ModeGraph -
A Modelica Library for Embedded Control Based on Mode-Automata

Martin Malmheden[1], Hilding Elmqvist[1], Sven Erik Mattsson[1], Dan Henriksson[1], and Martin Otter[2]
[1]Dynasim AB (A Dassault Systèmes Company), Ideon Science Park, SE-223 70 Lund, Sweden
[2]German Aerospace Center (DLR), Institute of Robotics and Mechatronics, Oberpfaffenhofen, 82234 Weßling, Germany
{Martin.Malmheden, Hilding.Elmqvist, SvenErik.Mattsson, Dan.Henriksson}@3ds.com , Martin.Otter@dlr.de

Abstract

The ModeGraph library is a new Modelica library for modeling of hybrid and embedded control systems based on Mode-Automata semantics. Actions can be associated with discrete states in a way that makes sure that the single-assignment rule is fulfilled. Consequently, non-deterministic variable assignment is impossible, which is usual in nearly all other state machine formalisms. Besides Mode-Automata, concepts from Sequential Function Charts (SFC)/Grafcet, Statecharts, and Safe State Machines (SSM) are utilized to provide a flexible modeling environment for safe, hierarchical state machines where Modelica is used as action language. ModeGraph shall replace the existing Modelica.StateGraph library. The implementation of ModeGraph requires extensions to the Modelica language, in order to support the Mode-Automata semantics and to drastically reduce code overhead and improve performance of modeled graphs.

Keywords: Statechart, Mode-Automata, Finite State Machines, Hybrid Control, StateGraph, Modelica

1 Introduction

The StateGraph library [5] is a sublibrary in the Modelica Standard Library 2.1 (from 2004) and later versions, providing components to model hierarchical state machines using Modelica as an action language. The StateGraph library has several significant drawbacks that are mainly due to the underlying implementation language Modelica 2, where some special features needed for hierarchical state machine modeling and for Mode-Automata are missing.

A new Modelica library for modeling hierarchical state machines is proposed in this paper. It is a more Statechart [2] oriented approach compared to StateGraph, but avoids several deficiencies of the Statechart formalism in order to arrive at safe state machines. The library is capable of handling extended state machine properties, such as hierarchy (meta states), orthogonality (parallel substates), synchronization, and preemption. All StateGraph functionality is available, but with a new simplified implementation. The ModeGraph library ensures safe state machines, especially with respect to

1. upper limit on execution time of one cycle,
2. guaranteed deterministic variable assignment.

The library is based on extensions to the Modelica language, e.g., ensuring mutual exclusivity between states. Usage of the new Modelica 3.0 graphical annotations provides a more modern look and feel.

In the following sections the ModeGraph library will be explained and excerpts of the implementation will be presented. A ModeGraph is defined in Modelica using Boolean equations. As a result, the exact semantics of ModeGraph is formally defined with the Modelica semantics (equations are sorted and iteration takes place, if pre(x) ≠ x). General concepts taken from Finite State Machines (FSM), Statecharts [2], Sequential Function Charts (SFC) [7], and Safe State Machines (SSM) [1] will be used as references and benchmarks to demonstrate the feasibility and applicability of ModeGraph.

2 Steps and Transitions

An FSM describes a behavior by decomposing it into a distinct finite set of states visualized by state-transition diagrams. States are usually illustrated by rectangles with rounded corners. An FSM is often used to model reactive systems, which means it reacts to certain stimuli, usually called inputs. A transition is depicted with an arrow between two states and a transition condition written next to the arrow. When the condition evaluates to true, the transition is taken, and a change of state is performed. As an ex-

ample, see Figure 1, where the system initially is in state `A`. When input α occurs, the state will change from `A` to `B`. The arrow originating in a small black dot is used to mark the initial state of the system.

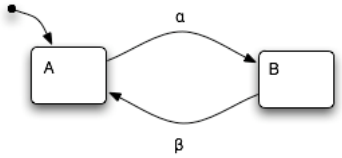

Figure 1: Simple state machine with two states and two transitions.

Inheriting much of the semantics from StateGraph, the basic components of ModeGraph are Steps and Transitions that are both similar to the corresponding StateGraph objects. Figure 2 shows the ModeGraph equivalent of Figure 1. We will proceed to describe the Steps and Transitions in more detail.

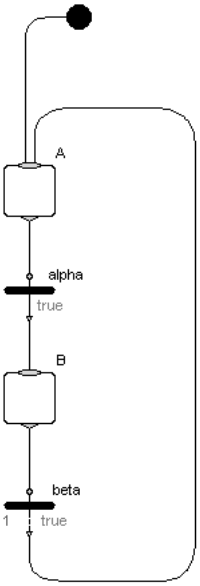

Figure 2: A ModeGraph comprised of two Steps and two Transitions.

2.1 Steps

There are two types of Steps: a regular Step and a StepWithSignal. The state of a regular Step is represented by a Boolean, `active`. In the case of the StepWithSignal, `active` is instead a BooleanOutput that can be graphically connected to other components, typically to logical blocks:

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

For a Step with one inport and one outport `available` is defined as:

```
available = active;
```

The function `anyTrue` iterates through its argument array of connectors and returns true if any of them is true. The state of the Step in the next iteration is called `newActive`, hence `active` is set to `pre(newActive)`. A Step is said to be available to the successor Transition when active is true.

Several transitions can lead to and from a Step, respectively. This is implemented with two vectors of connectors, called `inPort` and `outPort`. The Step component is said to be a mode, hence only one Step at each hierarchical level is allowed to be active at a given time instant. This requires restrictions on the outPort fire mechanisms, which will be explained in detail below.

2.2 Transitions

Transitions are used to decide when a change of state should be performed. A basic Transition will check if its predecessor Step is available and evaluate if its transition condition is true (visualised by the condition being colored green). If this is the case, it will send a signal, `fire`, to its surrounding Steps. Hence, the previous Step will turn inactive and the following will turn active.

```
inPort.fire  = condition and
                      inPort.available;
outPort.fire = inPort.fire;
```

The signal flow between Steps and Transitions is viewed in Figure 3.

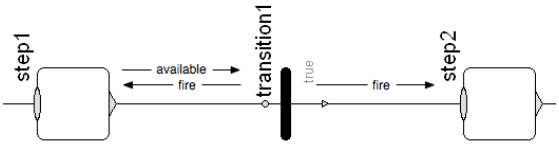

Figure 3: Signal flow between Steps and Transitions.

2.3 Delayed Transitions to Break Loops

Consider the sequence of Steps and Transitions with true conditions in Figure 4. A graph like this is said to be unstable. At a given time instant, the active Step is undefined, because all Transitions will evaluate to true at all times. The code below represents the evaluation of the chain in Figure 4.

```
s1.newActive = (pre(s1.newActive)
                      and not t1.fire)
             or t2.fire or entry.fire;
t2.fire = condition and
                    pre(s2.newActive);
s2.newActive = (pre(s2.newActive)
                      and not t2.fire)
                          or t1.fire;
t1.fire = condition and
                    pre(s1.newActive);
```

Examining this code, it is clear that there is no defined active Step at a given time instant, since it would immediately fire and activate the next Step. Loops like this illustrate the need for a Transition that requires the preceding Step to be available and its condition to be true for a certain period of time before it fires. This is shown by `t2` in Figure 5. This type of Transition is called delayed Transition and requires additional equations to decide how long a transition is delayed until it can fire.

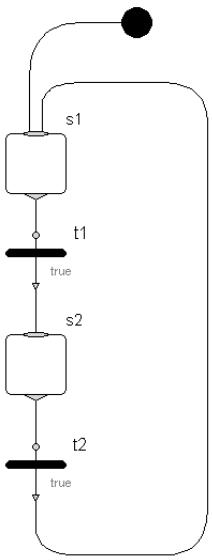

Figure 4: An infinite loop of true transitions.

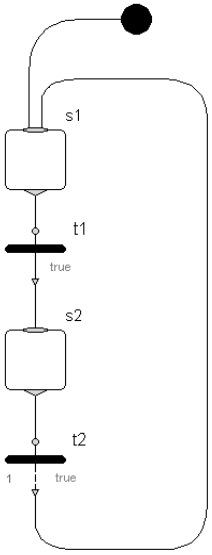

Figure 5: A loop broken by a delayed Transition `t2`.

In the present ModeGraph prototype, a parameter `waitTime > 0` defines the duration for which the fire conditions need to be true before the transition can fire. The release version will alternatively allow definition of the delay by the number of sample periods (with a default of one period), if the ModeGraph is used in a sampled data system. A delayed Transition is currently defined as:

```
enableFire = condition and
                    inPort.available;
when enableFire then
  t_start = time;
end when;
fire = enableFire and
          time >= t_start + waitTime;
inPort.fire = fire;
outPort.fire = fire;
```

The concept of delayed transitions is a generalization of the SFC semantics, where every transition from "bottom" to "top" is delayed by one cycle. Introducing delayed transitions explicitly allows drawing state machines arbitrarily without the restriction to always draw it from "top" to "bottom" which is not practical for Statechart-type state machines. Delayed transitions are, e.g., also present in SSM [1], where transitions are by default delayed by one cycle. In SSM "immediate transitions" (denoted with the "#" symbol) are "immediate" and equivalent to the normal Transitions in ModeGraph.

ModeGraph has the essential requirement, that every loop must have at least one delayed transition. In the next section it is described how a violation is detected during translation. This gives both a guarantee that infinite looping is not possible, and it gives an upper limit on the evaluation time of a ModeGraph at any time instant. Both properties are important for safe embedded control systems.

As mentioned above, Steps can have multiple input and output transitions, and only one Step is allowed to be simultaneously active at every level. This requires priorities among the output transitions. The most intuitive way is to use the index of the port array as priority. A lower index represents higher priority.

The available flag needs to take priority into account and a port is available if the Step is active and if no port with higher priority fires:

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

2.4 Graphs with Infinite Loops

Assume that a user creates a graph containing a loop where the conditions of all Transitions are true, as in Figure 4. With the current Step and Transition definitions, the graph will translate, but the solver will not be able to converge towards a single active Step. This kind of undefined behavior is obviously dangerous and is not allowed. To identify cases like this during translation, the signal flow can be slightly changed by introducing a Boolean, `loopTest`. The new signal flow between Steps and Transitions is depicted in Figure 6.

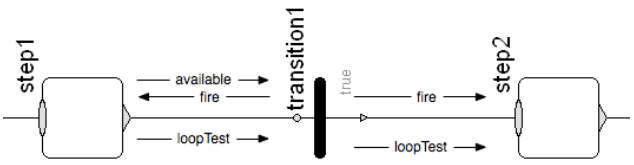

Figure 6: New signal flow with added loop checking.

The idea is to let Steps and undelayed Transitions just pass the signal on, while a delayed Transition and all entry points will set `loopTest` to true. If only Steps and undelayed Transitions are present in a loop, the translator will recognize an algebraic loop of Boolean equations, and will print an error message because Boolean algebraic loops cannot be solved. If a delayed Transition is included, the algebraic loop will be broken, and the graph will safely translate. The code for the loop testing is simple:

In a Step:

```
for i in 1:size(outPort,1) loop
  outPort[i].loopTest =
            anyTrue(inPort.loopTest);
end for;
```

In a Transition:

```
outPort.loopTest = inPort.loopTest;
```

In a delayed Transition

```
outPort.loopTest = true;
```

This "brute force" method has the slight drawback that no better loop breaking check can be provided. In principal, it might be possible to have only undelayed transitions and if the transition conditions are restricted, it might be possible to prove that infinite looping is not possible.

3 Encapsulation and Aggregation

The FSM formalism is adequate as long as the modeled behavior remains reasonably simple. When the number of states and transitions increases, the complexity of the FSM grows exponentially. This is fatal to readability and strongly confines the viability of the graph. Thus, when a state machine grows in complexity, a strong formalism should support object-orientation and proper encapsulation of isolated parts of the behavior to ensure well-defined interfaces.

Some remedies for the mentioned problems were introduced by David Harel in Statecharts [2], where several new properties were presented to extend FSM. Being able to cluster states into a superstate makes it possible to identify similarities between a number of states and draw advantages from common properties among them. Clustering of states enables reuse of larger parts of a behavior than just a single state. The superstate has a default entry point, which is connected to the initial state with the same notation as the initial state arrow. In Figure 7, `B` and `C` share the common property of transition β leading to state `A`.

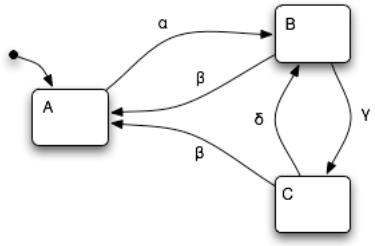

Figure 7: Three states of which two share common properties.

Thus, `B` and `C` can be clustered together into state `D` in Figure 8. Note the improved visual appearance in Figure 8 compared to Figure 7, despite the exact same behavior of states `A`, `B`, and `C`.

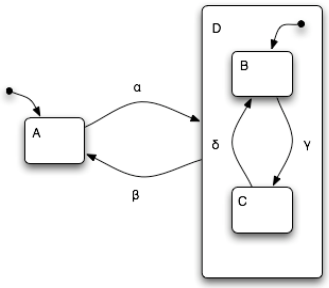

Figure 8: Two states clustered together in a superstate.

Refinement of a state involves identification of a number of child states with unique properties within a particular state. In Figure 8, states `B` and `C` can be said to be a refinement of state `D`. Hence, state `D` is said to be the superstate of state `B` and `C`. Being in one of the substates implicitly means also being in the superstate. The superstate `D` in Figure 8 is said to be the XOR-decomposition of its substates.

3.1 ModeGraph Composite

ModeGraph allows aggregation of states into superstates. A Composite component inherits from ModeGraph.Composite and has inPort and outPort connectors defined, like a regular Step, but also suspend ports and resume ports - like in StateGraph. Figure 9 shows a ModeGraph corresponding to the chart in Figure 8.

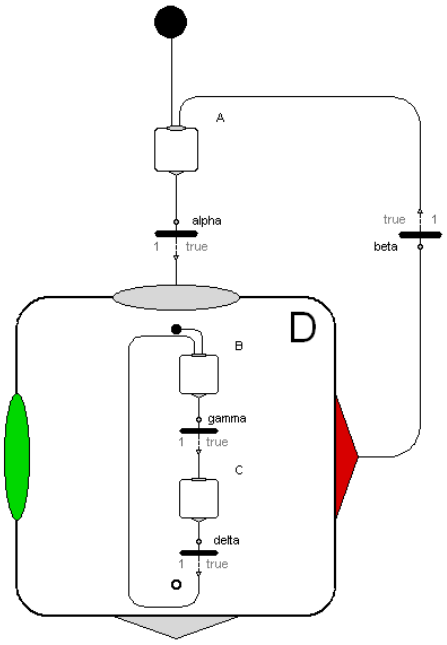

Figure 9: ModeGraph containing two Steps clustered inside a Composite. State `D` is a Modelica mode block where the diagram layer is visible in the icon. Compare with Figure 8.

The initial Step, B, of the Composite is connected to the entry port, depicted with a black dot. Similarly, there is an optional exit port, illustrated with two circles at the bottom of the Composite. This notation is inspired by the semantics of SSM, but is slightly modified to provide a more consistent look. In SSM, a specific 'final step' indicates when the superstate may be exited through the outPort, and is depicted with two circles. To prevent misuse, there is an exit port in the Composite and Parallel ModeGraph components that the 'final step' should be connected to. When this step is active, the outPort of the Composite becomes available.

The difference between entry/exit and the existing StateGraph approach extends beyond the mere graphical deviation. The entry model contains a state connected to the black connector dot that is initially true. Having an entry state, no specific InitialStep component is required. This prevents the user from making mistakes by, for example, placing two InitialStep components in a graph. The code below defining the entry point ensures that the state remains true for one iteration, when the Composite turns active, and then switches to false.

```
    Entry entry(fire(start = false,
                     fixed = true));
  protected
    Boolean active(start = true);
                   fixed = true));
  equation
    active = pre(active) and not
                     pre(entry.fire);
    entry.fire = pre(active);
```

When the Step connected to the exit port is active, the Transition connected to the outPort of the Composite may fire (if its condition is fulfilled). This calls for a definition of how the state of a Composite is evaluated:

```
  available = exit.exit.available and
      allSubBlocksFinished and active;
  newActive = (active and not
          anyTrue(outPort.fire) and not
          anyTrue(suspend.fire)) or
          anyTrue(inPort.fire) or
          anyTrue(resume.fire);
  active = pre(newActive);
```

In the code above, the state of the Composite, `active` is set to `pre(newActive)` to avoid an algebraic loop involving mode conditions that will be introduced later in this paper.

An important feature of ModeGraph is conditional execution. This applies for the Composite component, whose associated code is only executed when the composite is active. This will be further explained in Section 6.

4 Preemption and Exception

Aggregation of states introduces new possibilities. Being an own entity, it is possible to have a transition drawn directly from the superstate. This will result in a preemption, and the superstate is left regardless of which of the substates is active, see, e.g., transition β in Figure 8. Of course, normal exit is possible by having a transition originating in an inner state and targeting an outer. Notice how state D in Figure 10 is only left through transition β if state C is active.

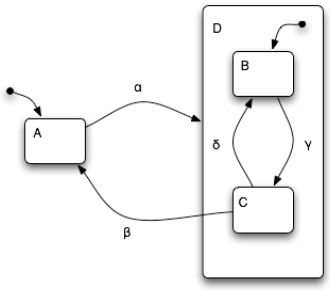

Figure 10: Superstate D can only be left when in substate C.

4.1 ModeGraph Exit and Preemption

To exit a Composite, the final step is connected to the mentioned exit port. When the final step is active, `exit.exit.available = true`, and a transition connected to the Composite outport becomes enabled. The ModeGraph realization of Figure 10 is shown in Figure 11.

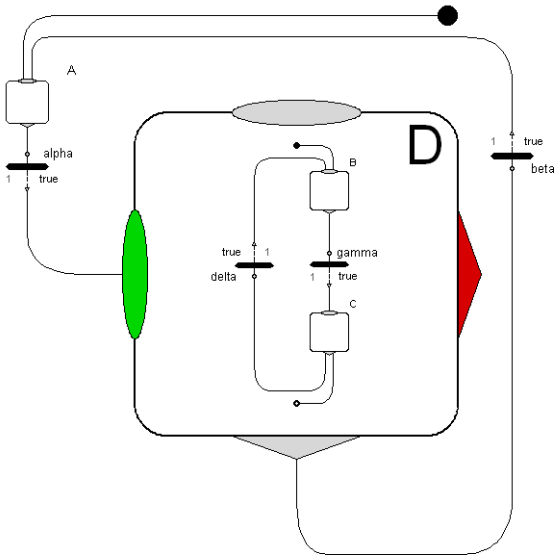

Figure 11: Composite D can only be left if Step C is active, compare with Figure 10.

A ModeGraph Composite has an array of suspend connectors. Recalling the active condition of the Composite, it is clear that after a suspend port fires,

the Composite is no longer active. This behavior is used to preempt a Composite without necessarily having reached the final Step, i.e., the one connected to the exit port. The condition of `suspend.available` needs to equal the state of the Composite, since it should be preempted only when it is active. The same kind of prioritization as for Steps is performed here:

```
for i in 1:nSuspend loop
  suspend[i].available =
         if i == 1 then
            active
         else
           active and not
                  suspend[i-1].fire;
end for;
```

The suspend port can be compared to the Statechart equivalence of drawing a transition directly from the superstate to an outer state, compare for example transition β in Figure 8 and its equivalent in Figure 9. The deactivation of the Composite does not, explicitly, influence the internal states of the Composite. The state of the subblocks will be kept, but all internal interaction will be frozen.

4.2 History and CLH

The concept of preemption introduces an additional way of entering a superstate. Normally, entry is performed through the default entry point, as mentioned above. This behavior can be compared to a subroutine that has only one entry point. There is an obvious advantage of offering additional ways of entering an aggregation, similarly to the ways a co-routine may be entered. Hence, re-entering a superstate, it is also reasonable to be able to enter the most recently visited substate.

Memory of the internal state of a superstate is called "entry by history" in Statecharts, and depicted with an encircled H to which transitions can be connected. The H-entry will make the previously visited state before preemption at the current level active. If the superstate is entered for the first time, the default entry arrow is used. Assume for example that state `C` is active and transition β is taken in Figure 12. If subsequently transition α is taken, state `C` (and of course also state `D`) will once again be entered.

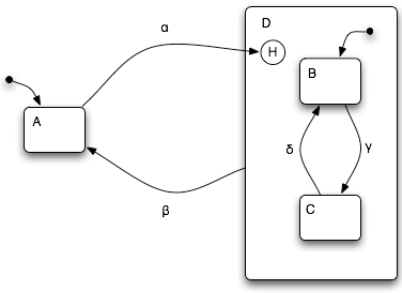

Figure 12: Superstate `D` is entered through an H-entry.

To handle history of several nested superstates, the H-entry can be extended to be applied all the way to the lowest level. This is in Statecharts called an H*-entry. Assume that state `C` in Figure 13 is active, and transition β is taken (leaving superstate `F`). If later transition α is taken, state `C` will be active, since α is connected to an H*-entry.

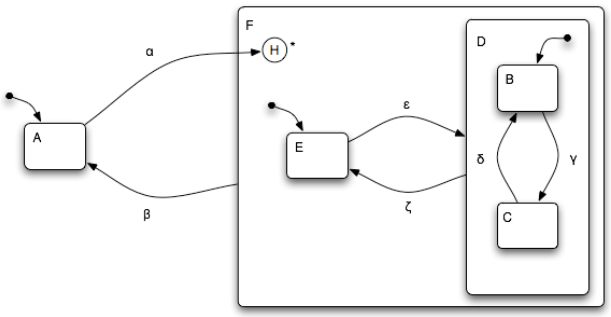

Figure 13: Superstate `D` is entered through an H*-entry.

Having the possibility to utilize history functionality, an obvious requirement is to also clear this memory and enter an aggregation as normal. We will introduce the concept of actions and activities before this property is defined.

4.3 Actions and Activities

A transition action in FSM can be performed when a transition fires, which is denoted at the transition condition after a '/' character. An action is assumed to be performed instantaneously in ideally zero time.

Statecharts also defines activities that, opposed to actions, are performed in non-zero time, and are used to carry out tasks of some sort. For each activity ∂, the following two actions are defined: start(∂) and stop(∂) which are true when an activity starts and stops, respectively. Also, a new condition is defined: active(∂), which is true when ∂ is active.

In SFC, actions are associated with a state instead of being executed upon a transition being fired. Actions in SFC are not instantaneous as in Statecharts and may also be conditional.

4.4 CLH

With the definition above, a special action called clear-history(state), clh(state), can now be defined. When clh is performed, the history at the level of the state is reset. Just as with the H-entry, it is possible to perform a clear-history down to the deepest level. This action is consequently called clh(state*). Consider the graph in Figure 14 and assume that state C is active when transition β is taken and clh(F) is performed.

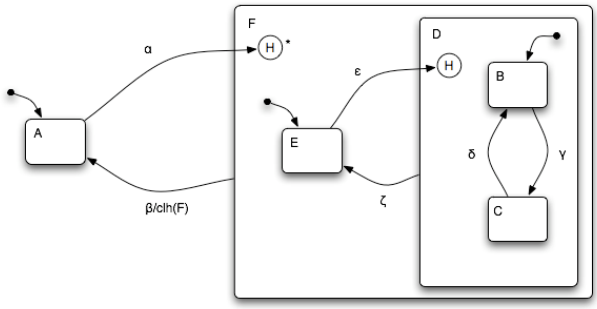

Figure 14: The history of superstate F is reset when transition β is taken.

If transition α is subsequently taken, the choice stands between state D or E, and since clh(F) has been performed at this level, the default arrow, and consequently, state E will be active. Note that if now transition ε is taken, state C will be active, since no clh occurred at this level.

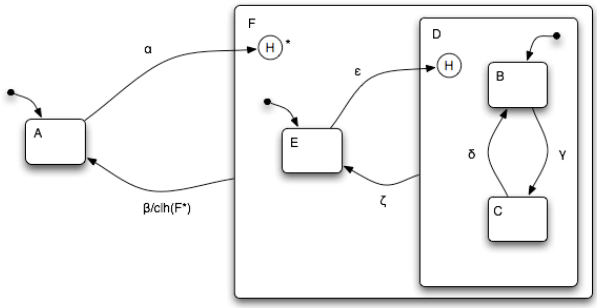

Figure 15: The history of superstate F and all descending substates are reset when transition β is taken.

In Figure 15 clh(F*) is performed instead. If now transition α is taken, state E would be entered. If transition ε is taken, it would result in state B being active, since all superstates are entered through their respective default arrows on all descending levels due to the earlier performed recursive clh.

4.5 ModeGraph History and CLH

The ModeGraph equivalence of the History junction is the resume port. When the resume port fires, the Composite is simply activated. This means that a superstate that is always entered through a history junction, like the one in Figure 12, is directly implementable in ModeGraph by always entering through the resume port, like in Figure 16.

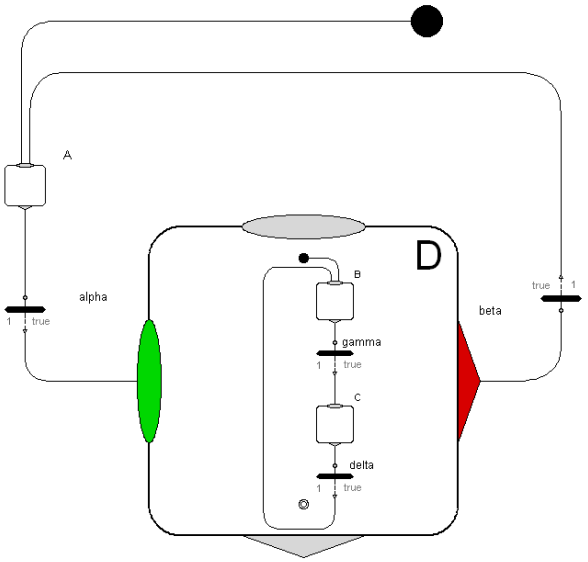

Figure 16: ModeGraph Composite being entered only through the resume port, compare with Figure 12.

Note that when a Composite is suspended, all states all the way down the hierarchy keep their current state, which actually corresponds to the H*-entry. Figure 17 is the ModeGraph implementation of Figure 13. Clear History is performed in ModeGraph upon normal entry through the inPort of a Composite.

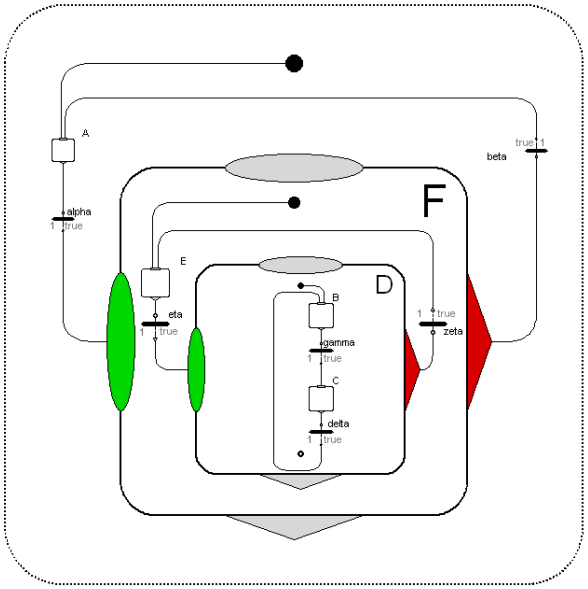

Figure 17: Two nested ModeGraph Composites that are both entered through their resume ports, compare with Figure 13.

5 Parallelism

Parallelism and synchronization are important properties of a state machine to prevent exponential blow-up of the number of states as complexity grows. Assume, for example, two subsystems having x and y states, respectively. When executing in parallel, the number of states would obviously be x + y. However, realizing the system without the parallel states would require x·y states.

Orthogonality provides the possibility to have several superstates executing in parallel. Assume state `D` being the orthogonal product of states `B` and `C`, `D` = `B` × `C`, then `D` is said to be the AND-decomposition of `B` and `C`, see Figure 18.

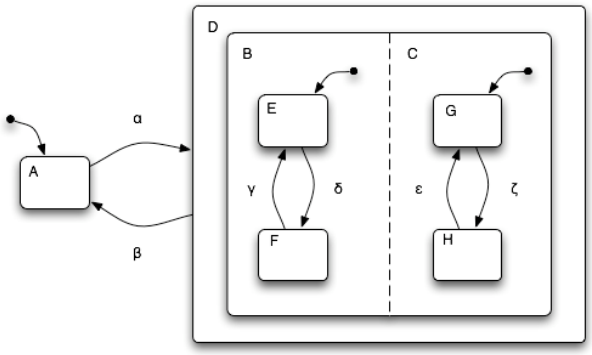

Figure 18: Superstate `D` is the orthogonal product of `B` and `C`.

In practice, it is common to graphically omit the surrounding orthogonal product state, and in this case instead connect transitions directly to the `B` × `C` state.

Another important aspect of subsystems running in parallel is synchronization. An orthogonal product of states should provide the possibility of only being left if a particular set of states is active. In Statecharts, this is performed by using guards on a preemptive transition originating in the orthogonal product state. This can successfully be used to let sequences synchronize before continuing further execution.

Being a sequence-control-oriented formalism, SFC/Grafcet implements parallelism somewhat differently compared to the illustrated example. In SFC, a transition can be split up in parallel paths. Consequently, several paths can be joined by an AND-junction. This sequential approach suits its sequence control purposes very well, and supports synchronization in a natural way.

5.1 ModeGraph Parallelism

The existing StateGraph Parallel component follows the Grafcet/SFC tradition by dividing one connection into a new given number of subpaths that are later joined to a single connection. Hence, synchronization is implicitly demanded of parallel branches. However, it is sometimes useful to have subsystems working independently of each other that never synchronize, as is the case for states `B` and `C` in Figure 18. Those two systems will run concurrently until they are preempted by transition β. Hence, no synchronization will ever occur in this case.

Implementing this in StateGraph will result in a rather messy graph with an unconnected Parallel join component, see Figure 19. This use of unsynchronized subsystems is common in Statecharts, and a more flexible way of implementing orthogonality is thus desirable.

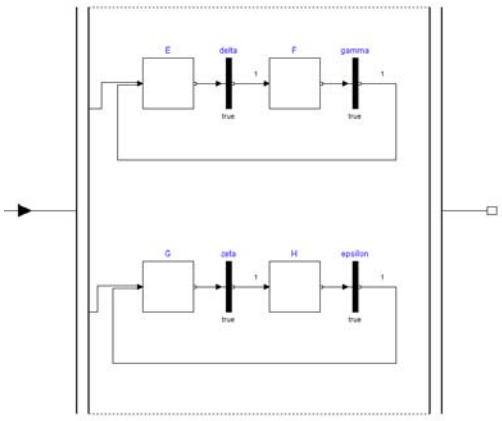

Figure 19: A StateGraph containing two unsynchronized subsystems.

In ModeGraph, a more Statechart-oriented design is introduced without compromising existing possibilities of synchronization. A Parallel component inherits from ModeGraph.Parallel and is placed within a Composite to enable preemption. Figure 20 shows a ModeGraph implementation of Figure 18.

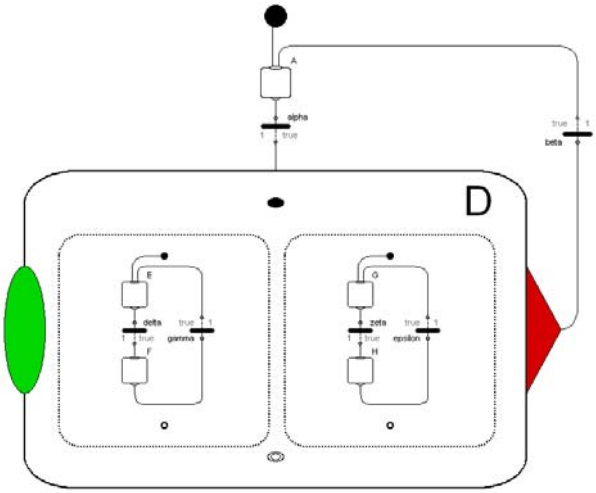

Figure 20: A ModeGraph Composite that contains two independent Parallel subsystems.

As can be seen, ModeGraph incorporates an approach to orthogonality that is very similar to Statecharts. Note, that one or more Parallels are placed in a Composite to provide the possibility of preemption and synchronization of the Parallel children. As shown in the code below, the active flag of a Parallel component is always true. The reason for this is that its activeness should always be decided by the parent Composite. Alternatively, if the Parallel is the root of the graph, it should indeed always be active.

```
output Boolean active
          "= true if parallel step is
           active, otherwise the
       parallel step is not active";
equation
  active = true;
```

One important feature of the ModeGraph Parallel component is that synchronization is still available. Each Parallel block also contains a Boolean variable, `finished`, which is true when the Step connected to the exit port is active.

Assume the scenario in Figure 18 with the modification that transition β can be taken only if Step `F` and Step `H` are simultaneously active. This would result in a ModeGraph implementation shown in Figure 21.

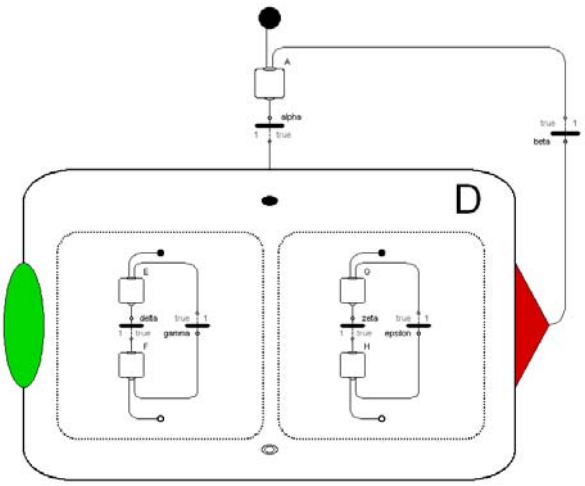

Figure 21: A ModeGraph Composite with two parallel subsystems that must synchronize to allow the Composite to exit. Note that the exit ports of the Parallel components are now connected.

To utilize exit connectors of the Parallel component, it is required to set the parameter `withExit` to true. If `withExit` is false, `finished` will be set to true. This becomes useful when synchronizing Parallel states with exits when there are additional Parallel states without exits present in the same Composite.

The new approach of parallelism supports safe graphs in a natural way. As stated in [5] the Parallel and Alternative components in StateGraph are vulnerable to misuse. The problem is that the Alternative/Parallel components are instantiated at the same level as their branches. This makes it possible for a user to freely connect a branch outside the component without properly synchronizing it, see Figure 22 for an example. Analysis to identify such cases forces unnecessary code overhead.

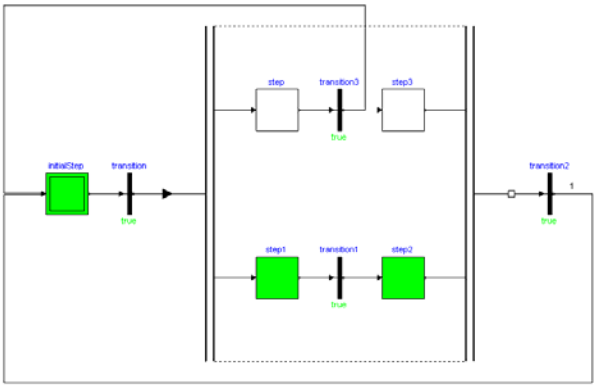

Figure 22: Example of unsafe StateGraph.

In ModeGraph, this kind of misuse is not possible. Since the user is forced to inherit from ModeGraph.Parallel and build the parallel branches within a model, i.e., on a different level, there is no way of connecting to outer Steps or Transitions, since the icon layer is closed.

6 Modelica Mode

To implement Mode-Automata in Modelica, a mechanism for enabling/disabling a block is needed. There must be a way to conditionally evaluate code within a Composite and enable/disable its children. The Modelica `mode` comprises five variables that define the behavior of the inheriting block. The variables define under what conditions equations within the block and its children will be evaluated and when to reset states and outputs. The proposed built-in base class `mode` is defined as:

```
partial block mode
  input Boolean finished = false
     "The execution of the mode
            block is finished";
protected
  Boolean enable = true
     "Enable/disable block and all
                          children";
  Boolean enableSubBlocks = true
     "Enable/disable children";
  Boolean resetStates = false
     "Reset all continuous and
      discrete states of this block
      and all its children";
```

```
Boolean resetOutputs = false
     "When a block is disabled, set
      all its outputs to their start
      values";
end mode;
```

The translator will assert that only one block inheriting from mode at every level is enabled at the same time instant. This will make it possible to ensure consistency of the single assignment rule in the Mode-Automata context.

Naturally, the ModeGraph Step component extends mode, and only one of Steps `A` and `B` in Figure 2 can thus be enabled at a given time instant. A proposed Modelica extension would make it possible to assign a variable `y` as:

```
Step A equation
  y = expr1;
end equation;
Step B equation
  y = expr2;
end equation;
```

The same restriction as in a when-clause applies, i.e., there must be a variable reference on the left hand side of the equal sign (here: `y`). This code will be transformed by the translator and will result in the following single equation:

```
y = if A.enable or
       A.enableSubBlocks then
       expr1
    elseif B.enable or
           B.enableSubBlocks then
       expr2
   else pre(y);
```

The expressions `expr1` and `expr2` are thus defined within `A` and `B`, respectively, and the equation above is generated by the translator to ensure that the single assignment rule is not violated.

As a consequence this means that in the generated code every variable is only defined at one place. For example, it will not be possible to assign the same variable in two parallel branches of a Composite step with two Parallel modes. If this is attempted, an error occurs, since the number of equations and unknowns is not the same. Nearly all other formalisms lack such a property and therefore it is possible to assign to the same variable several times and then non-intuitive rules are used to determine which assignment takes priority. Stated differently, ModeGraph guarantees deterministic variable assignment, whereas most other state machine formalisms have non-deterministic variable assignment.

6.1 Composite Mode

Just like the Step, the Composite component inherits from the `mode` base class. It is by purpose that a Composite and a Step on the same level are mutually exclusive. All components inside the Composite will in turn be gathered and evaluated in the same manner.

The modifiers of the mode block need to be configured according to the desired behavior of the Composite. When the inPort fires, `resetStates` is set to re-initialise all the states of the Composite and its children to behave exactly like if it was indeed the first time it was entered. The attribute `enableSubBlocks` will be true when the Composite is active, enabling children as long as the Composite stays active. When the block is not enabled, all outputs of the Composite and all children should be reset, hence `resetOutputs` is set to true. The mode modifier is shown below.

```
partial block Composite
  extends mode(
  enableSubBlocks = active,
  enable = true,
  resetStates = inport_fire,
  resetOutputs = true,
  finished = allSubBlocksFinished);
```

The proposed built-in operator `allSubBlocksFinished` expands to a check if all children of the mode have their finished variable set to true. Hence, if `allSubBlocksFinished` is true, the Composite may be left through the outPort, since its `finished` flag becomes true.

6.2 Parallel Mode

The final discussion relates to the Parallel Component. Since we think in terms of an orthogonal product, A×B, several Parallel components will indeed be simultaneously active. To avoid violation of the Mode-Automata semantics, the Parallel component is not itself a mode, but contains sets of modes. Since the sub-components are not instantiated at the same level as the Parallel components this does not conflict with the Mode-Automata theory.

Since the Parallel component is not a mode, it is not conditional. There is, however, no need for this, since Parallels are placed inside Composites, and thus 'inherit' the conditional behavior of the parent Composite. Note that a Parallel can be placed at the top level. In fact, this is the intended way to define a top level ModeGraph, since the top component of a graph should always be active.

7 Application Example – Harel's wristwatch

When David Harel introduced Statecharts in [2], he identified and mapped the behavior of a Citizen Quartz Multi-Alarm III wristwatch using the new semantics. This complex, yet comprehensible graph has been realized in ModeGraph as a case study. Selected parts of the ModeGraph implementation of Harel's wristwatch [2] will be used to illustrate the functionality of the mode concept. The main interface of the ModeGraph implementation is shown in Figure 23. It is comprised of a main display, buttons for interaction, and indicator lamps to show the status of the alarms.

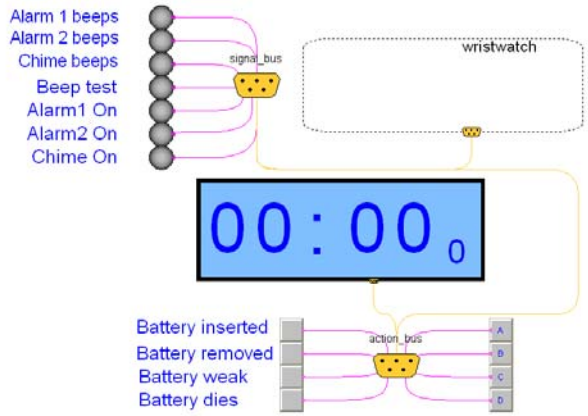

Figure 23: ModeGraph Wristwatch main window.

More information about this implementation of Harel's wristwatch can be found in [3].

An example where the mode semantics becomes very convenient can be found in the time update mechanism of Harel's wristwatch. In update mode, different time quantities can be traversed by pressing a button, `c`. When another button, `d`, is subsequently pressed, the quantity defined by the active state is incremented, see Figure 24.

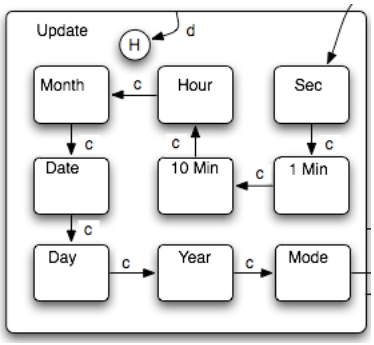

Figure 24: Update mechanism of wristwatch.

The ModeGraph realization of Update is shown in Figure 25.

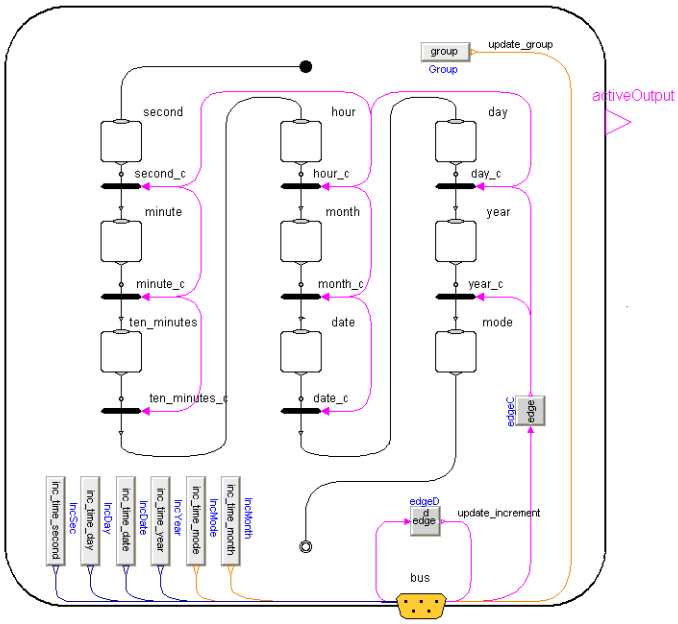

Figure 25: ModeGraph realization of Update.

Declaration of the Step components should according to the proposed mode declaration look like:

```
Step second equation
  inc_time_second = 1;
end equation;
Step minute equation
  inc_time_second = 60;
end equation;
```

Step day **equation**
inc_time_day = 1;
end equation;

Hence, `inc_time_second` would, e.g., be automatically gathered into a single if-statement like:

```
inc_time_second =
    if second.enable
       or second.enableSubBlocks then
       1
    elseif minute.enable
        or minute.enableSubBlocks then
       60
    ...
    else pre(inc_time_second);
...
```

Harel's wristwatch contains a state, `chime-status`, shown in Figure 26. This state controls the chime function that is an alarm that sounds every whole hour that may be either enabled or disabled. Additionally, when enabled, it can be either quiet (the default) or beeping every time the clock reaches a whole hour. Notice that when `chime-status` is active, it can be left regardless of which of the internal states is active. The ModeGraph realization of `chime-status` is shown in Figure 27. Recall that every time a ModeGraph Composite turns inactive,

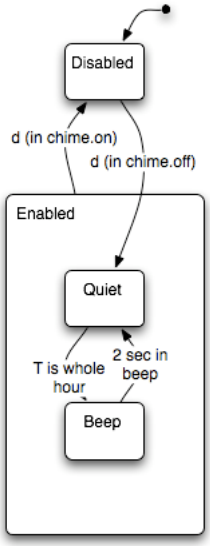

Figure 26: Chime – Status.

interactions between all child states are frozen and no code within the block is evaluated. Hence, if the Composite is activated anew, the last active sub-blocks will once again be active. When entering state `enabled`, sub-state `quiet` should be activated by default. In the ModeGraph realization, the step representing state `quiet` is connected to the entry point. Hence, entering `enabled` through the inport, `resetStates` becomes true, and the Step connected to the entry point will be active.

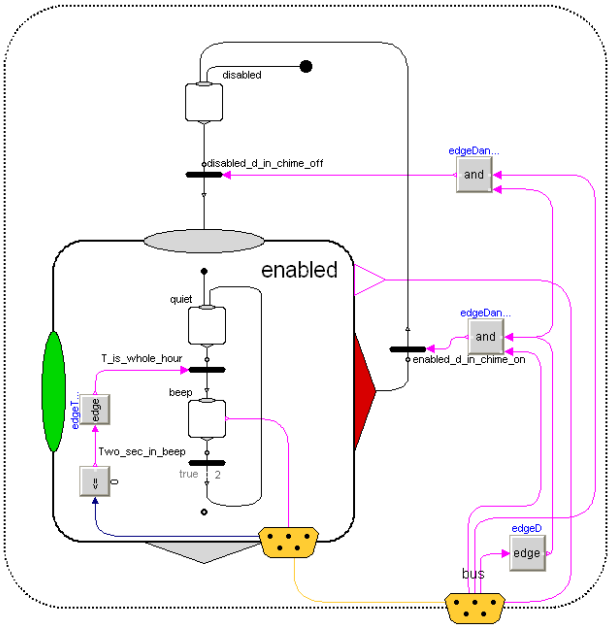

Figure 27: ModeGraph realization of Chime-Status.

The state `stopwatch` in Harel's wristwatch is a good example of the need of flexible parallel states that support easy synchronization. The Stopwatch can either display zeros or the running/frozen time, depending on the context of the parallel states `display` and `run`, see Figure 28.

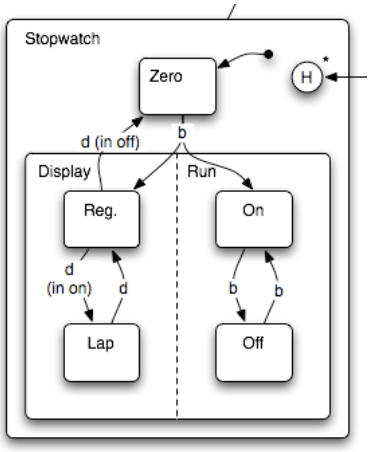

Figure 28: Stopwatch.

When the states `display` and `run` are entered, the stopwatch starts running (state `on`) and displays regular time (state `reg`). Pressing button `b`, the user can turn the stopwatch on/off. Pressing button `d` has different meanings depending on the current active state of `run`. If the stopwatch is running, pressing button `d` switches between display modes `regular` and `lap`. If instead the stopwatch is in state `off`, button `d` is used to exit to state `zero`, thus resetting the time of the stopwatch. The ModeGraph realization of the stopwatch is shown in Figure 29.

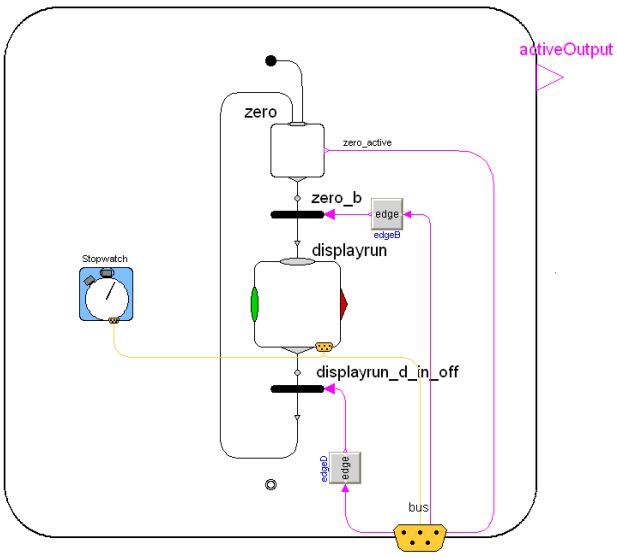

Figure 29: ModeGraph realization of Stopwatch.

An additional Composite `displayrun` is introduced to encapsulate the two parallel states `display` and `run`, see Figure 30. The transition condition *d(in on)* in Figure 28 becomes true when button `d` is pressed and state `on` is active. It is realized in ModeGraph by the state `on` (in Parallel `run`) sending its state out on the bus, which is read by the transition `reg_d_in_on` located in the Parallel `display`. This is a good example of how inter-mode

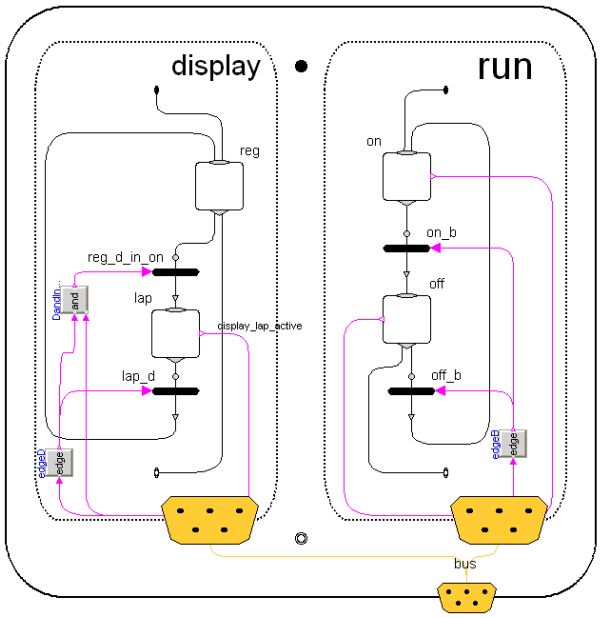

Figure 30: Contents of the ModeGraph Composite `displayrun`.

communication can be performed with expandable connectors, often called buses. This is an important difference in ModeGraph compared to other types of state machines. Since ModeGraph is implemented in Modelica and modes are basic blocks, the variables in a mode block are local variables. In other formalisms, variables are usually available as global entities on all levels. For embedded systems the ModeGraph approach is safer, since variables of composites are encapsulated.

Also note, in Figure 30, how Steps `off` and `reg` are connected to the exit points of their respective Parallel parent. When both these Steps are active, Parallels `run` and `display` both declare themselves finished, which enables transition `displayrun_d_in_off` in Figure 29 to fire, since its `allSubBlocksFinished` attribute will return true.

What has just been discussed is the core functionality of the ModeGraph library. The possibility of simply ignoring equations within a disabled mode, that also are guaranteed to be mutually exclusive with respect to other modes on the same level, reduces code and introduces powerful properties allowing equations to be associated with modes.

8 Conclusions

In this paper the ModeGraph library has been introduced. The motivation for ModeGraph originates in the inadequacy of StateGraph in terms of implementing Statechart-oriented state machines. ModeGraph offers improved flexibility of graphical modelling of state machines, regardless if they are SFC/Grafcet- or Statechart-oriented. Graphically, ModeGraph provides a modern look and feel with components based on Modelica 3.0 graphical annotations. Furthermore, the Mode-Automata semantics offers a convenient way of managing complex conditional structures for the user. Large-scale systems will successfully draw advantage of the fact that only relevant parts of the code (i.e., the code of the current active modes) are evaluated. The conditional structure also prevents the user from unintentionally abusing the available components in dangerous ways without having extensive code overhead.

9 Acknowledgement

This work was in parts supported by the ITEA2 EUROSYSLIB project (http://www.itea2.org/public/project_leaflets/EUROSYSLIB_profile_oct-07.pdf).

References

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

[2] 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. http://www.inf.ed.ac.uk/teaching/courses/seoc1/2005_2006/resources/statecharts.pdf

[3] Malmheden, M. (2007): **ModeGraph – A Mode-Automata-Based Modelica Library for Embedded Control.** Master's thesis, Department of Automatic Control, Lund University,Sweden. http://www.control.lth.se/database/publications/article.pike?artkey=5808

[4] Maraninchi, F. and Rémond, Y. (2002): **Mode-Automata: a New Domain-Specific Construct for the Development of Safe Critical Systems.** http://www-verimag.imag.fr/~maraninx/SCP2002.html

[5] 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. http://www.modelica.org/events/Conference2005/online_proceedings/Session7/Session7b2.pdf

[6] AFCET. (1997): **Normalisation de la representation du cahier des charges d'un automatisme logique**. J. Automatique et Informatique Industrielle.

[7] IEC Standard 61131-1 http://www.iec.ch/