blocks:

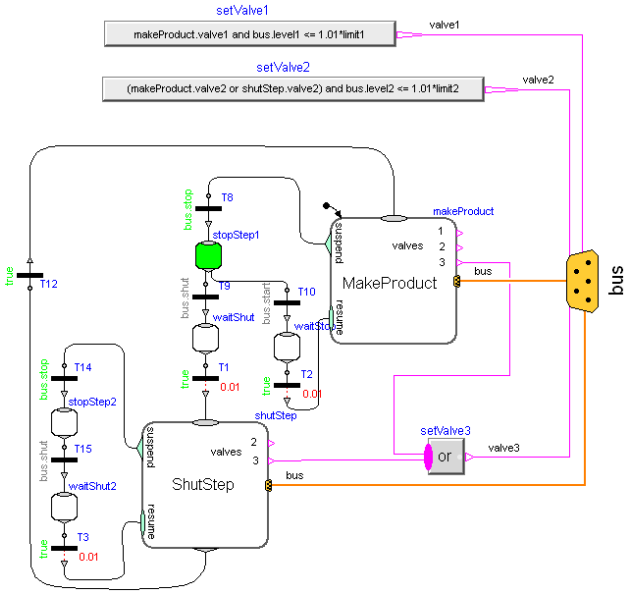



**Figure 13:** Top level view of tank controller logic.

The "MakeProduct" Parallel component is the initial step and performs the "normal" operation. When the "stop" button is pressed, the suspend transition T8 fires, the "MakeProduct" step is suspended and the graph goes in to step "stopStep1". Note, the transition condition of T8 is "bus.stop", i.e., this transition

fires when variable stop from the bus is true. When "start" is pressed again, the "MakeProduct" step is resumed at the place where it was suspended. When "shut" is pressed, the Parallel component "ShutStep" is entered to shut down the tank system. Here it is still possible to press the "stop" button and then again continue with "shut".

# 4 Formal definition of StateGraph

In section 2.1 an informal introduction to the StateGraph formalism was given. In this section, a precise mathematical description of StateGraph models will be presented. The formal definition describes the structure of a StateGraph model and its interpretation algorithm (= semantics).

## 4.1 Structure of a StateGraph model

A StateGraph model, $\Gamma$, is described by a 4-tuple:

$$\Gamma = < V_c, G, T, g_I >$$

where

- $V_c$ is a set of Boolean expressions. Boolean expressions are used as conditions of transitions. They are either external inputs or the outputs of Modelica models. A Modelica model consists of a set of differential, algebraic and discrete equations, see *(Modelica 2009, Appendix C)*.
- G is the set of Generalized Steps, $G = \{g_1, g_2, ,...\}$. A Generalized Step $g_i \in G$ can be active or not active, signaled by the Boolean Active($g_i$). A Generalized Step $g_i \in G$ is described by the 5-tuple
  $< I, R, O, S, \Gamma_s >$
  where
  I is a vector of in (entry) ports $I = [i_1,i_2,...]$,
  R is a vector of resume ports $R = [r_1,r_2,...]$,
  O is a vector of out (exit) ports $O = [o_1,o_2,...]$,
  S is a vector of suspend ports $S = [s_1,s_2,...]$,
  $\Gamma_s$ is a set of sub-graphs $\Gamma_s = \{\gamma_1,\gamma_2,...\}$
  A Generalized Step $g_i$ that has only in and out ports, $< I, O >$, is also called Step.
  A Generalized Step $g_i$ where R, S or $\Gamma_s$ is not an empty set, is also called Parallel Step.
  A sub-graph $\gamma_i \in \Gamma_s$ is described by a 5-tuple $< V_c, G, T, g_I, g_E >$ where $V_c$, G are a set of Boolean expressions and a set of Generalized steps as described above, T is the set of Transitions as described below, $g_I \in G$ is the initial generalized step that is first activated when the sub-graph $\gamma_i$ is "normally" activated and $g_E \in \{\varnothing, G\}$ is the optional exit generalized step that is the last active step, before the sub-graph $\gamma_i$ is de-activated.
- T is the set of transitions, $T = \{t_1, t_2, t_3, \ldots\}$. A transition $t_i \in T$ is defined by the 4-tuple
  $t_i = < p_{IR}(t_i), p_{OS}(t_i), Condition(t_i), Delay(t_i) >$
  where
  $p_{IR}(t_i)$ is a connected port of an in or resume vector of a succeeding generalized step $g_i \in G$.
  $p_{OS}(t_i)$ is a connected port of an out or suspend vector of a preceding generalized step $g_i \in G$.
  $Condition(t_i) \in V_c$ is the fire condition associated with $t_i$
  $Delay(t_i) \in \{\varnothing, R^+\}$ is the optional delay time associated with $t_i$. If present, the delay time is a positive real number, $Delay(t_i) > 0$.
  There is the restriction, that every "loop" must have at least one transition $t_i$ with $Delay(t_i) > 0$ in order to avoid infinite transition looping.
- $g_I$ is the initial generalized step, $g_I \in G$.

## 4.2 Interpretation Algorithm

The dynamic behavior of a StateGraph $\Gamma = < V_c, G, T, g_I >$ is given by the interpretation algorithm presented below:

(1) The initial step $g_I$ is activated. If the initial step has sub-graphs $\gamma_i \in \Gamma_s$, then all initial steps $g_I$ of these sub-graphs are activated as well. If an initial step $g_i \in G_I$ of a sub-graph has again sub-graphs, then all initial steps of these sub-graphs are recursively activated.

(2) Active($g_i$) of all Generalized Steps, including all recursive sub-graphs, is set to true, if $g_i$ is active. Otherwise it is set to false. Models are solved using Active($g_i$) as inputs.

(3) The condition expressions of all transitions in T and in all recursive sub-graphs are evaluated (either from external inputs or from outputs of models).

(4) All Transitions are determined where (a) the Transition condition is true and (b) the preceding Generalized Step is active and (c) all exit steps $g_E$ of all sub-graphs $\gamma_i \in \Gamma_s$ of the preceding Generalized Step are active, as well as of all exit steps of sub-graphs of exit steps recursively. For every such Generalized Step, at most one Transition can fire. For Transitions having the same preceding Generalized Step, the one connected to the out port or if both are connected to the same port vector, the one with the smallest vector index of the out or the suspend port respectively is marked as "fires".

(5) All Transitions that are marked as "fires" in (4) are firing, i.e., the respective preceding Genera-

lized Step of a Transition is deactivated and the succeeding Generalized Step of the Transition is activated. If a transition has a non-zero delay-time, it fires after the delay time, provided all conditions of (4) remain true during the delay time.

Deactivating a Generalized Step that has sub-graphs $\gamma_i \in \Gamma_s$ means, that all Generalized Steps in these sub-graphs and their recursive sub-graphs are deactivated as well.

Activating a Generalized Step that has sub-graphs $\gamma_i \in \Gamma_s$ means that either (a) all initial steps $g_I$ of these sub-graphs and their recursive sub-graphs are activated, or (b) the Generalized Steps are activated that have been active when this step was deactivated the last time. Case (b) is used, if the last deactivation of this step was performed via a transition of a suspend port. Otherwise case (a) is used.

Goto (2).

### 4.3 Example

The StateGraph given in **Figure 3** can be presented by the 4-tuple $\Gamma$.

$$\Gamma = < V_c, G, T, g_I >$$

where

$V_c = \{true, u\}$,
$G = \{s1, s6, p\}$
$s1 = < i[1], o[1], \varnothing, \varnothing, \varnothing >$
$s6 = < i[1], o[1], \varnothing, \varnothing, \varnothing >$
$p = < i[1], o[1], s[1], r[1], \{\gamma_1, \gamma_2\} >$
$T = \{T1, T5, T6, T7\}$
$T1 = < s1.o[1], p.i[1], true, Delay(T1)=1 >$
$T5 = < p.o[1], s1.i[1], true, Delay(T5)=1 >$
$T6 = < p.s[1], s6.i[1], u, \varnothing >$
$T7 = < s6.o[1], p.r[1], true, Delay(T7)=2 >$
$g_I = s1$

and a sub-graph can be represented by the 5-tuple $< V_c, G, T, g_I, g_E >$:

Sub-graph $\gamma_1$:
$V_c = \{\varnothing\}$,
$G = \{s2\}$
$s2 = < i[1], o[1], \varnothing, \varnothing, \varnothing >$
$T = \{\varnothing\}$
$g_I = s2$
$g_E = s2$

Sub-graph $\gamma_2$:
$V_c = \{true, time > 5\}$,
$G = \{s3, s4, s5\}$
$s3 = < i[2], o[1], \varnothing, \varnothing, \varnothing >$
$s4 = < i[1], o[2], \varnothing, \varnothing, \varnothing >$
$s5 = < i[1], o[1], \varnothing, \varnothing, \varnothing >$
$T = \{T2, T3, T4\}$
$T2 = < s3.o[1], s4.i[1], true, Delay(T2)=1 >$
$T3 = < s4.o[2], s3.i[2], true, Delay(T3)=1 >$
$T4 = < s4.o[1], s5.i[1], true, Delay(T4)=1 >$
$g_I = s3$
$g_E = s5$

## 5 Verification of StateGraph models

Even if a state machine is checked to be structurally correct, its behavior might be faulty and dangerous. A typical example is if the behavior would deadlock, i.e., that no further transitions can be performed. Such behavior is related to the action and transition logic, not only to the topology of the StateGraph itself. Dymola *(Dymola 2009)* has been experimentally extended to extract all Boolean equations in order to facilitate model checking with external tools. The language used is SMV (Symbolic Model Verification) and the tool used is NuSMV *(NuSMV 2009)*.

Consider the example in **Figure 14**. It has four independent StateGraph models, two are modeling some processes which compete on using two resources. The allocations of the resources are done in opposite order which means that there is a risk of deadlock. Detecting such potential problems is in general hard. Dymola produces code in SMV as shown below:

```
freeA_inport_fire := release1A_fire |
release2A_fire;
next(pre_freeA_newActive) :=
  freeA_inport_fire | freeA_active &  !
  freeA_outport_fire;
```

Relations are converted to unknown inputs. When-clauses are converted to if (case) according to Modelica specification. Condition for non-deadlock is expressed using temporal logic according to the Computational Tree Logic syntax, e.g.:

```
_Dymola_SMV(
"CTLSPEC AG (! pre_freeA_newActive ->
 EF pre_freeA_newActive)");
```

The String argument to the special built-in function _Dymola_SMV means "For **A**ll states such that **not** pre_freeA_newActive (resource A not free) there **E**xists eventually in the **F**uture a state when pre_freeA_newActive (resource A free)"

NuSMV uses a BDD (Binary Decision Diagram) algorithm to verify the specification (NuSMV command check_ctlspec). If the specification is not always true, NuSMV presents a sequence of input events that will show the failure, i.e., in this case deadlock. Such a deadlocked situation is shown in Figure 14 with active Steps marked green.

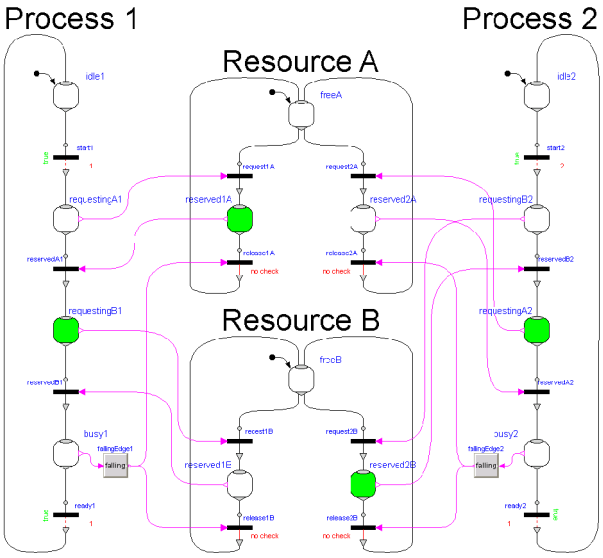



**Figure 14:** Two processes trying to acquire two resources ending up in a deadlock.

## 6 Conclusion

A new library Modelica_StateGraph2 was presented to model safe hierarchical state machines in combination with any Modelica model, e.g., controllers, logical blocks, functions and physical systems described by differential-algebraic equations. The library is designed to model the logic of reactive systems and to describe hybrid systems. The library is freely available from www.modelica.org/libraries, it is distributed in Dymola 7.3, and it is planned to include it in one of the next versions of the Modelica Standard Library. The work on the library will continue especially to take advantage of the features of the Modelica_EmbeddedSystems library *(Elmqvist et. al. 2009)*:

## 7 Acknowledgements

Partial financial support of DLR by BMBF (BMBF Förderkennzeichen: 01IS07022F) for this work within the ITEA project EUROSYSLIB (www.itea2.org/public/project_leaflets/EUROSYSLIB_profile_oct-07.pdf) is highly appreciated.The authors also would like to thank Daniel Weil from Dassault Systèmes for fruitful discussions.

## References

André, C. (2003): **Semantics of S.S.M (Safe State Machine)**. I3S Laboratory – UMR 6070 University of Nice-Sophia Antipolis / CNRS. www.i3s.unice.fr/~map/WEBSPORTS/Documents/2003a2005/SSMsemantics.pdf

Bauschat, M., Mönnich, W., Willemsen, D., and Looye, G. (2001): **Flight testing Robust Autoland Control Laws**. In Proceedings of the AIAA Guidance, Navigation and Control Conference, Montreal CA.

Benveniste A., Caspi P., Edwards S.A., Halbwachs N., Le Guernic P., and Simone R. (2003): **The Synchronous Languages Twelve Years Later**. Proc. of the IEEE, Vol., 91, No. 1. Download: www.irisa.fr/distribcom/benveniste/pub/synch_ProcIEEE_2002.pdf

Dressler I. (2004): **Code Generation From JGrafchart to Modelica**. Master thesis. Supervisor: Karl-Erik Arzen, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden. www.control.lth.se/documents/2004/5726.pdf

Dymola (2009). **Dymola Version 7.3**. Dassault Systèmes, Lund, Sweden (Dynasim). www.dymola.com/.

Elmqvist H., Otter. M., Henriksson D., Thiele B., Mattssson, S.E. (2009): **Modelica for Embedded Systems**. In Proc. of Modelica'2009 Conference, Como, Italy. www.modelica.org/events/modelica2009

Franke R., Casella F., Otter M., Proelss K., Sieleman M., Wetter M. (2009): Standardization of thermo-fluid modeling in Modelica.Fluid 1.0. In Proc. of Modelica'2009 Conference, Como, Italy. www.modelica.org/events/modelica2009

Harel, D. (1987): **Statecharts: A Visual Formalism for Complex Systems**. Science of Computer Programming 8, 231-274. Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel. www.inf.ed.ac.uk/teaching/courses/seoc1/-2005_2006/resources/statecharts.pdf

Lynch N., Segala R., and Vaandrager F. (2002): **Hybrid I/O Automata**. MIT Laboratory for Computer Science, techreport, MIT-LCS-TR-827b. Download: theory.lcs.mit.edu/tds/papers/Lynch/HIOA-final.ps

Malmheden M., Elmqvist H., Mattsson S.E., Henriksson D., and Otter M. (2008): **ModeGraph - A Modelica Library for Embedded Control Based on Mode-Automata**. B. Bachmann (editor), in Proc. of Modelica'2008 conference, Bielefeld, Germany. www.modelica.org/events/modelica2008/Proceedings/sessions/session3a3.pdf

Maraninchi, F. and Rémond, Y. (2002): **Mode-Automata: a New Domain-Specific Construct for the Development of Safe Critical Systems.** www-

verimag.imag.fr/~maraninx/SCP2002.html

Modelica (2009). **Modelica Language Specification 3.1**. www.modelica.org/documents/ModelicaSpec31.pdf

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. www.modelica.org/publications/papers/scsc98fp.pdf

NuSMV (2009): A symbolic model checker. nusmv.irst.itc.it.

Otter, M., Årzén, K.-E., Dressler, I. (2005): **StateGraph - A Modelica Library for Hierarchical State Machines**. Proceedings of the 4th International Modelica Conference. TU-Hamburg-Harburg, Germany. www.modelica.org/events/Conference2005/online_proceedings/Session7/Session7b2.pdf

Stateflow (2009): www.mathworks.com/products/stateflow

# Appendix

## A1 Mapping StateGraph to Modelica

In this section it is sketched how a StateGraph model is mapped to Modelica. This section is based on the implementation technique used in (*Mosterman et. al. 1998*, *Malmheden et. al. 2008*, *Otter et. al. 2005*): Steps, Transitions, and Parallel components are mapped to Boolean equations. These equations are handled as any other Modelica equations, e.g., for the code generation the equations are sorted and therefore the evaluation sequence of a StateGraph model and/or of a hybrid system is automatically determined. Therefore, defining how the StateGraph elements are mapped to Boolean equations defines automatically also the semantics of hybrid systems built by StateGraph and other Modelica models. The mapping algorithm starts with a sketch of the used interfaces between the elements:

A Step component has a vector of connectors called "Step_in" in order to connect from transitions to a step, and a vector of connectors called "Step_out to connect from a Step to Transitions.

A Transition component has a (scalar) connector called "Trans_in" to connect from a Step to a Transition and a (scalar) connector called "Trans_out" to connect from a Transition to a Step.

Only unary connections are allowed, i.e., exactly one connection must be made between one element of a vector of connectors and a scalar connector. The connector classes use pair-wise the same variables, but with different causalities (with exception of "node"), as shown in the next table:

| **connector** Step_out | **connector** Trans_in | |
|---|---|---|
| **output** | **input** | Boolean available |
| **input** | **output** | Boolean fire |
| **output** | **input** | Boolean checkLoop |
| | | Node node |
| **connector** Trans_out | **connector** Step_in | |
| **output** | **input** | Boolean fire |
| **output** | **input** | Boolean checkLoop |
| **input** | **output** | Boolean checkUnary |
| | | Node node |

```
record Node
   Boolean suspend;
   Boolean resume;
   function equalityConstraint
      input  Node node1;
      input  Node node2;
      output Real residue[0];
   algorithm
   end equalityConstraint;
end Node;
```

The meaning is the following: When an element of the "Step_out" vector at a Step is connected to the "Trans_in" connector of a Transition, then the signals "available, checkLoop" are computed in the Step and are communicated to the Transition. On the other hand, the signal "fire" is computed in the Transition and communicated to the Step. The meaning of "node" is explained in section A2.

When input/output prefixes are used in a Modelica connector, then block diagram semantics applies for a connector (e.g., only one signal can be connected to an input). Since connectors "Step_out" and "Trans_in" have both input and output variables, only unary connections are possible, as desired. The basic form of "Trans_out" and "Step_in" has either only "output" or "input" variables and therefore unary connections are not guaranteed. For this reason, the dummy variable "checkUnary" is introduced with opposite input/output prefixes. Now, only unary connections are here possible too[2].

A Transition is basically defined by the following equations, depending on the options that have been selected in the parameter menu:

| **Equations of a Transition component** |
|---|
| Immediate transition:<br>`fire = condition and trans_in.available;` |
| Delayed transition:<br>`enableFire = condition and` |

[2] The alternative to use an assert with cardinality is not possible, because the resume connector is conditional and then it cannot be referenced in an assert.

```
              trans_in.available;
when enableFire then
   t_next = time + waitTime;
end when;
fire = enableFire and time >= t_next;
```

Propagation of signals (in both cases):

```
trans_in.fire  = fire;
trans_out.fire = fire;
```

Basically, the equations state that variable fire = true, if (1) the fire condition "condition" is true and (2) if the preceding step is active (trans_in.available =true). For a delayed transition, additionally a time delay is introduced. The "fire" variable is then reported to the preceding and the succeeding steps.

A Step is basically defined by the following equations:

**Equations of a Step component**

Set active flag:

```
newActive =
  if node.resume then oldActive
  else anyTrue(step_in.fire) or (active
       and not anyTrue(step_out.fire))
       and not node.suspend;
active = pre(newActive);
when node.suspend then
  oldActive = active;
end when;
```

Set available flag:

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

The function **anyTrue**(..) returns true, if at least one element of the input vector is true. In a Step, the next value of "active" is computed (called: "newActive"). It is assigned in the next event iteration to the actual value, "active", via "active=**pre**(newActive)". The equations state, that the Step becomes active in the next iteration when one of the transitions connected to the step_in connectors fire. The Step remains active if it was active and no transition connected to one of the step_out connectors fire.

If the Step is used inside one or more Parallel components, the state of the nearest enclosing Parallel component is propagated via the record "node". Details are given in section A2. At this stage it is sufficient to know that if node.suspend = true, then an enclosing Parallel component was suspended and if node.resume = true, then an enclosing Parallel component was entered via the resume port. If a Parallel component is suspended, the current value of "active" is saved in "oldActive", and "newActive" is set to false. If a Parallel component is resumed, "newActive" is set to the saved value of "oldActive".

The "active" flag of a Step is reported to the transitions connected to this Step in the following way: If a step has only one outgoing transition:

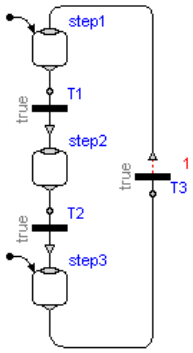
```
step_out[1].available =
    active and not node.suspend
```

Therefore, the "available" flag propagated to the Transition is set to true, if the step is active and if an enclosing Parallel component is not suspended.

If a Step has several outgoing transitions, two or more might fire at the same time instant. The transition that is connected to the lowest index of the step_out connector vector is defined to have highest priority. For example, if a Step has two outgoing transitions, then the "available" flag of step_out[1] is set as previously. The "available" flag of step_out[2] is only set to true, if the transition that is connected to step_out[1] does not fire and no enclosing Parallel component is suspended.

The equations for a Parallel component are handled similarly to a Step. For space reasons, they are not listed here.

## A2 Guaranteeing graph properties and propagation of suspend/resume flags




In the previous section A1 it is sketched how the basic elements are defined by Boolean equations and how only 1:1 connections can be made. Still some properties of a StateGraph are not yet guaranteed. For example, two initial steps might be defined in a simple StateGraph model (see Figure to the right). This gives perfectly legal Modelica code, but the simulation would be wrong. We will now discuss how the basic graph properties are guaranteed and how the suspend/resume information of Parallel components is propagated:

Record "node" in the connectors, see definition in section A1, is an "overdetermined record" due to function "equalityConstraint()", see *(Modelica 2009, section 9.4)*. The idea is the following: The overdetermined record R in a connector has more variables than permitted by a "balanced model". When two connectors c1 and c2 are connected, then the desired connection equations are c1.R = c2.R. If a loop of connected components is present, this might give too many equations (= more equations as unknowns). If this is the case, exactly for one connection set in a loop the equations "0 = R.equalityConstraints(c1.R,

c2.R)" have to be used instead of the desired equations "c1.R = c2.R". For example, a transformation matrix has 9 redundant elements describing 3 independent variables. In this case, the equalityConstraint(...) function has to return the 3 constraint equations between the 9 redundant variables.

In order that a translator can select which connection equations to use, built-in operators are provided to construct an undirected dependency graph of the connectors. For example, if a component has two connectors ca and cb, a definition of the form:

```
Connections.branch(ca.R, cb.R);
```

must be present in the component. This definition states that cb.R is equal to ca.R in this component. One connector must be defined as root of the graph. As a result, a set of undirected graphs is constructed. The translator has to arbitrarily cut a graph at connection sets, so that a spanning tree is constructed. In the "tree", connection equations of the form c1.R = c2.R are used. For all connectors that have been removed to arrive at a "tree", the connection equations 0=R.equalityConstraint (c1.R, c2.R) are used.

In the StateGraph library, suspend and resume flags are stored in an overdetermined record "Node". The Node.equalityConstraints(..) function returns a vector with size zero. Therefore, no equations are generated for connections that have been removed to arrive at a "tree". When the root of a graph is appropriately selected, then the suspend/resume flags are just propagated to all components in this graph, even if loops are present (since the loops are cut, and no connection equations for node variables are introduced at these cuts).

The operators available in Modelica 3.1 are not sufficient and two additional ones had to be introduced: "**Connections.uniqueRoot**(R, message)" states that "R" is a unique root of the graph. If this operator is used, the corresponding graph must have exactly one such definition. The second argument "message" shall be reported in the error message, if more than one root is defined.

The usage of "uniqueRoot(..)" and of "branch(..)" are sketched in **Figure 15**: Roots are defined at the initial step (root1) and at the entry port of every branch of a Parallel component (root2, root3). Then "branches" are defined along the corresponding state machine structure. If any such connection graph has more then one root, the StateGraph graph is wrong. E.g., if two initial steps would be defined, or if a branch of a Parallel component would branch out into the "outer" loop, the connection graph would have two roots which would trigger an error.

With the new built-in operator "I = **Connections.uniqueRootIndices**(Ra, Rb, message)", information about the connection structure of a Parallel component can be obtained: Ra is a vector of roots and Rb is a vector of other overdetermined records. The function returns an Integer vector "I". I[i], i=1:size(Rb,1), defines that there is a path from root I[i] to record Rb[i]: Ra[I[i]] → Rb[i]. It is an error if such a path does not exist. The remaining elements of vector I are the indices of Ra that do not have a path to an element of Rb. Due to the construction, the function returns an error, if there are no paths to all exit ports. So, every branch that ends at an exit port, must start at an entry port of the same Parallel component.

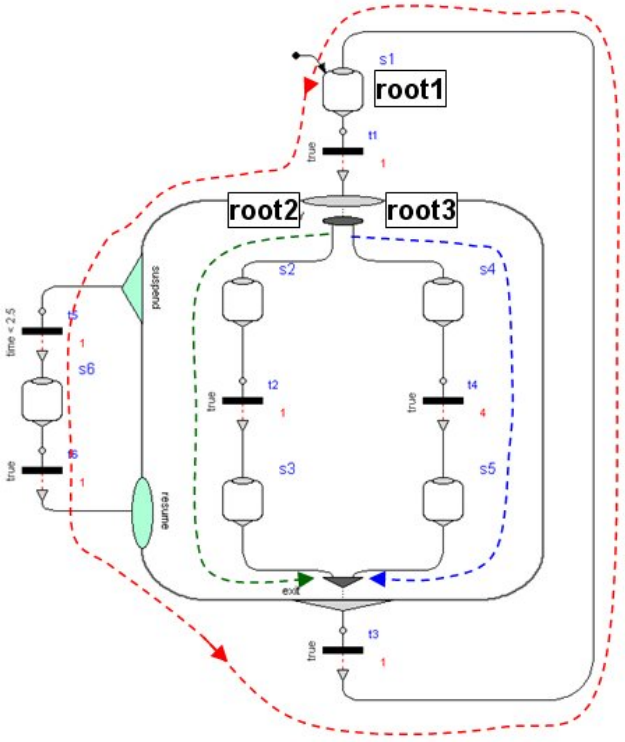



**Figure 15:** 3 virtual connection graphs to verify StateGraph properties and to propagate resume/suspend flags.

Typical usage of this function:

```
  EntryPort entry  [nEntry];
  ExitPort  exit   [nExit];
  Integer   indices[nEntry];
equation
  Connections.uniqueRoot(entry, "...");
  indices = Connections.uniqueRootIndices
                      (entry, exit, "...");
```

Example: The function returns the following values for the graph in Figure 15:

```
nEntry=2, nExit=2,
indices[1] = 1, indices[2] = 2
```

The meaning is that there is a path from entry[indices[1]] (connected to Step s2) to exit[1] (connected to Step s3), and a path from entry[indices[2]] (connected to Step s4) to exit[2] (connected to Step s5).

### A3 Avoiding infinite transition loops

The basic semantics of a StateGraph graph is that at one time instant, during event iteration, all transitions fire, until none of the transitions can fire anymore. In order that no infinite looping can occur, there must be at least one delayed transition in "every loop", since at a delayed transition the looping stops at the current time instant.

In order to verify this property, the Boolean flag "checkLoop" is propagated through the connection structure, see connectors in section A1. At delayed transitions and at steps that do not have an input transition, this flag is initialized. If there is no delayed transition in a loop, an algebraic system of Boolean unknowns occurs. Since this system of equations cannot be solved, an error is triggered. In the connectors, "checkLoop is defined with the new annotation "BooleanLoopMessage = string". If the corresponding variable appears in an algebraic loop with Boolean unknowns, the BooleanLoopMessage is included in the error message, in order to get meaningful error reporting.

### A4 Automatic connection to next free index

When connecting a Step with a transition, the dimension of the vector of connectors Step.outPort has to be increased by one, say to dimension N, and then the connection has to be performed from Step.outPort[N] to the scalar transition input port. Performing this manually is very inconvenient and error prone. For this reason, in Modelica 3.1 *(Modelica 2009, section 17.6)* the new annotation "connectorSizing" was introduced, that is used for all vector connections in the StateGraph library.

Example:

```
model Step
   parameter Integer nIn=0 annotation(
             Dialog(ConnectorSizing=true));
   StepIn inPort[nIn];
   ...
end Step;
```

When this model is used and a connection is made to vector "inPort", then the tool increments the dimension nIn by one and performs the connection to this new index. Therefore, performing connections between Steps and Transitions is convenient for a user and only requires dragging a line between the corresponding connectors.