ISSN 0280-5316
ISRN LUTFD2/TFRT--5808--SE

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

Martin Malmheden

Department of Automatic Control
Lund University
December 2007

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
December 2007

Document Number
ISRNLUTFD2/TFRT--5808--SE

Author(s)
Martin Malmheden

Supervisor
Hilding Elmqvist and Sven Erik Mattsson at Dynasim AB in Lund.
Karl-Erik Årzén, Automatic Control in Lund (Examiner)

Sponsoring organization

Title and subtitle
ModeGraph – A Mode-Automata Based Modelica Library for Embedded Control. (ModeGraph – Ett modautomatbaserat Modelica-bibliotek för inbyggd reglering)

Abstract
The ModeGraph library offers improved flexibility of hybrid modelling and incorporates powerful properties of the Mode-Automata semantics. Concepts from Finite State Machines (FSM), Sequential Function Charts (SFC)/Grafcet and Statecharts are efficiently utilised to provide a flexible modelling environment that a broad audience can feel familiar with. Being able to guarantee mutual exclusivity of all modes on each level ensures consistency with regards to the single assignment rule used in Modelica. The current Modelica library for state machine modelling, StateGraph, has proven insufficient in terms of graph safety and efficiency. The goal of ModeGraph is to provide an alternative to StateGraph by introducing extensions to the Modelica language to drastically reduce code overhead and improve performance of modelled graphs.
To enable exported code to be highly modular and easily integrated, AUTOSAR-compliant C-code export has been investigated in the thesis. An increasing number of companies and organisations are embracing the AUTOSAR standard which makes it important to offer compliant interfaces of generated code.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
86

Recipient's notes

Security classification

Preface

First of all, I would like to thank Karl-Erik Årzén at the department of Automatic Control, LTH, for making this thesis possible by putting me in contact with the people at Dynasim.

My deepest gratitude goes to my supervisors, Hilding Elmqvist and Sven Erik Mattsson, for being incredible sources of inspiration and for providing me with guidance along the way. Hilding Elmqvist has additionally contributed to the thesis by proposal of the original idea of ModeGraph and the basic conceptual design of the Modelica Mode concept. Sven Erik Mattsson has contributed by performing all implementation work in Dymola, enabling support for the Mode concept and the conditional execution. Being the main author of the StateGraph library, Martin Otter has been a great support, cordially helping out via e-mail at all hours of the day. Large parts of the work has also been based on his previous work on the StateGraph library. Many thanks goes to Dan Henriksson for vast help with the report, and for spending untiring hours of proof reading.

Furthermore, I would like to thank everyone at Dynasim, who have all been very kind, helping me to accomplish this. Special thanks to Ulf Nordström for always taking time to explain Modelica issues, Dag Brück for sorting out every issue one can imagine - as well as the ones one cannot, and to Erik Areskog for great cooperation with graphical implementation issues. Thanks also goes out to the EXTESSY EXITE team, and especially Marc Höper for professional support of my EXITE ACE issues.

Finally, I would most of all like to thank my family for their unconditional love and support.

Contents

1. Introduction

1.1 Motivation/Overview

Modelica [15] is a multi-domain object-oriented language for modelling of physical systems. Components are described with differential algebraic equations (DAEs) and are constructed very similar to the corresponding physical objects to facilitate understanding. Dymola, Dynamic Modeling Laboratory [13], provides a complete tool for creation, manipulation, and simulation of physical systems based on the Modelica language. A number of standard libraries are available in Modelica covering a wide range of application fields.

A state machine is a behaviour decomposed into a number of states that can change upon certain stimuli, also called input. This kind of system is also known as a reactive system. If the state machine has a finite set of states it is often called a finite state machine (FSM). However, finite state machines are rather limited due to the lack of hierarchical organisation of states. FSM has been extended in several ways to also incorporate hierarchy as well as support for memory and preemption. This enables larger and more complex state machines to be modelled without losing readability. Modelling of state machines is currently supported in Modelica by the StateGraph library [8].

Dymola provides an add-on option of C-code export of a given Modelica model. The exported C-code routine can be integrated either in Hardware In the Loop (HIL)/Software In the Loop (SIL)-simulation or on a microcontroller. In the automotive industry there is a need for an open industry standard for Electric/Electronic (E/E) architectures. AUTOSAR (AUTomotive Open System Architecture) [10] is a collaboration between numerous leading automobile manufacturers to provide an improved standard that allows more complex software architectures while still keeping them maintainable.

This thesis consists of two parts. In the first part an improved Modelica library for modelling of reactive systems will be introduced. The second part treats AUTOSAR-compliant code generation from Modelica models. The two parts are closely related, since control systems are often defined as reactive systems and automatic code generation is an essential step in the development of embedded controllers. It should be pointed out that both parts of this thesis are self-contained and can be read independently of each other.

1.2 Problem Definition

Insufficient Support for State Machines in Modelica

The current Modelica library for state machine modelling, StateGraph, has in several ways proven insufficient. The need for a simplified implementation, as well as improved versatility of state machine modelling, is tangible. Currently, the implementation of hierarchal properties is complex and a refined implementation would benefit from additions to the Modelica language. The risk of unsafe graphs

is today solved with analysis forcing overhead code to be present in the components. Another drawback of the StateGraph library is that all equations are always evaluated, even though they might not be located in an active state. Furthermore, with the available library, there is no way of guaranteeing mutual exclusivity between states. A consequence of this is that an equation cannot be directly associated with a specific state, which makes it cumbersome and error prone to implement larger state machines.

Need For C-code Export With Standardised Interface

For code generation, it is required to support standards utilised in the industry. The automotive industry currently moves toward a change of standards with more and more companies embracing the AUTOSAR standard. This motivates Dymola support of a C-code export option meeting the AUTOSAR standard.

1.3 Goal

A New Modelica Library for Modelling of Reactive Systems

A new Modelica library for modelling hierarchical state machines will be proposed. The new library should be easy to learn and intuitive to use, since these properties together with visual descriptiveness embody the strengths of graphical modelling of reactive systems. Basic components should be conceptually easy to grasp and unambiguous to use. The library should, furthermore, be capable of handling extended state machine properties, such as hierarchy (meta states), orthogonality (parallel sub-states), synchronisation, and preemption.

Naturally, support for all the functionality of the StateGraph library with a new simplified implementation is required. Necessary modifications to the Dymola translator needed to simplify and to optimise code should be suggested. The library should support proposed extensions to the Modelica language enabling mutual exclusivity between states. This would introduce the possibility of equations being assigned depending on the current state, enabling support for integrated systems. The library should ensure safe state machines in terms of avoiding algebraic loops and unreachable graphs. An application example will be used as a feasibility and benchmark study.

Dymola Support of AUTOSAR-compliant C-code

It should be possible to export selected parts of a Dymola model, e.g., controllers defined using the new library for reactive systems, to AUTOSAR compliant C-code. The user should only be required to specify what parts of the model that should be exported, and possibly also additional information, such as sample time, priority, microcontroller placement etc. It should be simple and straightforward to integrate and perform software in the loop-simulations in a co-simulation environment. When the native code is eventually placed on a microcontroller and operates on a physical plant, only an insignificant amount of configuration should be required to get the system operational. If there are several communicating parts of the model that are exported as separate entities, the aim is to ensure that communication works regardless of the physical placement. Since the optional C-code export add-on is already present in Dymola, the scope of this part of the

thesis is to investigate and make sure that the export is performed in coherence with AUTOSAR standards.

2. Background

2.1 Modelica

Modelica is a free object-oriented language for modelling of physical systems. The language covers multiple domains, such as electrical, mechanical, thermal, and fluid systems. A standard library is provided containing numerous standard components within those domains, including a wide range of components stretching from SI-units, defined physical medias, and mathematical functions to control system components, such as PID-controllers and hierarchical state machines. In addition, there is a number of commercially available libraries supporting modelling of vehicle dynamics, air conditioning systems, hydraulics, vehicle power trains, and pneumatic systems.

Physical components are modelled in Modelica by defining mathematical equations that describe the behaviour of the modelled entity. Graphical representation of components facilitates organisation of larger modelled systems. A component comprises differential algebraic equations (DAEs) that are manipulated by a tool to achieve efficient simulation of very large systems. Even without any particular understanding of Modelica, it is possible for an engineer to identify and comprehend rather complex systems just by investigating the composition of graphical components.

2.2 Dymola

Dymola [13] is a commercial tool that supports modelling and simulation of physical systems based on the Modelica language. Hence, modelling and simulation of integrated systems has strong support in Dymola and enables engineers to incorporate several engineering disciplines at once. To fully take advantage of the capabilities of the Modelica language, Dymola provides state of the art symbolic manipulation to enable large systems (100000+ equations) to be simulated on a standard personal computer. Dymola supports object-orientation and re-use of components and the environment is open in the sense that any component can be modified and adapted to the application needs.

2.3 StateGraph

The current Modelica library for modelling hierarchical state machines is called StateGraph and is introduced in [8]. StateGraph is in many ways very similar to Grafchart [2] and its Java implementation JGrafChart [3]. Grafchart was introduced as a hierarchical version of Grafcet [9] and combines flow chart semantics with the possibility of hierarchical structuring. Grafcet was in 1993 accepted as an international standard (IEC 61131-3) by the International Electrotechnical Commission (IEC) and additionally in 1998 in a standard named IEC 848. In these standards, Grafcet is referred to as Sequential Function Charts (SFC).

Hierarchical structuring was presented by D. Harel in the Statechart semantics [5], and additionally the concepts of preemption and memory were introduced in this paper. These has proven to be very powerful properties that have made it possible to model larger, and significantly more complex reactive systems.

Like Grafchart, StateGraph contains all the basic components of Grafcet: Steps, Transitions, Parallel Split/Join (Distribution/Junction AND in Grafcet), Alternative Split/Join and Composite steps. The Composite makes it possible to define subsystems that can be suspended and resumed to replace the Grafcet 'macro actions' and introduce proper hierarchy. Composites support preemption of execution when one of its 'suspend' ports fires and execution of the Composite is continued when one of its 'resume' ports is set.

Nested steps and Composites communicate through inner/outer connectors called 'CompositeStepPort' that inform the parent state about how many active Steps it currently contains. A variable declared as 'outer' has a corresponding variable outside the component with the exact same name that are treated as the same variable. When the Composite is suspended, all its children receive a suspend signal through the CompositeStepPort and store away their state. When a Composite is later resumed, this occurrence is propagated down the hierarchy as well, and all child steps set their active flag to their respective stored values.

As previously mentioned, the action language used in StateGraph is Modelica. A consequence of this is that actions within a step cannot be defined as entry-, normal- and exit-actions, as in Grafchart. This is due to the Modelica *single assignment rule*, which states that every variable is defined by no more than one equation. This introduces a very nice feature – deterministic behavior of the graph is guaranteed by the Modelica translator.

However, inconsistencies have been found in the StateGraph library. Assume, for example, that a Composite is entered only through the resume port to always continue where it was last preempted. This results in undefined behaviour when entering the Composite the first time. If entry is always performed through the resume port, the graph will have no active states the first time the resume port is fired, since all child states are initially inactive. Hence, it is desirable that the active state of the Composite is independent of its children and that a default state is present, that may be reset upon inport entry. A better way to solve this logic is required to improve the flexibility of the component.

Another issue is orthogonality for autonomous subsystems. An autonomous subsystem refers in this context to an independent set of states and transitions. Assume for example that a meta state contains several autonomous subsystems running in parallel. They have no interaction with each other, and need not synchronise to allow the meta state to suspend. This kind of situation is common in Statecharts, but it does not harmonise very well with the flow chart semantics tradition of Grafchart/Grafcet. It is implementable, but will require modifications to the Parallel Step, and turns out rather messy.

2.4 AUTOSAR

The AUTOSAR initiative is a collaboration between automobile manufacturers to achieve an open standardised architecture. The AUTOSAR partnership strives to properly handle increasingly complex E/E software without compromising quality

aspects. Other goals include enhanced portability and possibility of effortless exchange of hardware and software in automotive systems throughout the whole product life cycle. Software developers should have to spend less time on hardware integration/interaction and instead put all focus and effort on development of the actual software components. Enhanced modularity alongside clear organisation and separation of working areas are means to increase productivity and allow automotive systems to grow in complexity and still remain maintainable.

An introduction to AUTOSAR is given in [10]. Only a short orientation of the concepts of AUTOSAR will be given here. As mentioned earlier, AUTOSAR aims to modularise automotive systems to enable maintainability, portability and relocability. This is achieved by carefully defining the different types of components of the complete system and their interaction among each other. A conceptual view of the AUTOSAR architecture can be seen in Figure 1 below.

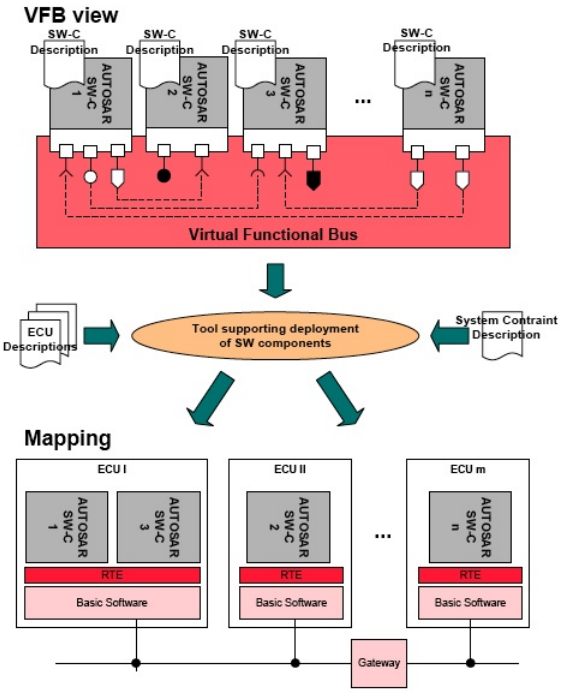

Figure 1: Conceptual view of AUTOSAR Software Components being mapped on to Electronic Control Units (ECU).

The AUTOSAR Electronic Control Unit (ECU) Architecture can be divided into four distinctive layers as can be seen in Figure 2 below.

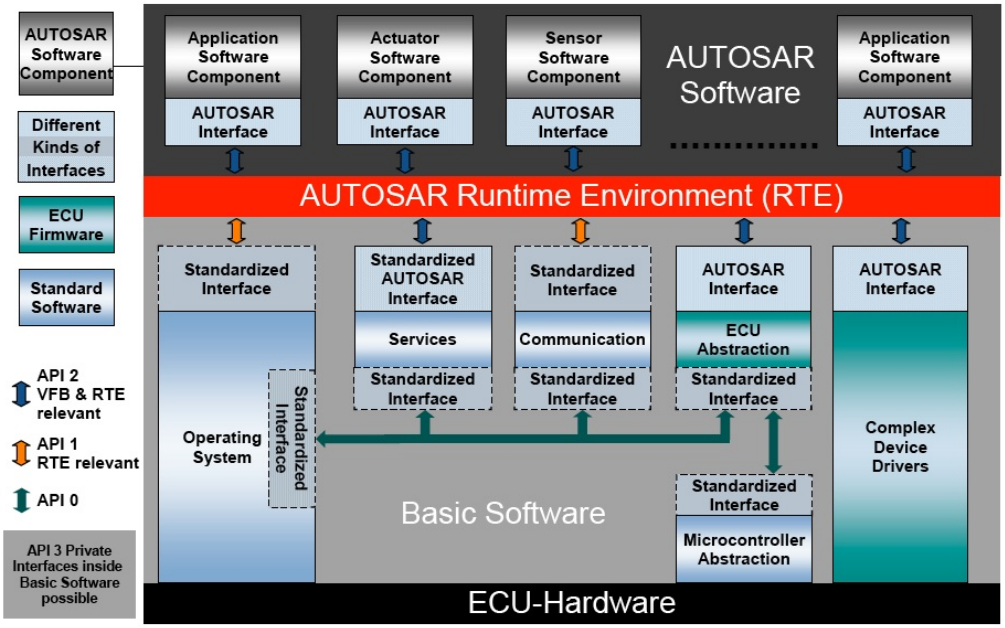

Figure 2: Organisation of the AUTOSAR ECU software.

The ECU-Hardware layer includes the hardware microprocessors controlling various parts of the automotive system.

The software operating at the lowest level interacting with the ECU hardware is called Basic Software (BSW) and is comprised of:

- The Operating System located at the ECU.
- A Microcontroller Abstraction that provides a standardised interface against the actual hardware making sure that higher-level software does not need to access the hardware directly.
- Services and Communication taking care of memory, diagnostics, I/O management and communication-specific tasks.
- Three different AUTOSAR Interfaces, each communicating with the Real-Time Environment (RTE) to provide access to the previously mentioned BSW modules.
- The Complex Device Drivers allow direct access to ECU hardware for particularly critical tasks.

The AUTOSAR Software Component (SW-C) is the piece of high-level software that is placed on an ECU. Communication between SW-Cs is handled by the RTE that organises inter- and intra-ECU information exchange and thus provides a communication abstraction between the SW-Cs. Hence, this abstraction supports relocability of SW-Cs since they are independent of their respective ECU location. Interaction with hardware is handled by the BSW layer.

3. ModeGraph

In this section the ModeGraph library will be thoroughly explained and excerpts of the implementation will be presented. General concepts taken from FSM, Statecharts and SFC will be used as references and benchmarks to prove the feasibility and applicability of ModeGraph.

3.1 Steps and Transitions

An FSM describes a behaviour by decomposing it into a distinct finite set of states visualised 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 fire 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 example, see Figure 3, 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.

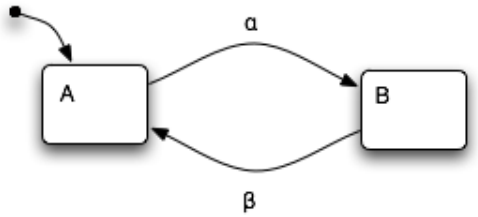

Figure 3: Simple state machine with two states and two transitions.

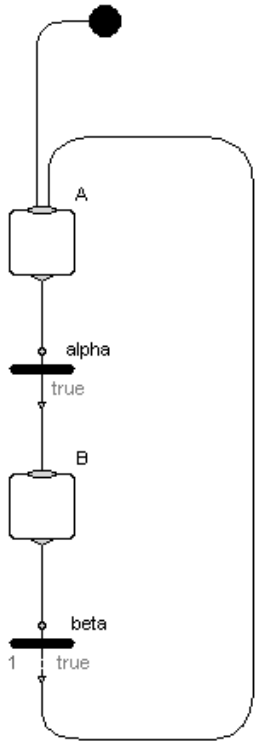

Figure 4: Simple ModeGraph containing two Steps and two Transitions.

Inheriting much of the semantics from StateGraph, the basic components of ModeGraph are Steps and Transitions which are both similar to the corresponding StateGraph objects. Figure 4 shows the ModeGraph equivalent of Figure 3. We will proceed to describe the Steps and Transitions in more detail.

Steps

There are two types of Steps: a regular Step and a StepWithSignal - both inheriting from the partial component PartialStep. 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. Because of this, the state is called `localActive` internally in PartialStep:

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

The function `anyTrue` is a function that iterates through the argument array of connectors and returns true if any of them is true. The variable `active` is set to `localActive` in the modifier when the Step inherits from PartialStep. The variable `newActive` is the state of the Step in the next iteration, hence `localActive` is set to **`pre`**`(newActive)`. A Step is said to be available to the successor Transition when `localActive` is true. A Step has multiple in- and outports i.e. several transitions can lead to and from the Step respectively. 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. The concept of Modes will be thoroughly explained later in this chapter. This requires restrictions on the outPort fire mechanisms which will be explained in detail below.

Transitions

Transitions are used to decide when a change of state should be performed. In Modelica, the behaviour of a transition needs to be modelled as a separate entity. The reason for this is that it needs to contain different types of conditions and optional delay logic, that cannot be included in a mere connection. A basic Transition will check if its predecessor Step is stated available and evaluate if its firing condition is true (visualised by the condition being coloured 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 can be viewed in Figure 5.

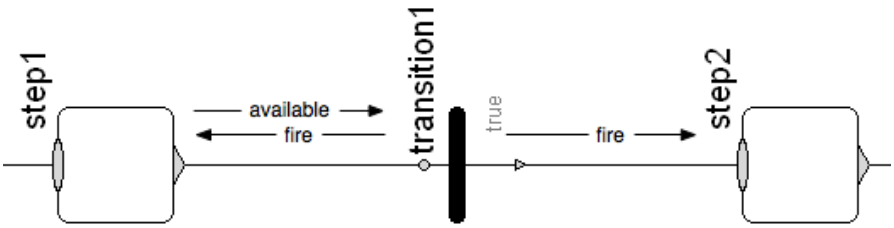

Figure 5: Signal flow between Steps and Transitions.

Communication Between Steps and Transitions

Consider the sequence of Steps and Transitions with true conditions in Figure 6.

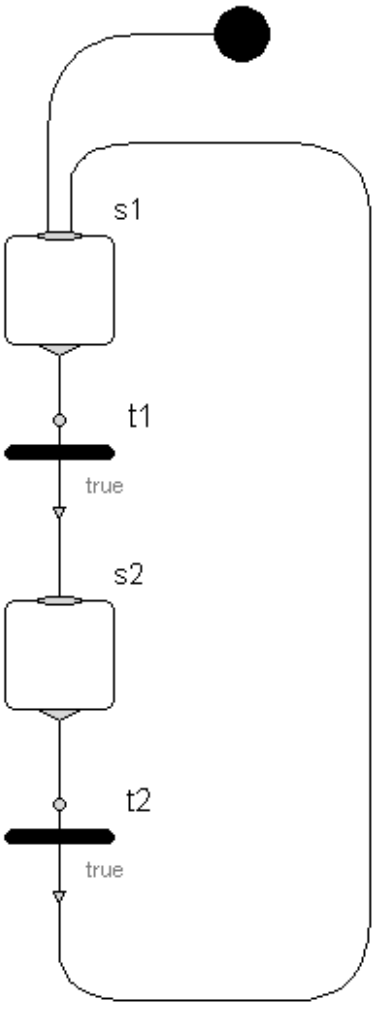

Figure 6: An illegal loop of Steps with Transitions being true between them.

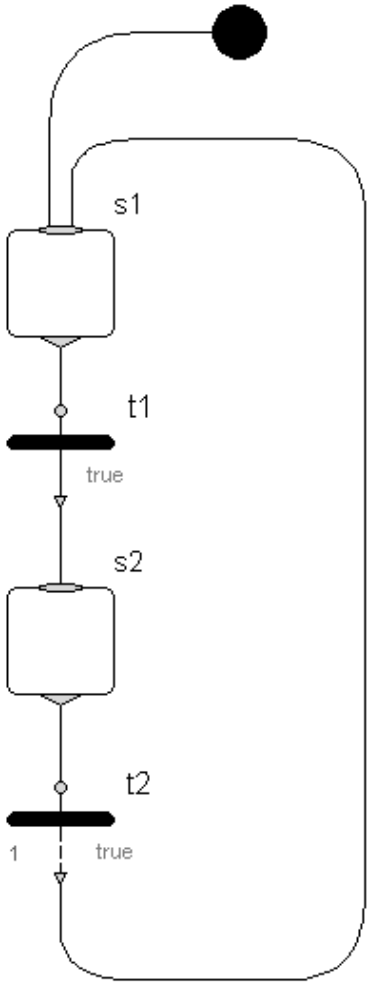

Figure 7: A graph with a sequence of Steps containing a delayed Transition to break the loop.

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 6.

```
s1.newActive      = (pre(s1.newActive) and not t1.fire)
                    or t2.fire or entry.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 the transition t2 in Figure 7. This type of Transition is called DelayedTransition and requires additional equations to decide how long after the condition becomes true it is allowed to fire. The parameter that defines the duration for which the fire conditions need to be true is called `waitTime`. DelayedTransition has the following code:

```
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;
```

As mentioned above, Steps can have multiple in-ports/outports and only one Step is allowed to be simultaneously active at every level. This requires priority among the outports. The most intuitive way is to let the index of the port array decide in what order ports are allowed to fire. Low index represents high priority. The code defining the boolean `available` in the outPort connector of the Step has so far been omitted. The `available` flag needs to take priority into account and a port is said to be available if the Step is active and if no port with higher priority fires. The code handling outPort of a Step can be viewed below:

```
for i in 1:nOut loop
  outPort[i].available = if i == 1 then localActive else
            localActive and not outPort[i-1].fire;
end for;
```

Unsafe graphs

Assume that a user for some reason creates a graph containing a loop of true Transitions, as in Figure 6. With the current Steps and Transitions, the graph will translate, but the solver will not be able to converge towards a single active Step. This kind of undefined behaviour is obviously dangerous and thereby not allowed.

To identify cases like this in 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 8.

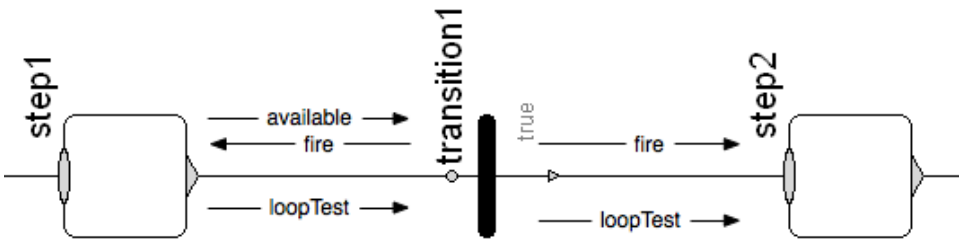

Figure 8: New signal flow with added loop checking.

The idea is to let Steps and undelayed Transitions just pass the signal on, while a delayed Transition will set `loopTest` to true. If only Steps and undelayed Transitions are present in a loop, the translator will recognise an algebraic loop, halt the translation and print an error message. 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 very simple:

- In PartialStep:

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

- In PartialTransitions:

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

- In PartialDelayedTransition

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

However, there are drawbacks to this method. Only because a closed loop contains only undelayed transitions, it does not necessarily mean it is an unsafe graph. If the transitions conditions are restricted, the graph might work fine.

Ideally, a more sophisticated method to ensure graph safety would be preferable. The built-in Modelica functions defineBranch and defineRoot can be used to define a connection, *branch*, through a number of a component's connectors. A branch could successfully be placed in undelayed transitions, and of course none should be present in a delayed transition. If branches are defined between inports and outports of step, composite and parallel components, one could use those branches to analyse the structure of the graph and localise unsafe loops. Exploiting this feature in a sensible manner would enable an elegant solution to the problem with minimal code overhead. However, to utilise this method to the full extent, support in Dymola is needed. Unfortunately, further elaboration of graph safety analysis falls outside the scope of this thesis.

3.2 Encapsulation and Aggregation

The FSM formalism is adequate as long as the modelled behaviour 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 general viability. 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. Remedies for the mentioned problems were introduced by David Harel in Statecharts [5], 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 9, the states B and C share the common property of transition β leading to state A.

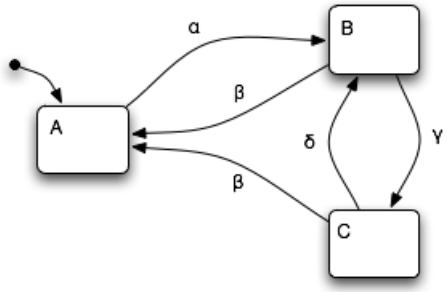

Figure 9: Three states of which two share common properties.

Thus, B and C can be clustered together into state D in Figure 10. Note the improved visual appearance in Figure 10 compared to Figure 9, despite the exact same behavior of states A, B, and C.

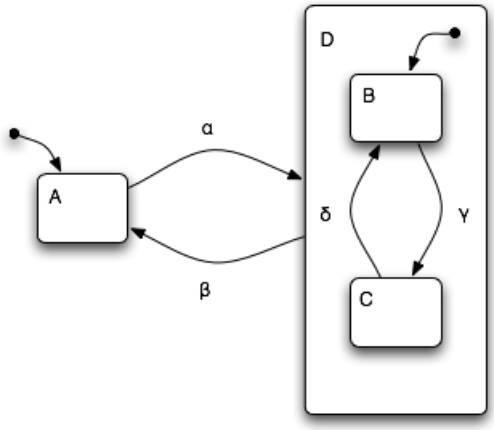

Figure 10: Two states clustered together in a superstate.

Refinement of a state involves identification of a number of child states with unique properties within a particular state. In Figure 10, 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 in Figure 10. Being in one of the sub-states implicitly means also being in the superstate. The superstate D in Figure 10 is said to be the XOR-decomposition of its sub-states.

ModeGraph Composite

Just like StateGraph, ModeGraph allows aggregation of states into superstates. A Composite component inherits from ModeGraph.Composite and has inports and outports, like a regular step, but also suspend ports and resume ports - like in StateGraph. Figure 11 shows a ModeGraph corresponding to the chart in Figure 10.

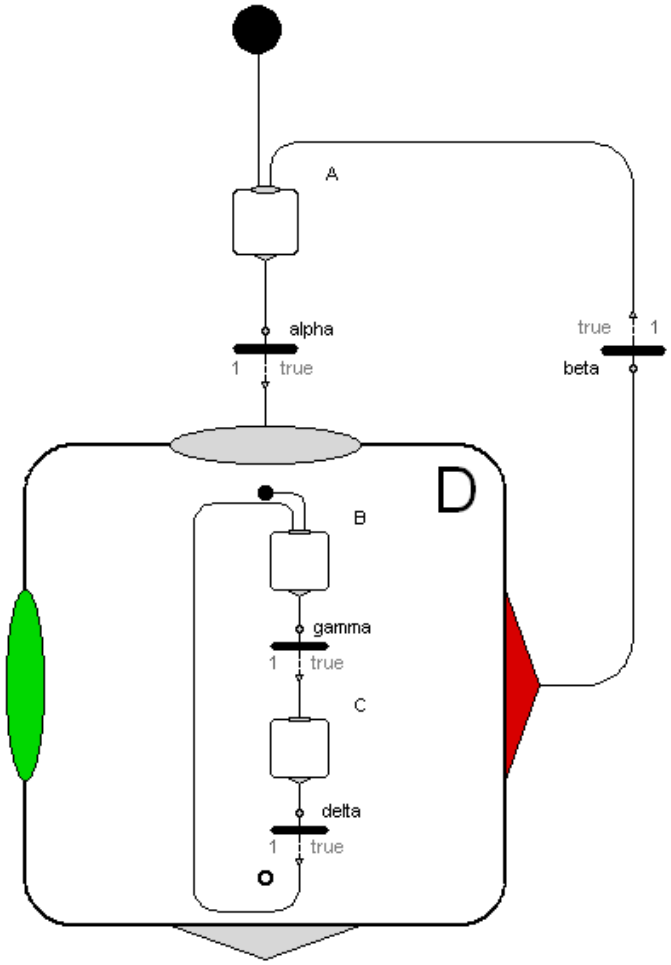

Figure 11: ModeGraph implementation of two steps clustered inside a Composite.

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 Safe State Machines (SSM) [1] but 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 in the Composite and Parallel ModeGraph components an exit port, 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 ensures that the state remains true for one iteration, and then switches to false.

```
Entry entry(fire(start=false));
protected
  Boolean active(start=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 = pre(localActive) or anyTrue(inPort.fire)
    or anyTrue(resume.fire);
localActive = available and not anyTrue(outPort.fire)
    and not anyTrue(suspend.fire);
active = pre(localActive);
```

In the code above, the state of the Composite, `active`, differs from the corresponding variable in Step. Instead of being equal to `localActive` it is instead set to **`pre`**`(localActive)`. The reason for this is to avoid an algebraic loop involving Mode conditions that will be introduced later in this thesis. 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 elaborated in Section 3.6.

3.3 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 sub-states is active, see, e.g. transition β in Figure 10. Of course, a normal exit is possible by having a transition originating in an inner state and targeting an outer. Notice how state D in Figure 12 is only left if transition β is taken when state C is active.

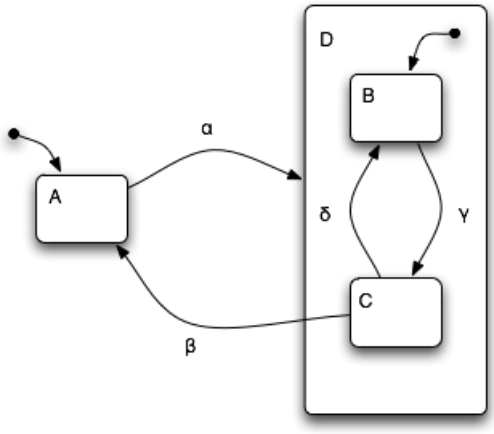

Figure 12: Superstate D can only be left when in sub-state C.

ModeGraph Preemption

A ModeGraph Composite has an array of suspend ports. Recalling the active condition of the Composite, it is clear that after a suspend port fires, the Composite is no longer active. This behaviour 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 prioritisation 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 10 and its equivalent in Figure 11. The deactivation of the Composite does not, explicitly, influence the internal states of the Composite. In the generic setup of Mode parameters, the state of the sub-blocks will be kept when the Composite is suspended. The state of the sub-blocks will be kept, but all internal interaction will be frozen.

3.4 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 reasonable to be able to enter the most recently visited sub-state.

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 active 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 13. If subsequently transition α is taken, state C (and of course also state D) will once again be entered.

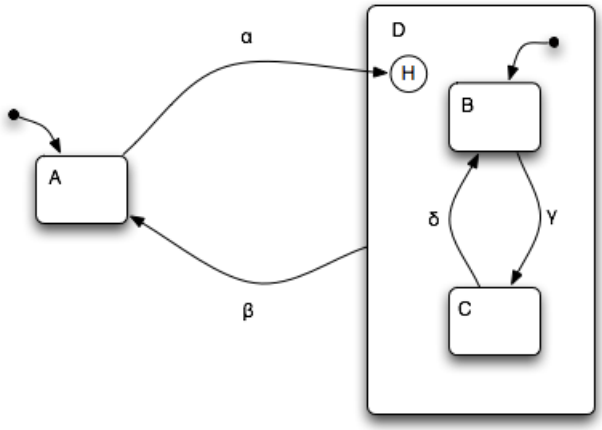

Figure 13: Superstate D is entered through an H-entry.

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 14 is active, and transition β is taken. If later transition α is taken, state C will be active, since α is connected to an H*-entry.

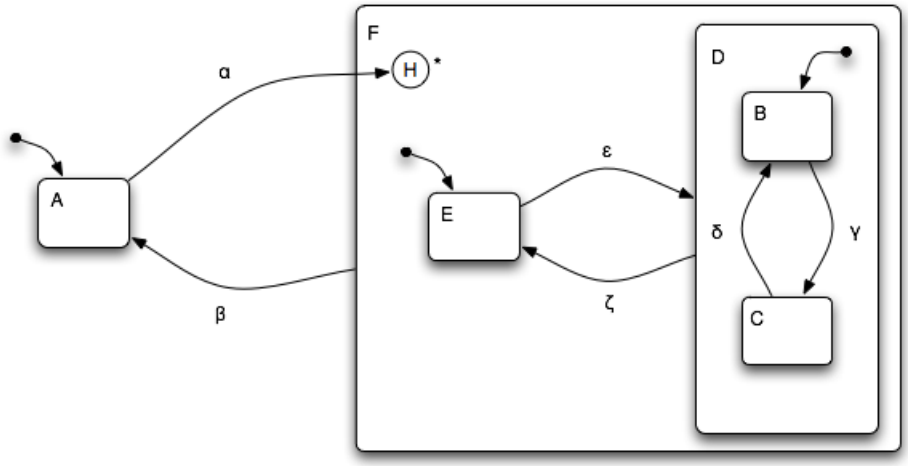

Figure 14: Superstate D is entered through an H*-entry.

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

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 obviously 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.

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 15 and assume that state C is active when transition β is taken and clh is performed. 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 E, will be active. Note that if now transition ε is taken, C will be active, since no clh occurred at this level.

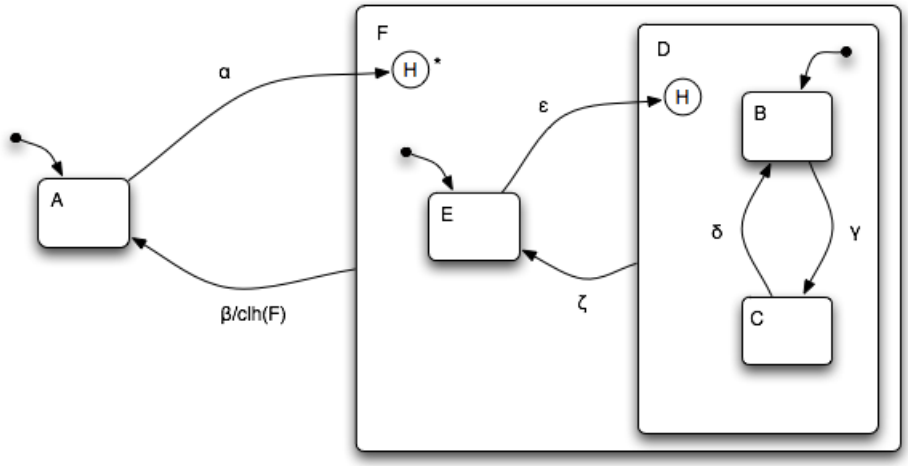

Figure 15: The history of state F is reset when transition β is taken.

In Figure 16, 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.

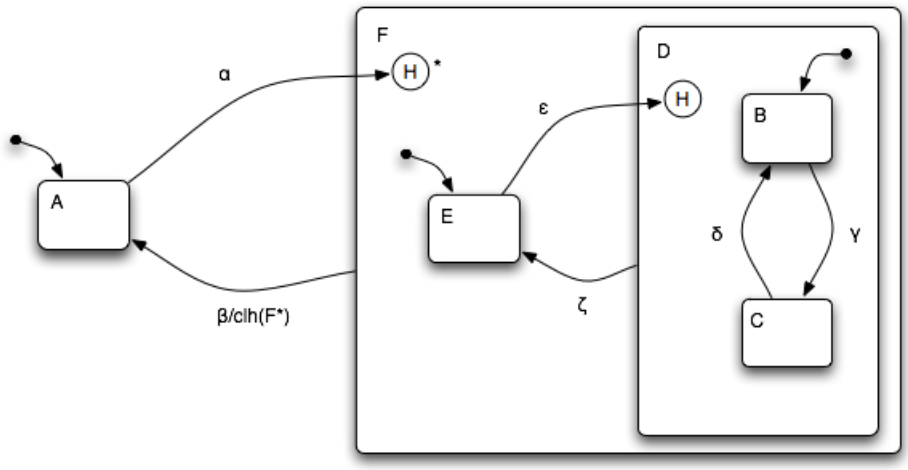

Figure 16: The history of superstate F, and all descending substates are reset when transition β is taken.

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 13, is directly implementable in ModeGraph by always entering through the resume port, as in Figure 17.

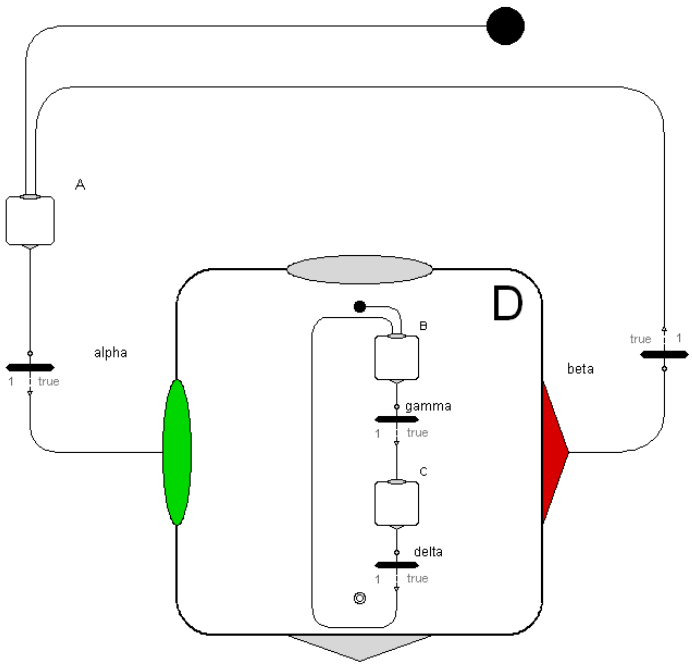

Figure 17: ModeGraph Composite being entered only through the resume port.

Note that when a Composite is suspended, all states all the way down the hierarchy keep their current state, which corresponds to the H*-entry. Figure 18 is the ModeGraph implementation of Figure 14.

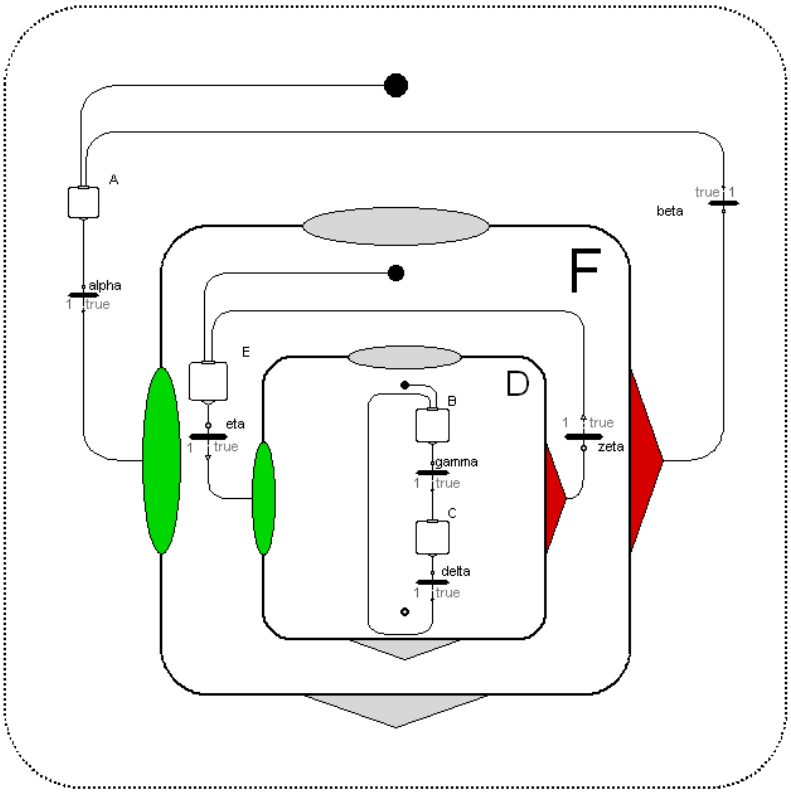

Figure 18: Two nested ModeGraph Composites that are both entered through their resume ports.

Clear History is performed in ModeGraph upon normal entry through the inport of a Composite.

3.5 Parallelism

Parallelism and synchronisation are important properties of a state machine to prevent the exponential blow-up of 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, realising 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 19. In practice, it is common to omit graphically the surrounding orthogonal product state, and instead connect transitions in this case directly to the B × C state.

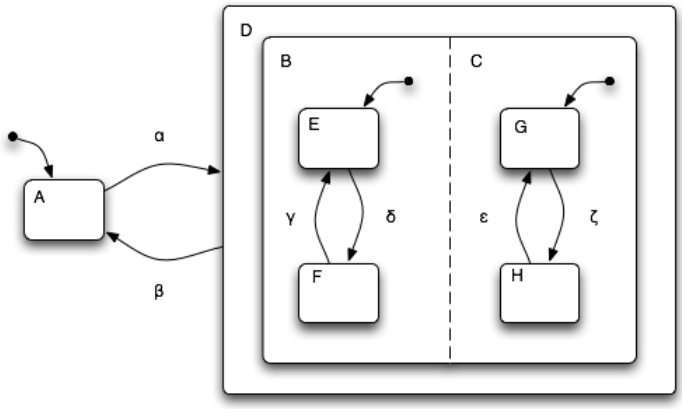

Figure 19: Superstate D is the orthogonal product of B and C.

Another important aspect of subsystems running in parallel is synchronisation. 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 synchronise before continuing continuing further execution.

Being a sequence control oriented formalism, SFC/Grafcet [9] implements parallelism somewhat differently compared to the illustrated example. In SFC, a transition can be split up in parallel paths. Consequently, several parallel paths can be joined with an AND-junction. This sequential approach suits its sequence control purposes very well, and it supports synchronisation in a natural way.

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, synchronisation is implicitly demanded of parallel branches. However, it is sometimes useful to have subsystems working independently of each other that never synchronise, as is the case for states B and C in Figure 19. Those two systems will run concurrently until they are preempted by transition β. Hence, no synchronisation 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 20. This use of unsynchronised subsystems is common in Statecharts, and a more flexible way of implementing orthogonality is thus desirable.

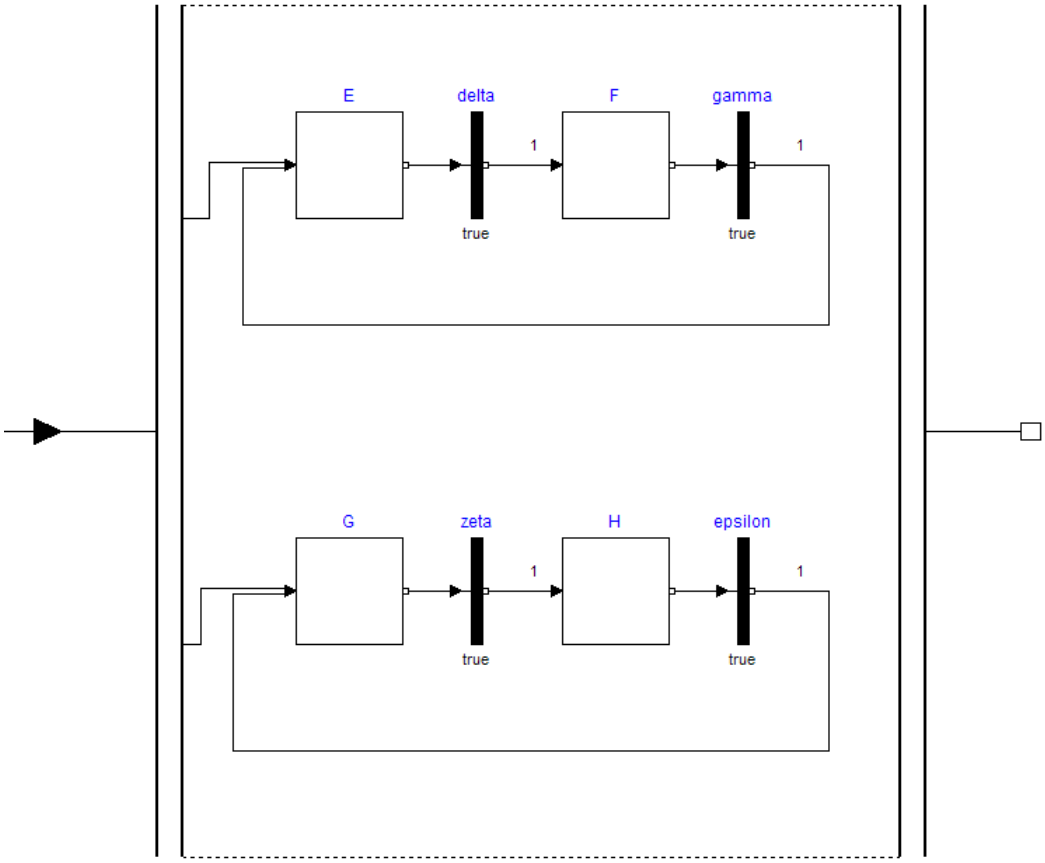

Figure 20: Unsynchronised parallel subsystems in StateGraph.

In ModeGraph, a more Statechart-oriented design is introduced without compromising existing possibilities of synchronisation. A Parallel component inherits from the ModeGraph.Parallel and is placed within a Composite to enable preemption. Figure 21 shows a ModeGraph implementation of Figure 19. As can be seen, ModeGraph incorporates an approach to orthogonality that is very similar to Statecharts. Note that one or more Parallel steps are placed in a Composite step to provide the possibility of preemption and synchronisation 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 its 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 synchronisation 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. If no exit is desired, `exit.exit.available` is set to true.

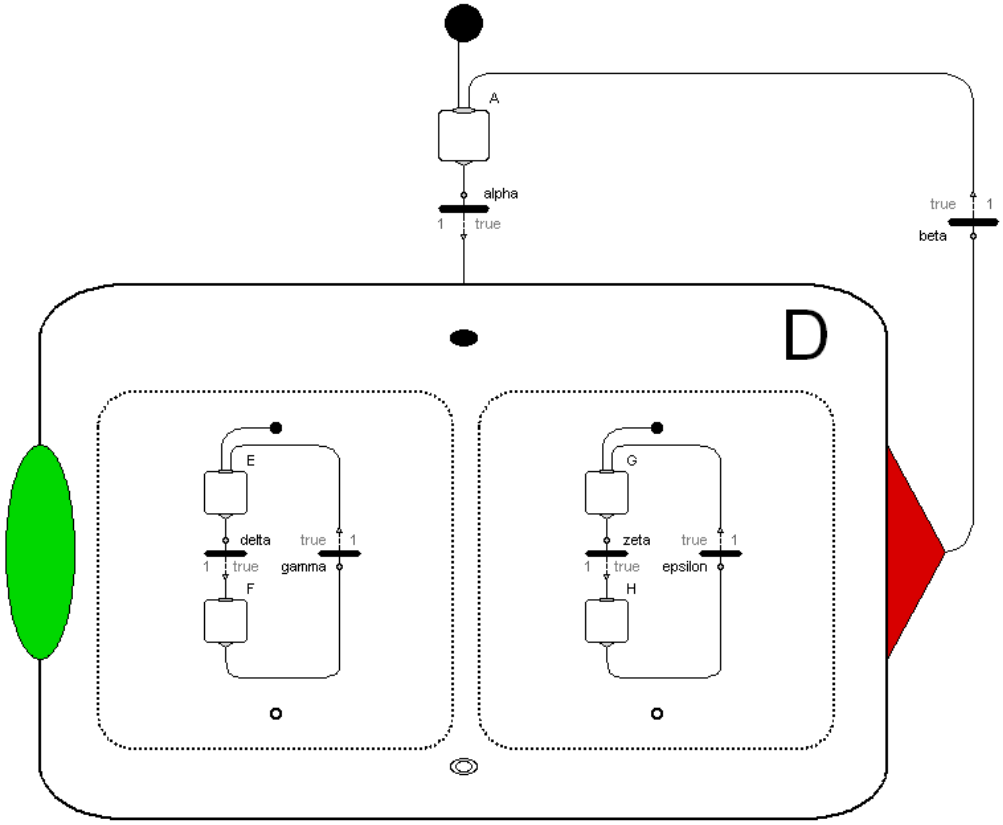

Figure 21: A ModeGraph Composite that contains two independent Parallel subsystems.

Assume the scenario in Figure 19 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 as shown in Figure 22.

To utilise exit connectors of the Parallel component, it is required to set the parameter `withExit` to true. If `withExit` is false, `finished` will always be true. This might seem strange, since the outport of Composite D in Figure 21 will work exactly like the suspend port. This becomes useful when synchronising Parallel

states with exits when there are additional Parallel states without exits present in the same Composite.

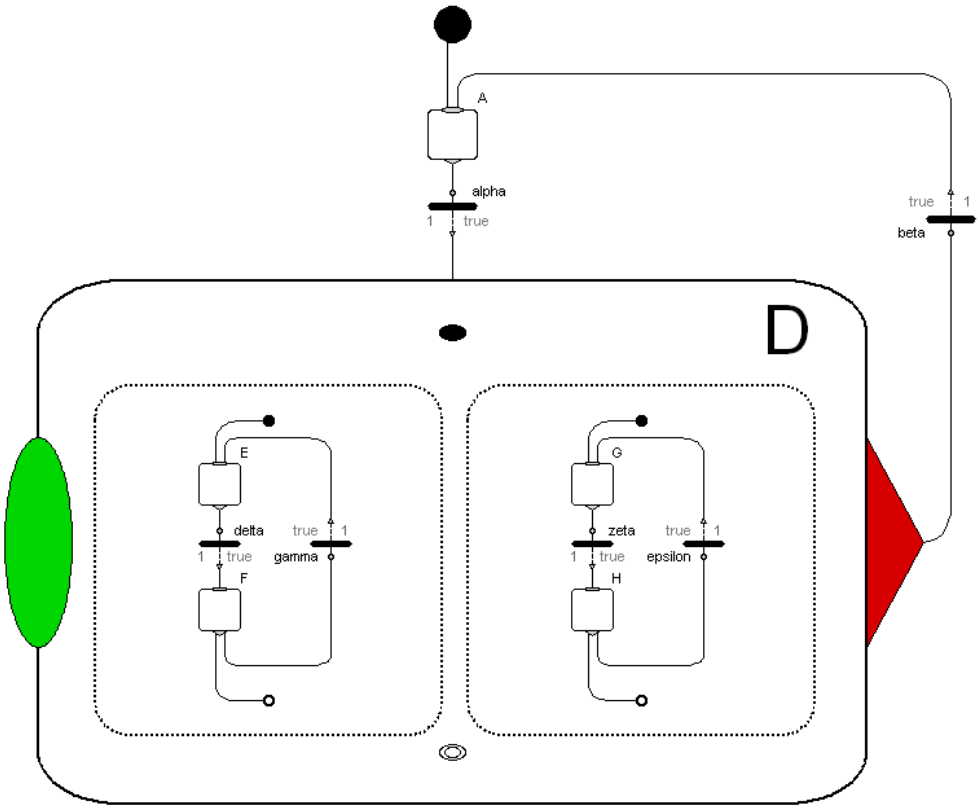

Figure 22: A ModeGraph Composite with two parallel subsystems that must synchronise to allow the Composite to exit.

The new approach of parallelism supports safe graphs in a natural way. As stated in [8], the Parallel and Alternative components in StateGraph are vulnerable to misuse. The problem is that the Alternative/Parallel component is instantiated at the same level as its branches. This makes it possible for a user to freely connect a branch outside the component without properly synchronising it. See Figure 23 for an example. Analysis to identify such cases forces unnecessary code overhead.

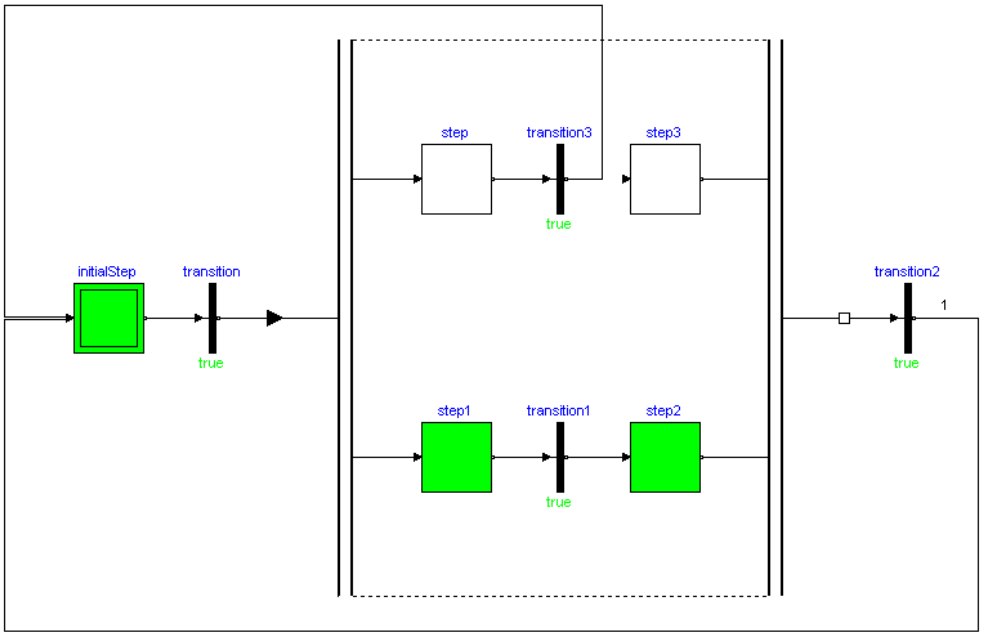

Figure 23: Unsafe usage of StateGraph.Parallel component.

In ModeGraph, this kind of misuse is not possible. Since the user is forced to inherit from ModeGraph.Parallel, there is no way of connecting to outer steps or transitions, since the icon layer is closed.

3.6 Modelica Mode

To implement Mode-Automata in Modelica, a mechanism for enabling/disabling a block is needed. As previously mentioned, there must be a way for a Composite to conditionally execute its associated code and enable/disable its children. The Modelica `Mode` comprises five variables that define the behaviour of the inheriting block. The variables define under what conditions that equations within the block will be evaluated and when to reset states and outputs. The class Mode is defined as:

```
partial block Mode
   input Boolean 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 mentioned Step component extends Mode, and only one of Steps A and B in Figure 4 can thus be enabled at a given time instant. This makes it possible to assign a variable y as

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

This code will be evaluated by the translator and generate the following single resulting 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.

Just like the Step, the Composite component inherits from the `Mode` base class. This produces no theoretical conflict since Steps and Composites occurring at the same level will have their conditional variables gathered. 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 behaviour 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 variable `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 Interfaces.Mode(
  enableSubBlocks = active,
  enable = true,
  resetStates = inport_fire,
  resetOutputs = true,
  finished = allSubBlocksFinished);
```

The built-in function `allSubBlocksFinished` checks 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. This leaves us with the Parallel Component. Since we think in terms of an orthogonal product A×B this might seem confusing since several Parallel components will indeed be simultaneously active. The remedy is that 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 behaviour 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 it should always be active.

This is the core functionality of the ModeGraph library. The possibility of simply ignoring equations within a disabled Mode, that also is 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.

3.7 Mode-Automata in Modelica

Decomposing behaviours into State Machines, it is natural to think in terms of independent modes. If a finite set of modes can be distinguished and guaranteed to remain mutually exclusive, this property can efficiently be exploited in the context of hybrid simulation. Being able to define an equation depending on the current mode presents a powerful semantics that supports the Modelica single assignment rule in a convenient way.

The concept of Mode-Automata is introduced in [6] and [7] and will be summarised below in a ModeGraph context. A mode-automaton is in [6] described with a 7-tuple: $(Q , q_0 , V_o , V_i , I , f , T)$ where:

- Q is the set of states.
- $q_0 \in Q_0$ is the initial state.
- V_o and V_i are the sets of output and input variables respectively.
- $I : V_o \rightarrow R \cup Z \cup B$ defines the initial values of every output variable.
- $f : Q \rightarrow V_o \rightarrow \mathsf{eqR}(V_o \cup V_i)$ is a total function that for each state assigns a right hand side (eqR) to every output variable.
- $T \subseteq Q \times C(V_o \cup V_i) \times Q$ is the set of transitions with C being the conditions on the set of input and output variables.

For a detailed definition of the mentioned sets, consult [6].

Implementing the Mode-Automata semantics with Modelica requires definition and mapping of the concepts onto the available properties of the language. This will, in combination with extended functionality compared to Mode-Automata, introduce additional semantic elements and concepts.

In ModeGraph a state is represented by a step that can be either active or inactive. Furthermore, a Composite step may also represent a state. A partial class, Mode, is introduced from which a Modelica component that should represent a Mode-Automata state should inherit. This way the translator will know what components it should handle as Mode-Automata states to be able to guarantee mutual exclusivity. Hence, every Step, s, and Composite, c, needs to extend Mode, and their state, active, is thus associated with a state $q \in Q$.

To avoid having a particular initial step component, another, safer, approach is taken in ModeGraph. Similar to Safe State Machines (SSM) [1], ModeGraph

marks a step as being the initial state by connecting it to an entry port, depicted with a black dot. A consequence of this is that all states are actually initially inactive. However, the initial step will become active in the first time instant. It is thus reasonable to define $I(q_0)$ = true and $I(q_n)$ = false, $\forall n > 0$.

S.S.M also defines a final state that when active allows a normal termination of the refined state (Composite in the ModeGraph case). In ModeGraph this is implemented similar to the initial step – the final state is connected to the exit port. Let $q_f \in Q$ be the final state of a mode-automaton.

By default, if a state q is active and none of its attached transitions (q,c,q'), $q \neq q'$ are taken, it remains active in the next time instant. This implicitly defines the (q,c,q'), q = q' transition, which is always present in ModeGraph.

An equation clause is associated with every state $q \in Q$. In this clause every variable, $x \in V_o$ can be assigned to an expression specific for the state corresponding to the Mode-Automata $f(q_n)$ function. Similar to restrictions on algorithm clauses, it is only allowed to make a variable assignment within an equation clause associated with a Modelica Mode, i.e., a legal assignment includes a variable reference on the left hand side and an expression on the right hand side. The translator collects and generates:

$$x \in V_o,\ e(x) = \textbf{if } q_0 \textbf{ then } f(q_0) \textbf{ else if } q_1 \textbf{ then } f(q_1) \ldots \textbf{ else if } q_n \textbf{ then } f(q_n)$$

where e(x) is the global expression of the variable x.

Hierarchy in Mode-Automata

As indicated above, the Composite step inherits from the Mode class and is thereby a mode. The Composite is, as the name indicates, the ModeGraph implementation of the refinement operator $\triangleright$ defined in [7]. Hence, a Composite comprises a number, n, of Mode-Automata according to:

$$M \triangleright \{M_j\}_{j\in[0,n]} = (Q \triangleright \{Q_j\}_{j\in[0,n]},\ q_0 \triangleright q_0^0,\ V_i',\ V_o',\ I',\ f',\ T')$$

In ModeGraph additions to the handling of the outputs and internal equations of modes are made. When a Composite is inactive, no internal interaction should be performed at all. This is achieved by simply turning off the outputs of the refined states, never evaluating the equations associated with the child modes but remembering the states. Hence, the refined modes are somehow linked to the parent mode. Since the number of refined modes are arbitrary, implementing this behaviour needs to be done with care. To avoid lengthy communication between parent and child modes, a mode has been given attributes that inform the translator of the properties of the Composite Mode. These attributes will be defined below.

The attribute M.enableSubBlocks controls whether or not the equations within sub block Modes of a Composite Mode M should be evaluated or not. This is to maximise performance by never evaluating inactive code. In the Composite M.enableSubBlocks is set to the active state, evaluating its children's associated equations only when it is active.

$$f'(x)(q_j \triangleright q_k^j) = \begin{cases} f(x)(q_j), & x \in V_{o,} \\ \textbf{if } M_j.enableSubBlocks \textbf{ then } f^j(x)(q_k^j) & \\ \textbf{else } pre(x), & x \in V^j_o,\ k \in [0,n_j], \\ pre(x), & \text{otherwise.} \end{cases}$$

M.enableSubBlocks applies also to assignment of ouputs and local variables, but in a slightly different fashion. M.resetOutputs decides the output of the child Modes of the current Mode. When M.enableSubBlocks is false and resetOutputs is set to false, outputs act as normal - but when the latter is instead true, all child mode outputs are set to false:

$$\bigcup^{n}_{j=0} V'^{j}_{o} = \text{if } M_j.\text{enableSubBlocks then } \bigcup^{n}_{j=0} V^{j}_{o}$$
$$\text{else if } M_j.\text{resetOutputs then false}$$
$$V_o' = V_o \cup \bigcup^{n}_{j=0} V'^{j}_{o}$$

This efficiently prevents internal interaction during inactivity of a Composite mode. Note how no explicit communication is needed between the different hierarchical levels of the modes.

Similarly, an attribute for resetting the internal states is needed. This is called M.resetStates and recursively resets states on all underlying hierarchical levels to their initial values. In [7], this is defined as being the only way to enter a refined state. But to enable the possibility of suspend/resume functionality of a Composite, the `resetStates` flag is required. Hence, the definition is here that the mode-automaton refining q_d is set to its initial state q_0^d if M_d.resetStates = true when q_d is entered, otherwise it is set to q_j^d.

$$q' \in Q = \text{if } M_j.\text{resetStates then } I(q) \text{ else } q;$$

Parallelism in Mode-Automata

ModeGraph handles parallelism very similar to Mode-Automata. Each parallel subsystem is instantiated to prevent abuse and unsafe graphs. The Parallel components are not modes themselves, but contain an arbitrary number of modes. Definition 3 in [7] gives the complete definition of parallel sub-systems in Mode-Automata. The parallel composition is denoted $M_1 \times M_2$ and is briefly summarised as:

- The set Q = Q1 × Q2 is naturally the orthogonal product of the two sets of states.
- (q_0^1, q_0^2) are the initial states.
- All inputs that are not also an output are inputs also of the parallel composition: $(V_i^1 \cup V_i^2) \setminus (V_o^1 \cup V_o^2)$.
- Outputs are the union of the output sets: $(V_o^1 \cup V_o^2)$.
- Initial values for every output variable x are set depending on which parallel subsystem it belongs to.
- The function assignment of variables in (q^1, q^2) are dependent on the set of outputs, V_o^1 or V_o^2, that x belongs to.

- The set of transitions comprises both the parallel components' set of transitions, but also the case where the condition C is the conjunction of C^1 and C^2.

For synchronisation, each Parallel component needs to signal to its parent when it is ready to synchronise. This is done by setting a variable, finished. If a Composite contains one or more Parallel components, it can check if all Parallel components are ready to synchronise by using the built in function AllSubBlocksFinished, which returns true if all parallel sub-systems have finished = true.

4. Application Examples

Two application examples will be presented below to provide motivation for the ModeGraph part of the thesis.

An example of a production line will first be presented to prove feasibility of the library. It will be shown how modularisation can achieve flexible and effective abstractions using the ModeGraph formalism.

Harel's classic wristwatch [5] will be used to prove that the ModeGraph Library is capable of implementing complex state machines defined with the Statecharts formalism.

4.1 Production Line

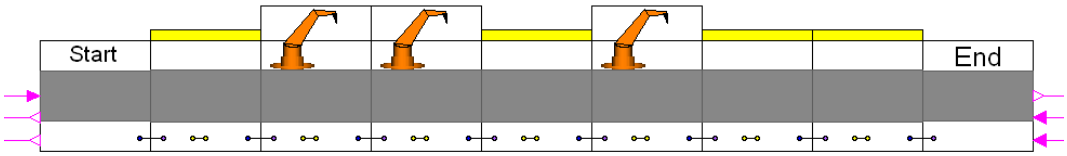

Figure 24: A production line with three robots.

Introduction

This example is a Modelica implementation of an application example first presented in the paper 'Cooperating Distributed Control Objects' [4] published in 1991 by H. Elmqvist. One of the emphases of this paper is the possibility of a high modularisation of a given system. In larger control systems it is essential that decomposition into smaller logical units can be performed in an efficient manner. A straightforward way to perform this decomposition is to treat each physical unit as a decomposed entity.

In this example, a production line handling an arbitrary product is constructed. The product is thought of as a car being manufactured. The production line is comprised of conveyor tables and working units (e.g. industrial robots), see Figure 24. To enable an arbitrary set-up of the production line, with respect to a specific set of tasks to be performed on the car, a set of interconnectable units is defined. Each of these modules will contain its own ModeGraph logic and be synchronised with the adjacent modules with bidirectional Boolean data flows.

Structure of Production Line

Because of the need for high flexibility due to the dynamic nature of a production line in a plant, it is important that the modules are as independent as possible. It is, e.g., natural to separate a table module from a robot module, since there may exist different types of table modules as well as different robot modules. As the communication interface of a table needs to be consistent regardless of whether a

robot is present or not, the need for a terminator/dummy-robot is clear. A table connected to a terminator will make the table function solely as a transportation unit. The graphical representation of the modules should make it intuitive and straightforward to compose a full production line without any profound knowledge of the embedded logic. It should be easy and obvious how to interconnect the different modules.

Communication

Communication and synchronisation between the modules are realised with Modelica connectors. The connectors are visualised as small circles on the module, see Figure 25.

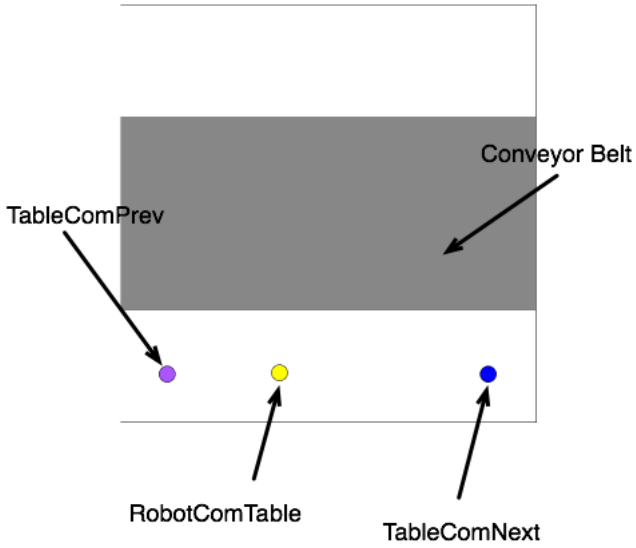

Figure 25: The Table component with three connectors.

As defined in [4], the connectors between the tables send one message to its successor table and two messages to its predecessor table. This means that one connector for the predecessor (`TableComPrev`) and one for the successor (`TableComNext`) is needed. The signal flow found in [4] is depicted in Figure 26 below. A signal, `Ready`, tells the successor that the work is done and that the table is ready to commence out-transport. When a table is Empty and waiting for in-transport, it should inform its predecessor with an `Idle` signal. To make it possible for the predecessor to know when to start or to stop out-transport by activating or deactivating its motor, a `Transport` signal is sent during in-transport.

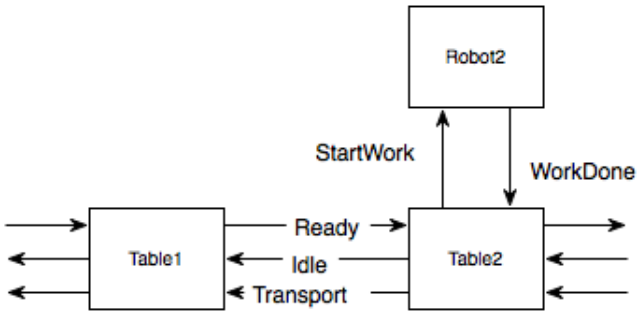

Figure 26: Signal flow between tables and working units.

The Modelica implementation of `TableComNext` uses a connector, defined as:

```
connector TableComNext "Connects with the successor table"
  output Modelica.Blocks.Interfaces.BooleanOutput Ready
    "Sends signal to successor that table is ready to
      commence out-transport of the car";
  input Modelica.Blocks.Interfaces.BooleanInput Idle
    "Receives signal from successor that the table is Idle";
  input Modelica.Blocks.Interfaces.BooleanInput Transport
    "Receives signal from successor when out-transport
    should be performed";
  ...
end TableComNext;
```

The connector `TableComPrev` is defined in a similar fashion, but with the input/output ports switched:

```
connector TableComPrev "Connects with the predecessor table"
  input Modelica.Blocks.Interfaces.BooleanInput Ready
    "Receives signal from predecessor that table is ready to
    commence out-transport of the car";
  output Modelica.Blocks.Interfaces.BooleanOutput Idle
    "Sends signal to predecessor that table is Idle";
  output Modelica.Blocks.Interfaces.BooleanOutput Transport
    "Sends signal to predecessor that out-transport should
    be performed";
  ...
end TableComPrev;
```

As shown in Figure 26, a robot communicates with a table by receiving a `StartWork` signal, which the table sends when a car is in position. When the robot has finished its work, it sends a signal, `WorkDone`, to inform the table it is done working. More details of signal flow can be found in [4]. The robot connectors `RobotComTable` and `RobotComRobot` are defined as:

```
connector RobotComTable "Connects table to robot"
  output Modelica.Blocks.Interfaces.BooleanOutput StartWork
    "Sends signal when car is in place and robot can start
    working";
  input Modelica.Blocks.Interfaces.BooleanInput WorkDone
    "Receives signal when robot has finished working";
  ...
```

```
end RobotComTable;

connector RobotComRobot "Connects robot to table"
  input Modelica.Blocks.Interfaces.BooleanInput StartWork
    "Receives signal when car is in place and work can be
    commenced";
  output Modelica.Blocks.Interfaces.BooleanOutput WorkDone
    "Sends signal when work is done";
  ...
end RobotComRobot;
```

A UML-diagram showing the structure of the available modules can be viewed in Figure 27 below.

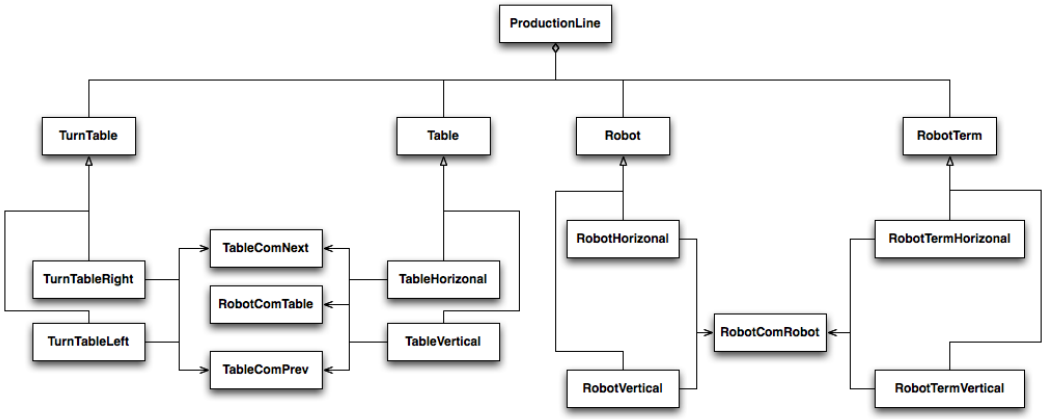

Figure 27: Structure of modules of the production line.

State Machine Implementation

The idea is to have each table autonomously communicate with its predecessor and successor using boolean messages. A table is initially `Empty` and waiting for a `Ready` signal from its predecessor. When this happens, it commences `InTransport`. On completion (when a limit switch has been reached) the table notifies the robot that a car is in place with a signal, `StartWork`, and then waits for a `WorkDone` signal from the robot. When this occurs, the table waits for the successor to send a signal, `Transport`, which means it is ready to begin in-transport. When out-transport is finished and the car is in place on the successor table, it stops sending the `Transport` signal. The state machine logic is illustrated in Figure 28.

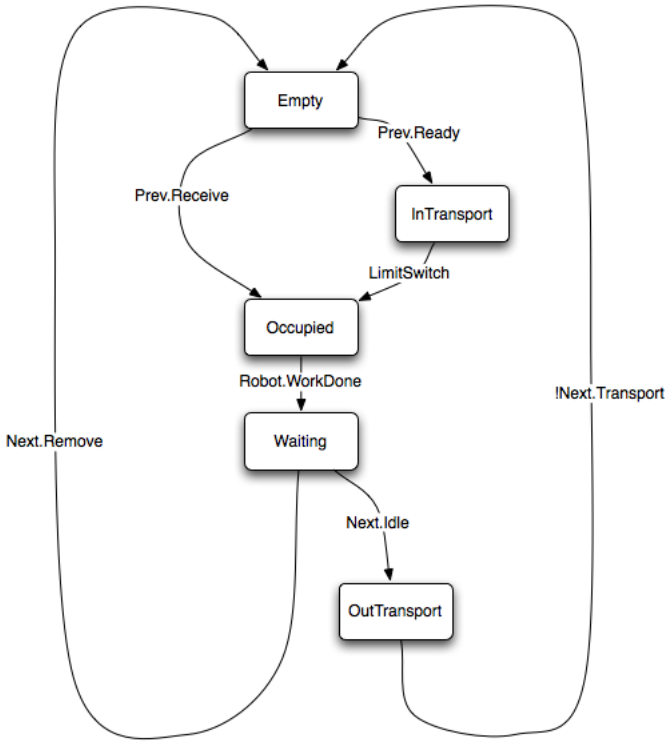

Figure 28: State machine representing the behaviour of a table.

The transition `Next.Remove` is used to handle the case when a car is removed from a table by being lifted from above. This functionality has been chosen not to be included, hence, the transition will have a false condition. As a reminder of how the table module is visually represented , see Figure 29.

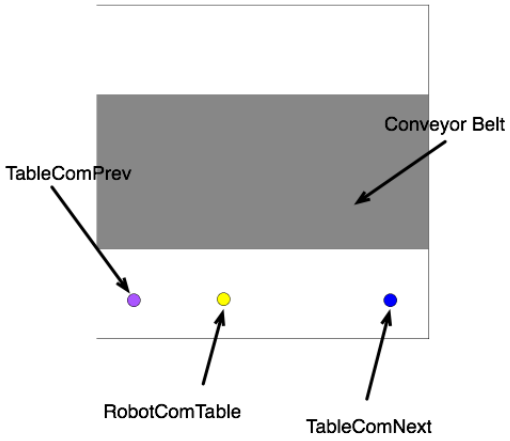

Figure 29: Visual representation of a table.

Control

To make the tables behave properly with respect to their adjacent neighbours, it is required to define control rules regarding how to react to input signals and when

to send output signals. The following equations define the control behaviour of the graph and can also be found in [4].

```
Prev.Idle = Empty.active
Prev.Transport = InTransport.active
Next.Ready = Waiting.active
Motor = InTransport.active OR OutTransport.active
Robot.StartWork = Occupied.active
```

The Modelica component `Modelica.Blocks.Sources.BooleanExpression` is used to realise the control rules and are connected to the transitions of the state machine, see Figure 30. Notice how the connectors are utilised in the diagram layer for control of the ModeGraph.

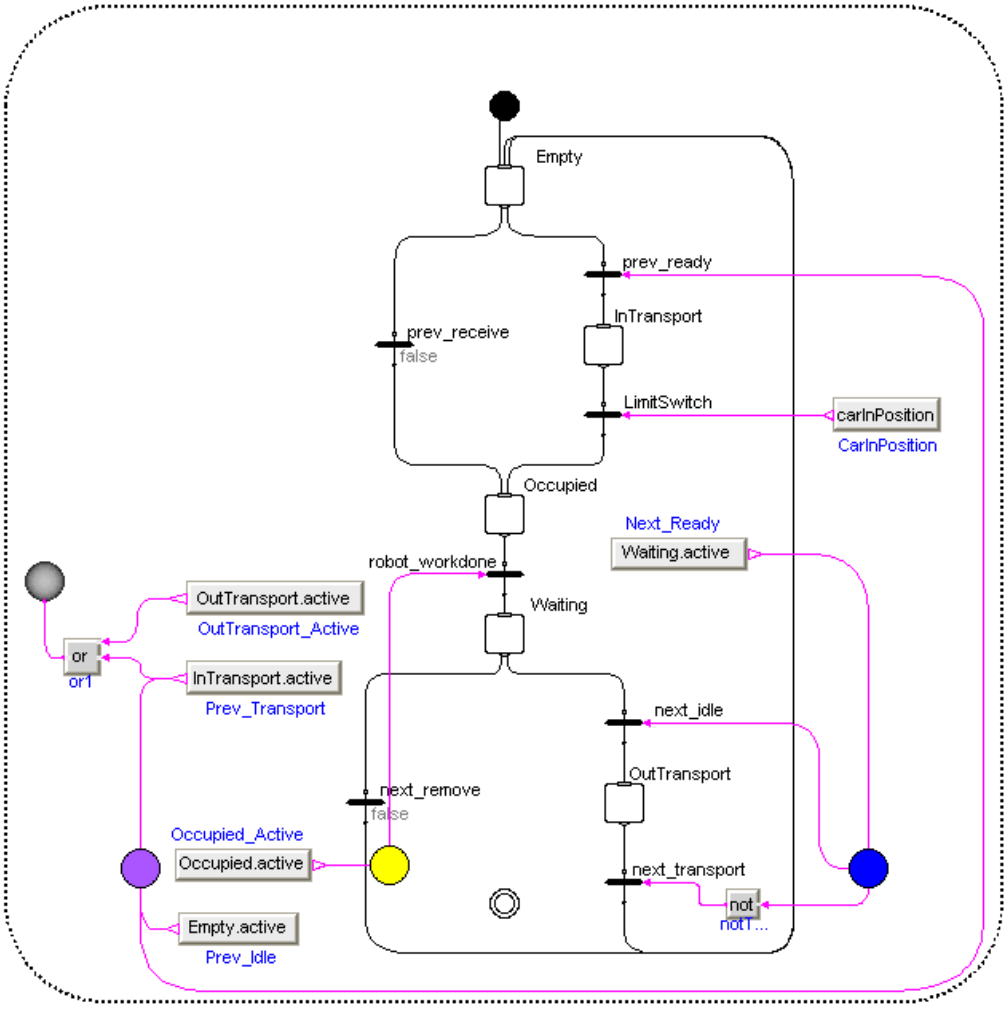

Figure 30: ModeGraph realisation of the Table component.

Animation

Each table is responsible for animating the car once it leaves its Empty state. This means it will paint the car even though it has not yet arrived at the surface of the table. The reason for this is to synchronise animation between the tables. The table keeps animating the car throughout the whole cycle, until it once again

enters its `Empty` state. The animation is carried out with the help of the `DynamicSelect` function. `DynamicSelect` lets you change parameters dynamically (e.g. depending on time) of another function. Due to the nature of simulation, a velocity needs to be set and utilised during the transfer and the limit switch is simply treated as true when the car is in the middle of the table. Figure 31 displays a car being animated by two tables while it is being transferred between them.

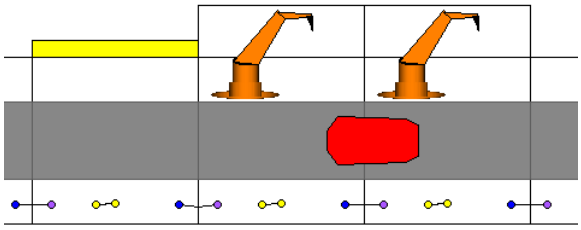

Figure 31: A car being transferred between two tables with connected working units.

The robot module is required to animate when it has received `StartWork` to visualise that it is currently carrying out work on the car. As soon as it sends a `WorkDone` signal, it returns to its normal icon state. The animation of the robot performing its task can be viewed in Figure 32.

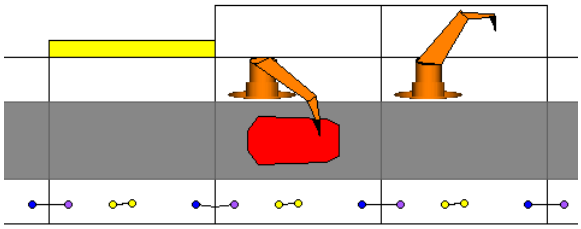

Figure 32: A table is halted while the connected robot is working on a car.

4.2 Harel's Wristwatch

In 1987 D. Harel introduced a set of extensions [5] to the existing state machine formalism. His goal was to introduce better support for refinement of states, hierarchy, and handling of memory.

Harel identified and mapped the wristwatch functionality onto a Statechart in [5] to show feasibility of the semantics of Statecharts. The wristwatch has been accepted as a challenging benchmark for reactive systems, and is therefore chosen in this thesis to be used to show the capabilities of the ModeGraph library. The complete chart of the watch will be described in more detaile below along with its corresponding ModeGraph realisation. The complete wristwatch statechart is shown in Figure 60-61 in the Appendix.

To illustrate his extensions, he used a Citizen Quartz Multi-Alarm III wristwatch as a case study. The watch display comprises six different displays

showing time, 12/24h time setting, AM/PM setting, alarm on/off, chime on/off and stop watch indication. It is operated by four buttons: A, B, C and D. The watch can display time and date (day of month, weekday, date, month, year). It has a chime function, display back-light for improved illumination, low-battery warning, a stop watch with lap/reg mode options and two alarms.

Button A works as a switch between display modes of the watch. The different display modes are: Time (default), alarm1, alarm2, and stopwatch. In each of these, button C will enter update mode where the current time or the alarm time can be set. Inside the update modes, C will flip through different time entities and finally return from update mode. Button B is used to exit update regardless of which internal state being active. In stopwatch mode, button B is used as on/off switch. Additionally, the illumination is turned on every time B is pressed down, and shut off on release. Button D serves as an on/off switch of the current active display mode. If in time mode, pressing button D will result in display of the current date. In stopwatch mode, button D works as a selector between reg/lap display mode. For more detailed descriptions of the behaviour of the wristwatch, consult [5].

Parallel components of the wristwatch

The user interface of the ModeGraph implementation of the wristwatch is shown in Figure 33.

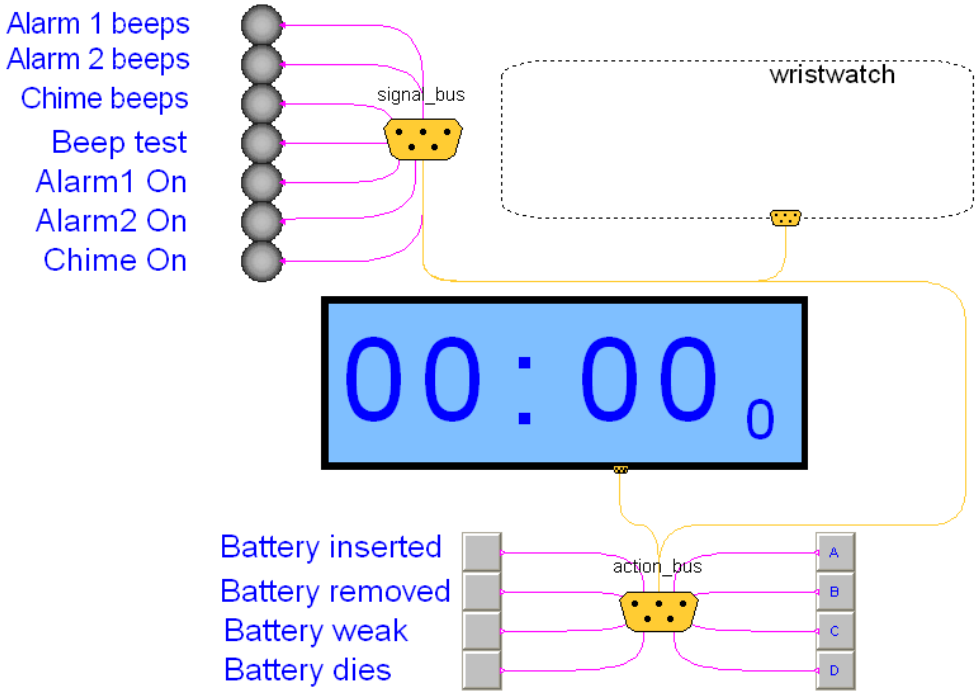

Figure 33: Overview of the wristwatch user interface.

The wristwatch can be decomposed into four distinct parts:

- Visual display window
- Time logic
- Input/Output
- Behavioural logic

The first three are considered to be more or less outside the scope of this thesis, while the latter constitutes the interesting part with respect to the ModeGraph library.

The visual display is implemented in Modelica as a separate component that communicates with the reactive system to obtain information of what should be displayed.

Input stimuli is required to enable a user to manipulate the wristwatch in various ways. As previously mentioned, input is given by four buttons, A, B, C, and D. In the Modelica model of the watch, these are realised with a component, `PressButton`, similar to the `TrigButton` component from the UserInteraction library. The difference between the two is that a `PressButton` signals true as long as the button remains pressed, whereas a `TrigButton` just send a single true pulse when the button is pressed. In other words, the output of a `TrigButton` can be seen as the edged output of a `PressButton`. Hence, to only trigger as the button is pressed down (for normal operations), all signals are edged when used as conditions. Of course there are exceptions, for instance when a button should be held down for a period of time. The code of the `PressButton` component looks like:

```
model PressButton
  ...
  Modelica.Blocks.Interfaces.BooleanOutput buttonState;
  annotation (Icon( ...,
    fillColor=DynamicSelect({192,192,192}, if
      buttonState > 0.5 then {255, 0,0} else {192,192,192}),
            ...,
          interaction={OnMouseDownSetBoolean(buttonState,true),
          OnMouseUpSetBoolean(buttonState,false)}));
end PressButton;
```

To show on/off status of the alarms and to indicate when they are sounding, `IndicatorLamp` components from the `UserInteraction` library are used. In this context time logic means the periodic increments of the current time, but also covers setup of the current time. Also this part is implemented as a separate Modelica Component, `Time`, that also handles the alarm times. Since the actual time logic is outside the scope of [5], the code will be presented with only minor explanations.

```
Model Time
  ...
algorithm
  /* Time logic */
  if not update_active.status then
    time_second :=mod(init_time_second - timeStamp + time, 60);
    time_minute :=mod(div(init_time_minute*60 + init_time_second
            + time - timeStamp, 60), 60);
```

```
    time_hour :=mod(div(init_time_hour*3600 + init_time_minute*60
            + init_time_second + time - timeStamp, 3600), 24);
    time_day :=mod(init_time_day +div(init_time_hour*3600
            + init_time_minute*60 + init_time_second + time
            - timeStamp, 24*3600), 7);
    time_year :=init_time_year + div(time
            - timeStamp,365*24*3600);
    time_date := init_time_date + div(init_time_hour*3600
            + init_time_minute*60 + init_time_second + time
            - timeStamp, 24*3600);
    time_month := init_time_month;

    while time_date > days_in_month[time_month+1] loop
      time_date := time_date - days_in_month[time_month+1];
      time_month := time_month + 1;
    end while;
    time_mode :=init_time_mode;
  end if;

  //increment time setting depending on what time state is active
  when update_increment then
    init_time_second :=mod(init_time_second+inc_time_second, 60);
    init_time_minute :=mod(init_time_minute+inc_time_minute, 60);
    init_time_hour :=mod(init_time_hour + inc_time_hour, 24);
    init_time_month :=mod(init_time_month, 12) + inc_time_month;
    init_time_date :=mod(init_time_date,
      days_in_month[init_time_month]) + inc_time_date;
    init_time_day :=mod(init_time_day, 7) + inc_time_day;
    init_time_year :=init_time_year + inc_time_year;
    init_time_mode :=mod(init_time_mode + inc_time_mode, 2);
  end when;

  when alarm1_increment.status then
    alarm1_time := alarm1_time + alarm1_inc_multiplier;
  end when;
  when alarm2_increment.status then
    alarm2_time := alarm2_time + alarm2_inc_multiplier;
  end when;

equation
  update_increment = update_inc.status;
  when {not update_active.status, main_active.status} then
    //take a timestamp when the clock is switched on
        timeStamp =  time;
  end when;
  ...
end Time;
```

The `time_<quantity>` variables represent the current time and are sent to the 'Display' component for visualisation. The `init_time_<quantity>` variables are used to describe the initial time setting of the watch. When a user increments the time with respect to the currently active quantity state, the corresponding values are sent to the `time` component and are added to the initial variables, as shown above.

Communication between components located at different hierarchical levels of the system is performed exclusively with expandable connectors, also called *Buses*. The code of such a bus looks like:

```
expandable connector Bus
  extends Modelica.Icons.SignalBus;
end Bus;
```

The expandable prefix enables a variable number of connectors to be connected together into a single bus. This efficiently decreases the number of connections that stretches throughout the hierarchical composition.

Overview of Parallel entities of the wristwatch

The watch needs to handle several things simultaneously. It should display the time, but simultaneously compare the current time to the set alarm time and sound the alarm if they are equal. Furthermore, it should be possible to illuminate the watch display when in any display mode and additionally, chime functionality should also be working independently of the current active display mode. This motivates the structural parallel decomposition shown in Figure 34, which is also given in more detail in Figure 60 and in Figure 61 in the Appendix.

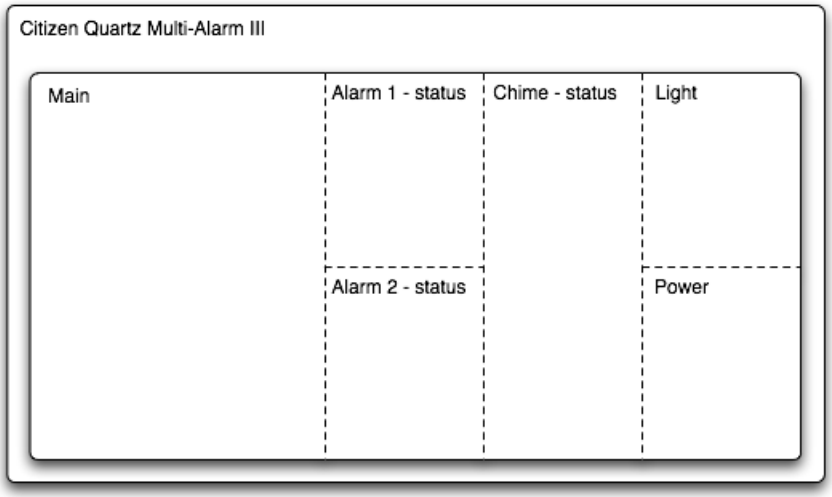

Figure 34: Overview of the parallel subsystems of the wristwatch.

Parts of the parallel states communicate with each other to a certain extent, but they never synchronise. The ModeGraph implementation of the parallel construct of the wristwatch comprises a Composite encapsulating six Parallels, as shown in Figure 35.

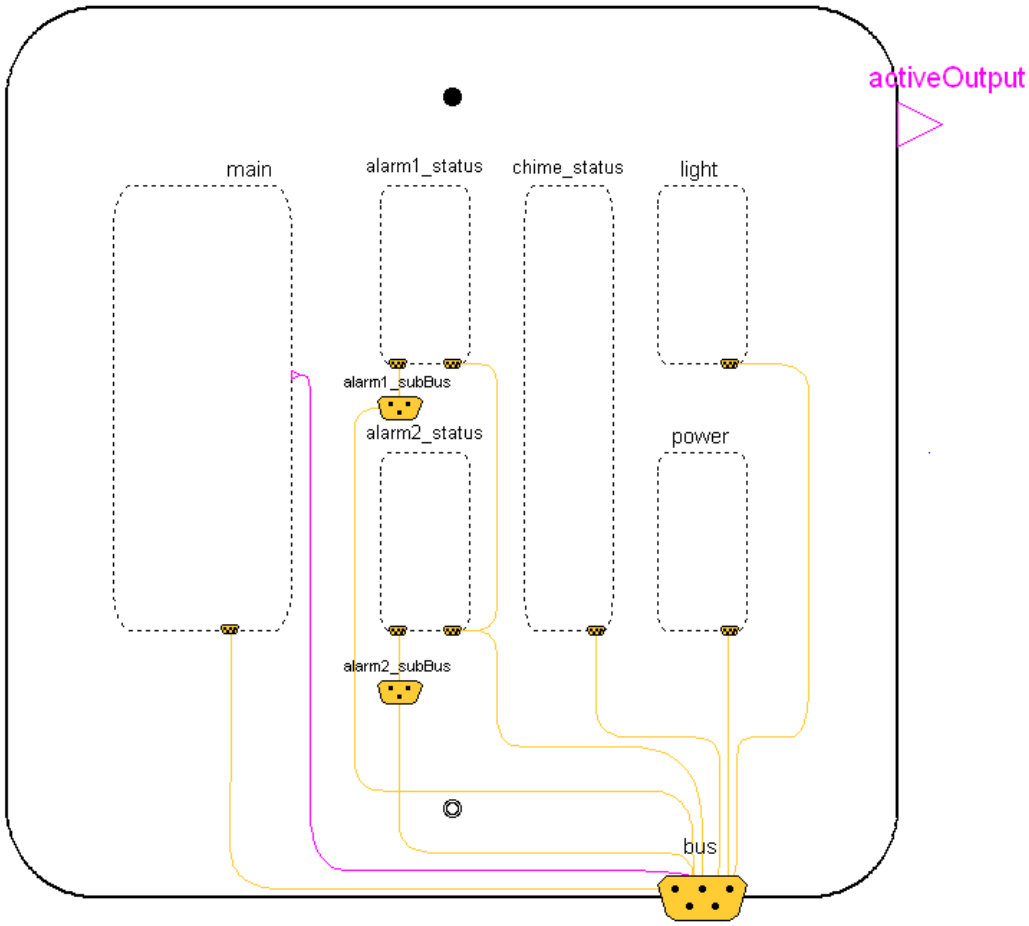

Figure 35: ModeGraph realisation of the parallel subsystem decomposition, called CQMA.

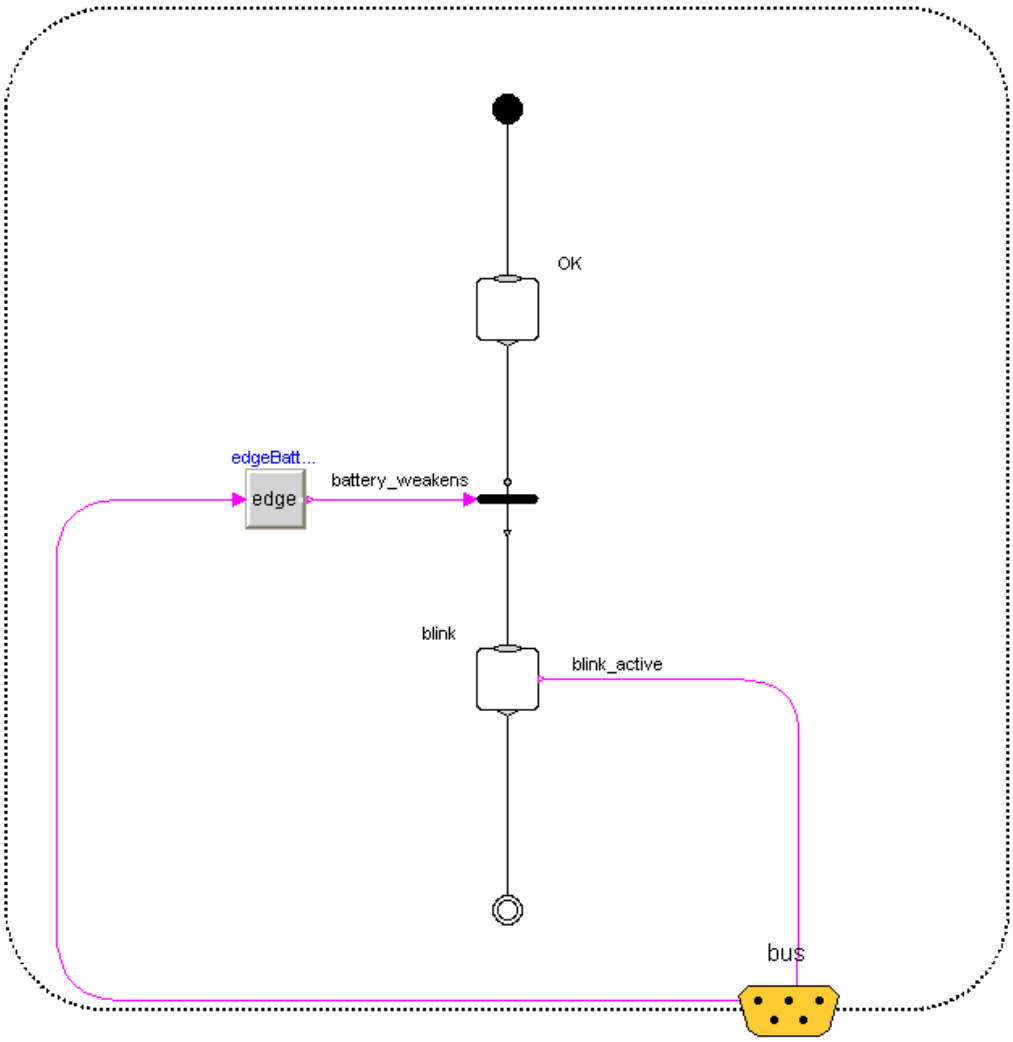

Figure 36: ModeGraph realisation of Parallel component Power.

The Parallel component `power` describes whether the watch battery is OK or if it is weak and risk to die. When it eventually dies, the entire `CQMA` state is left, and all functionality is shut off. Notice that the battery cannot die if it has not first weakened, in contrast to the case where it is explicitly removed from the watch. In the latter case the `CQMA` state is left regardless of the internal states. The ModeGraph realisation of Power is shown in Figure 36.

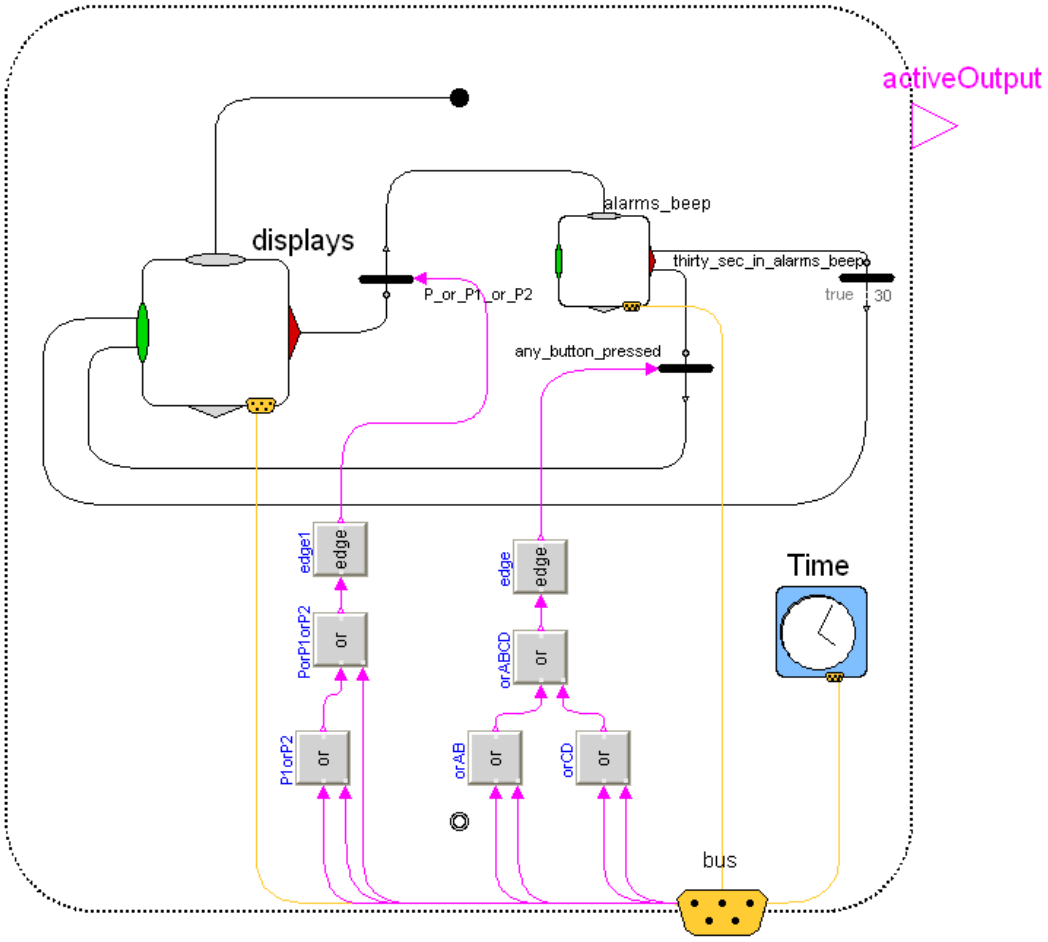

Figure 37: ModeGraph realisation of Main.

All the main time-, date-, alarm-, and stopwatch-functionalities are gathered in the state Main. The state Main is decomposed into a Displays state and a Alarms-beep state, see Figure 60 in the Appendix. In the former, normal display of time, alarm and stopwatch are described, and the latter is activated once the current time matches one of the alarm time settings. Furthermore, the `time` component described above is placed in the state `main`, see Figure 37.

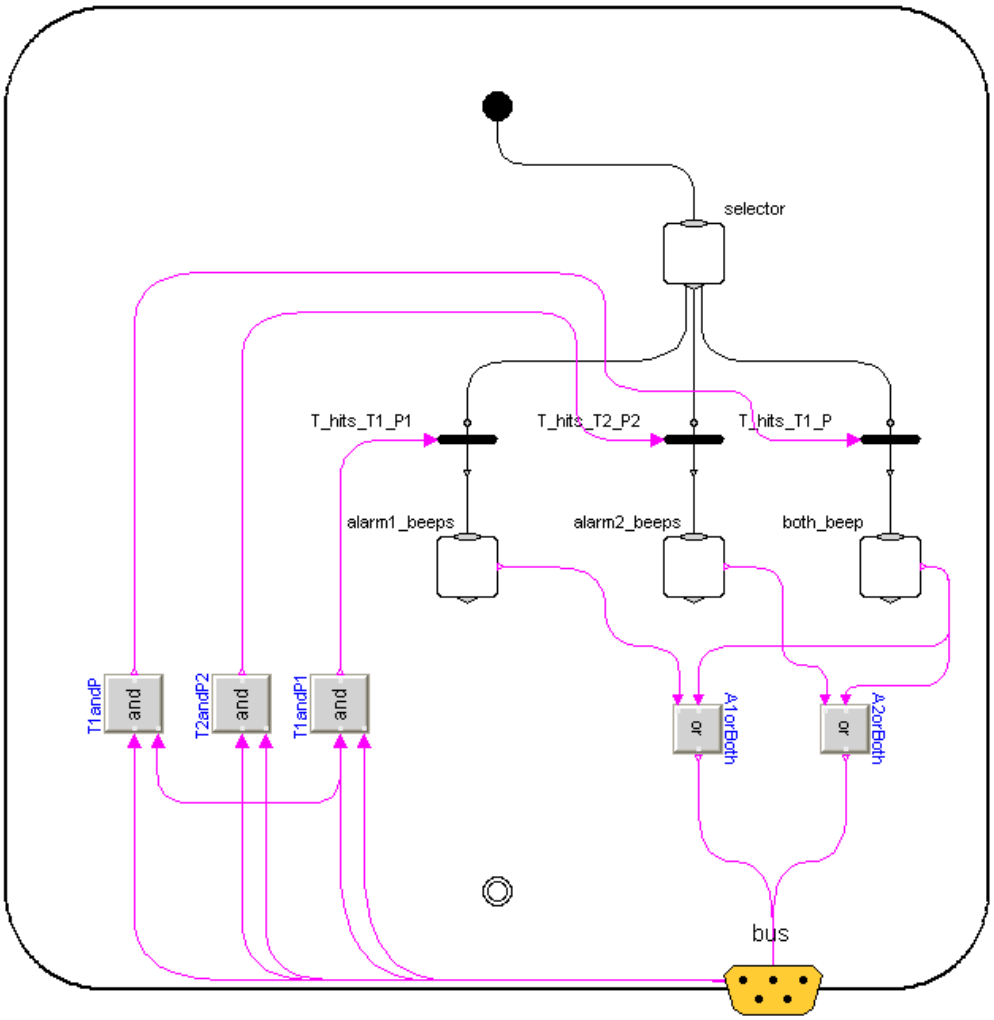

Figure 38: ModeGraph realisation of Alarms-Beep.

As soon as an alarm time matches the current time, the Displays state is preempted, and the alarm X beeps state in Alarms-beep corresponding to the triggered Alarm X is entered. The ModeGraph implementation of Alarms-beep is shown in Figure 38. The state Displays will be elaborated in the following section, see Figure 43.

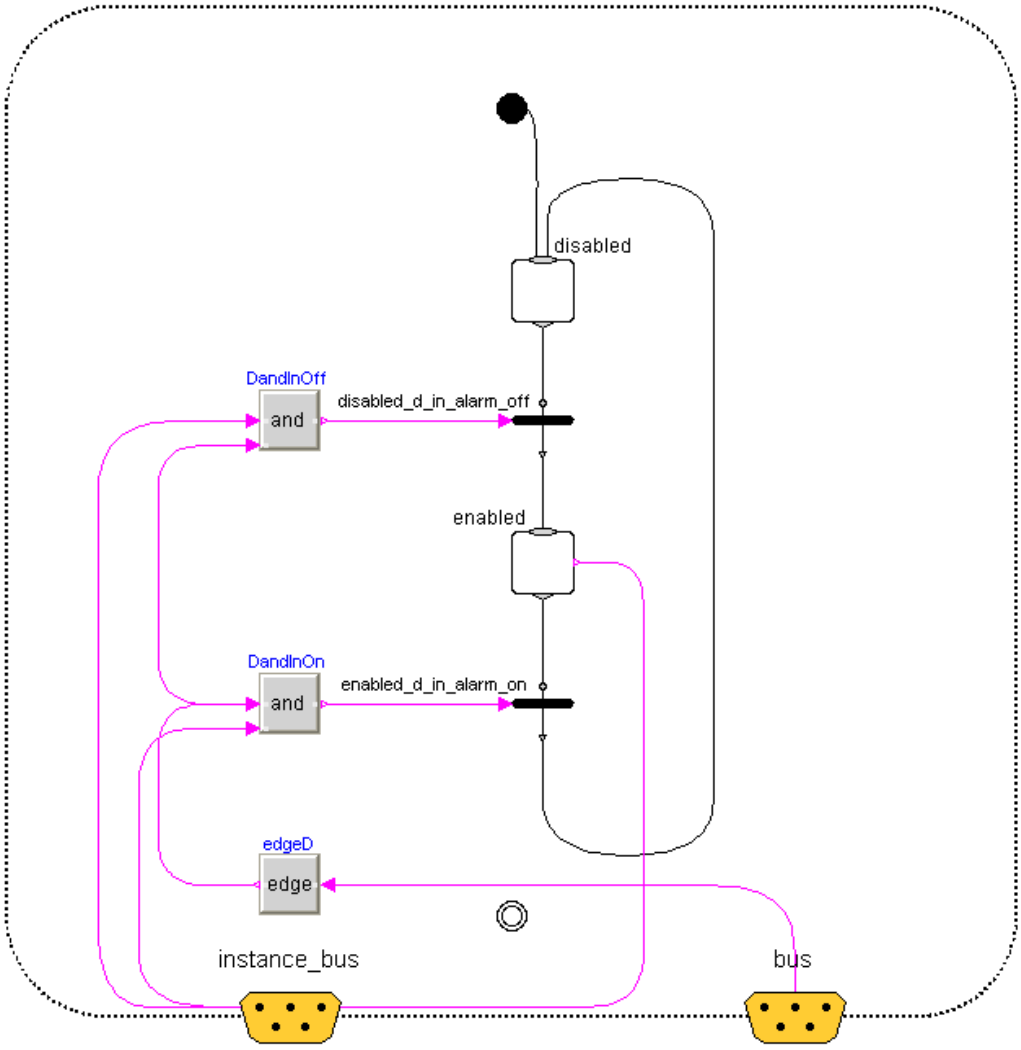

Figure 39: Alarm X - Status realised in ModeGraph.

There are two parallels describing the current status of the two alarms of the watch. These are identical except for the transition conditions. Both alarms are enabled/disabled by pressing button D, but there is an additional transition condition. To be allowed to switch enable states, the corresponding alarm on/off states inside Displays needs to be active, see Figure 61 in the Appendix. This will be explained in more detail later. It is clear that there is a need to prevent both alarms from being similarly enabled/disabled every time button D is pressed.

When realising two subsystems that are this similar, it is natural to create one ModeGraph component and instantiate it twice. The ModeGraph Alarm_Status component can be viewed in Figure 39.

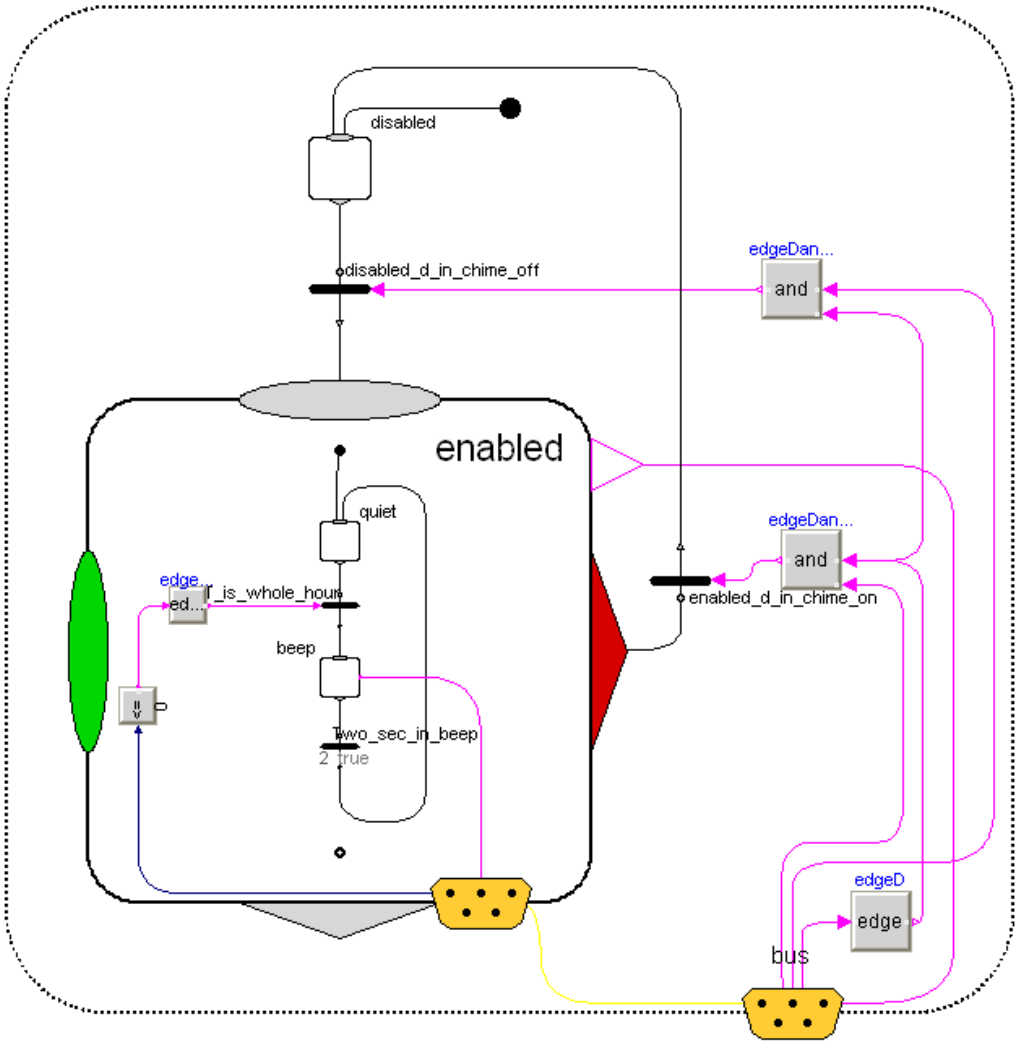

Figure 40: ModeGraph realisation of Chime – Status.

Since this component will be instantiated twice, it is necessary to separate the instance-specific communication from each other. For instance, Alarm1 – Status should only change states depending on the current status of Alarm1 and not Alarm2. This can easily be solved by introducing a sub-bus. All instance-specific communication is connected to another bus, `instance_bus`. This bus is connected to a sub-bus, `AlarmX_subBus`, which in turn is connected to the main bus, see Figure 35. In the other end, an identically named sub-bus can be connected to successfully collect the instance-specific signals.

Similar to the Alarm X - Status parallel sub-states, there is a Chime parallel sub-state. Chime works like an alarm that, if enabled, chimes every whole hour. The Enabled state is further decomposed into one Quiet state and one Beep state. The latter is active for two seconds every whole hour. The ModeGraph realisation is showed in Figure 40.

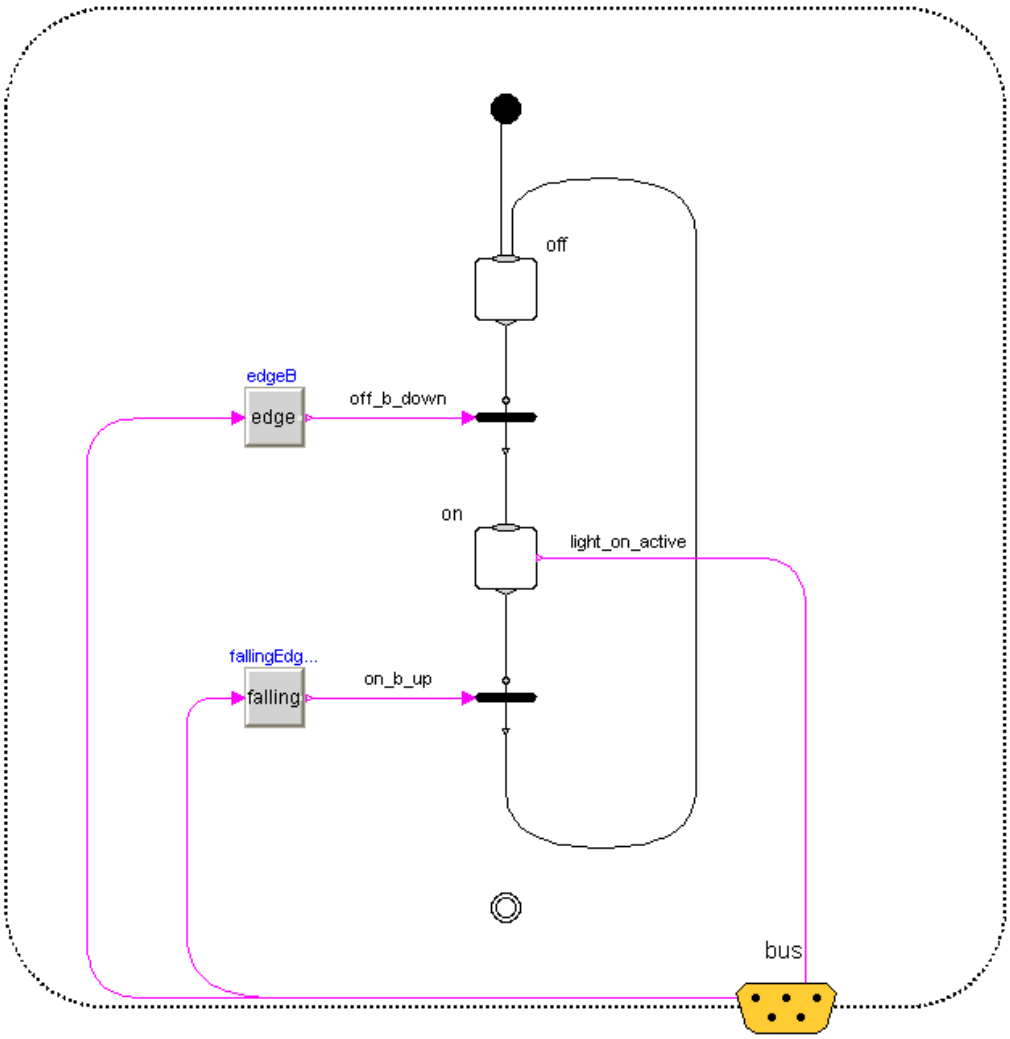

Figure 41: Subsystem Light realised in ModeGraph.

The ModeGraph realisation of Light is shown in Figure 41. The Parallel `light` changes state to `on` every time button B is pressed down, and returns to `off` as soon as the button is released. This is performed independently of all other functionality triggered by the button B, and the display will thus be illuminated also when other functionality triggered by this button is used.

Display modes

The main functionality of the watch is described in the state Displays. It is in this state that update and display modes of the current time and date are handled. Furthermore, alarms can be set and the stopwatch can be run in different display modes. Harel's statechart version of Displays is shown in Figure 42.

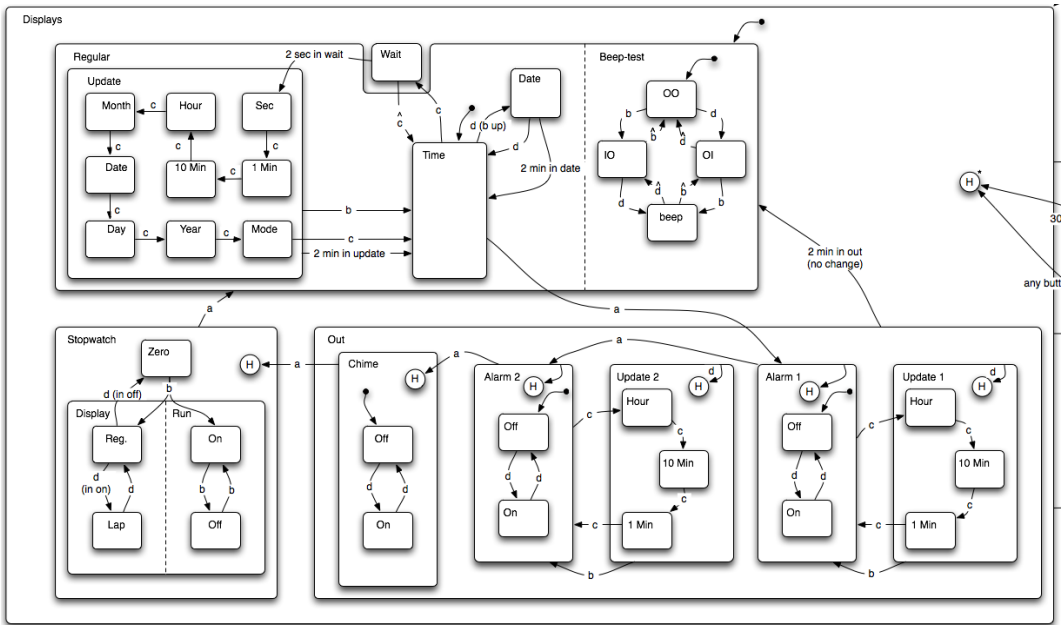

Figure 42: The Displays state described in statecharts by D. Harel.

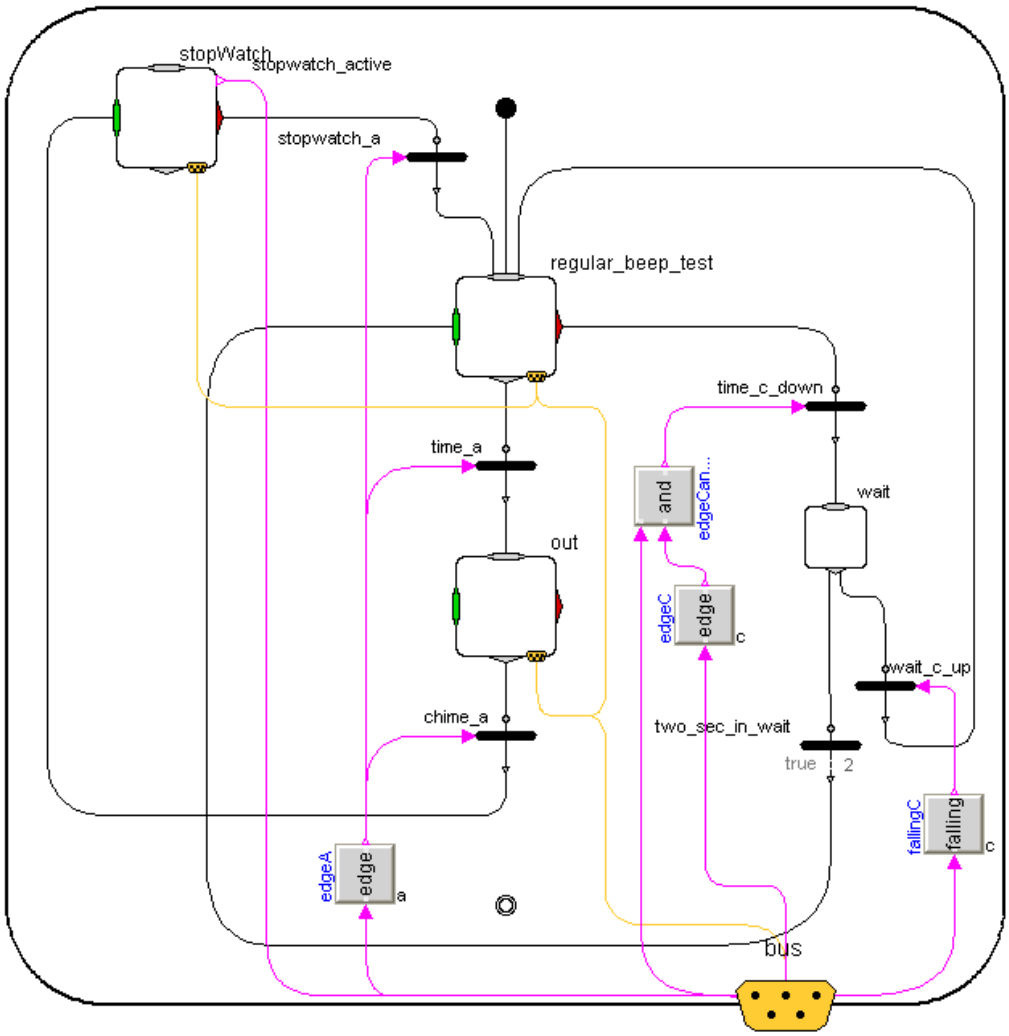

Figure 43: ModeGraph realisation of state Displays.

The most complex state, regular, runs in parallel with the state Beep-test, as can be seen in Figure 60. The functionality of the latter is somewhat self-explanatory. Pressing button B and button D simultaneously will sound the alarm to check that it is functioning properly. The ModeGraph realisation of Displays is shown in Figure 43.

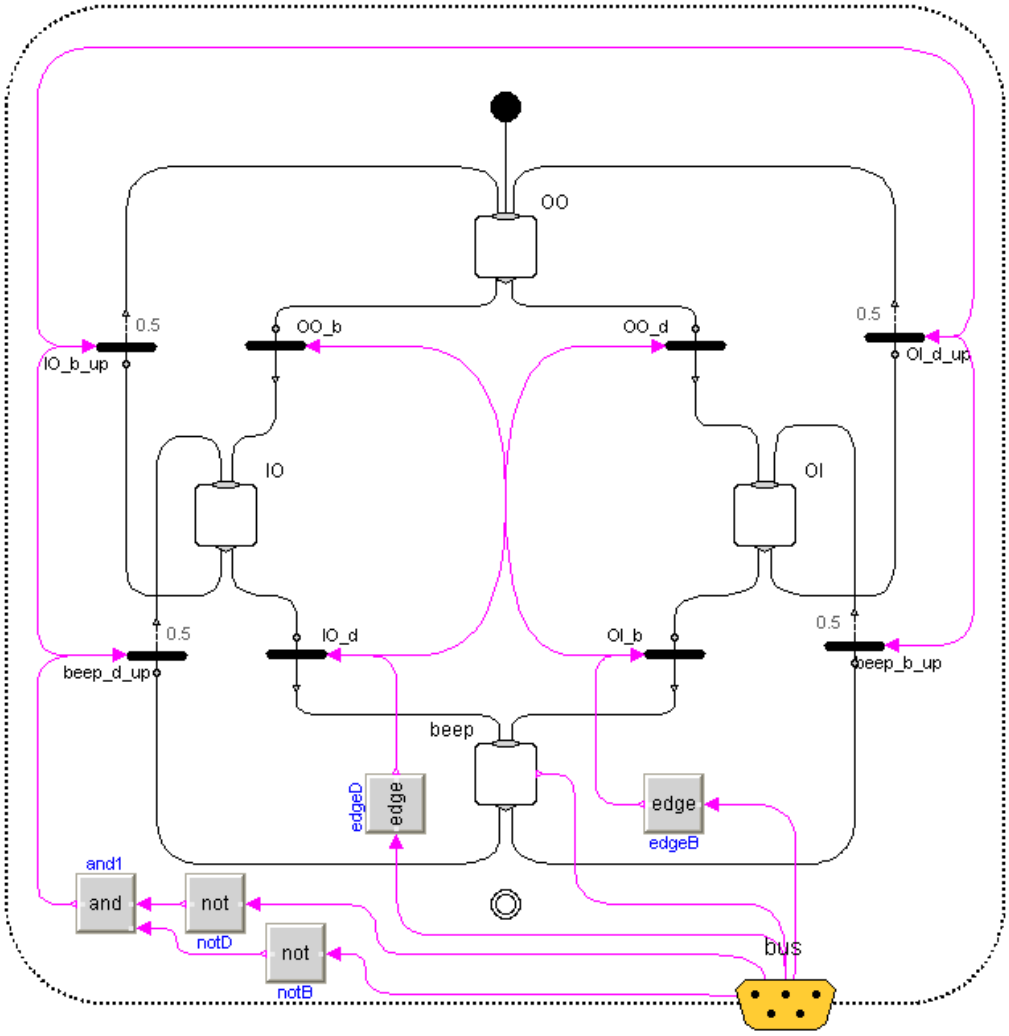

Figure 44: ModeGraph realisation of Beep-test.

In the ModeGraph implementation of Beep-test, it is, for obvious reasons, not possible to press two buttons at the same time. Instead the logic will sound the alarm if the buttons are pressed in consecutive order with a delay of no more than half a second, see Figure 44. Setting the time of the alarms, and turning them on/off, as well as chime functionality is handled in the Out sub-state. The stopwatch sub-state contains control- and display-logic for the stopwatch.

Regular

Inside the Regular state, either Time or Date can be chosen for visualisation on the display. Additionally, the time can be set by entering the Update state by holding down the button C for more than two seconds. During this time, the state Wait, localised outside of Regular, is active. Figure 45 shows the ModeGraph implementation of Regular. Note that pressing button C in state `Time` will enter the `update` component. However, examining Figure 43 it is clear that `regular` is suspended when button C is pressed. If C is held down for two seconds, `regular` will be resumed, and `update` becomes active. If C is released within two seconds, `regular` is entered though the inport, which resets all underlying states, thus entering `Time`.

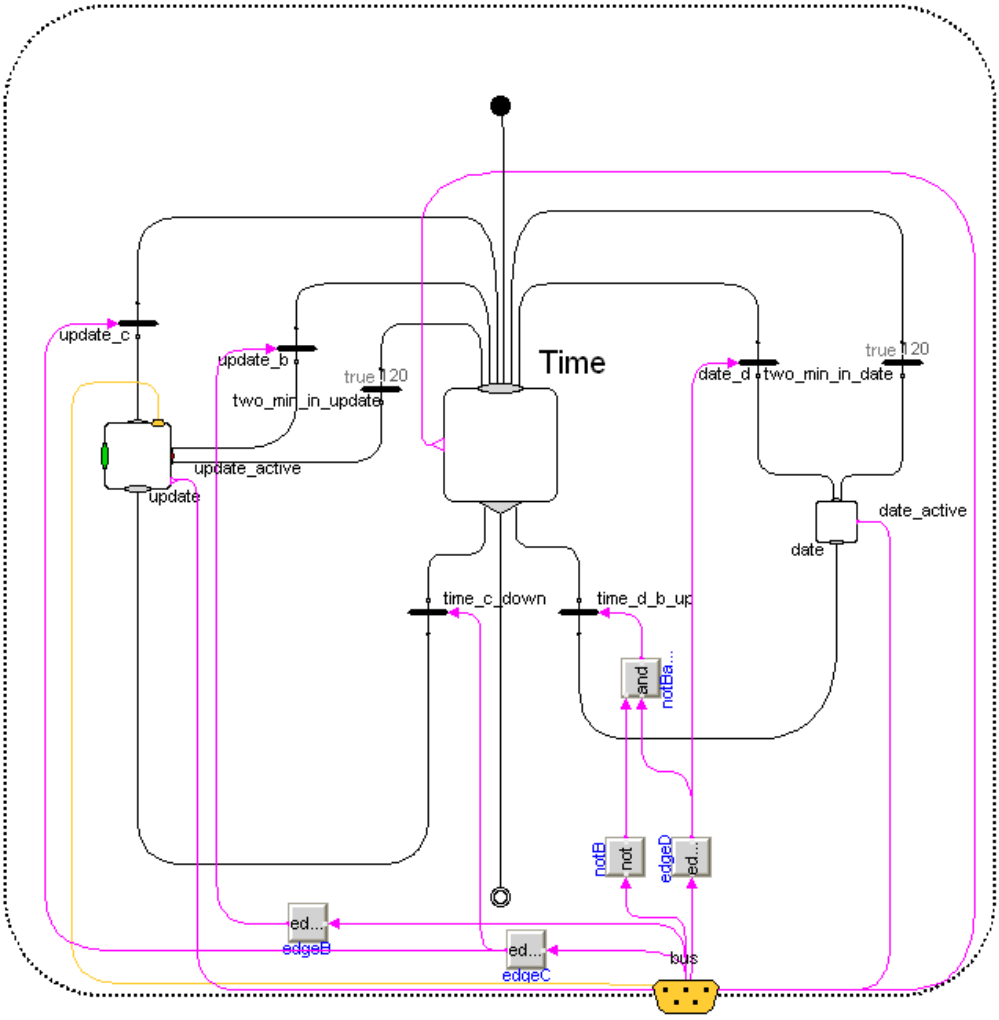

Figure 45: ModeGraph realisation of Regular.

Once inside Update, the user may flip through different time quantities by pressing the button C. Pressing button D increments the current time with the value associated with the active quantity state.

In the ModeGraph implementation of Update, a connection from button D (extracted from the bus, as always) is filtered through an `Edge` component and

then connected back to a new variable, `update_increment`, in the bus, see Figure 46. The `RealOutputs` and `IntegerOutputs` at the bottom of the component are used to connect the local increment variables, that are soon to be described, to the bus.

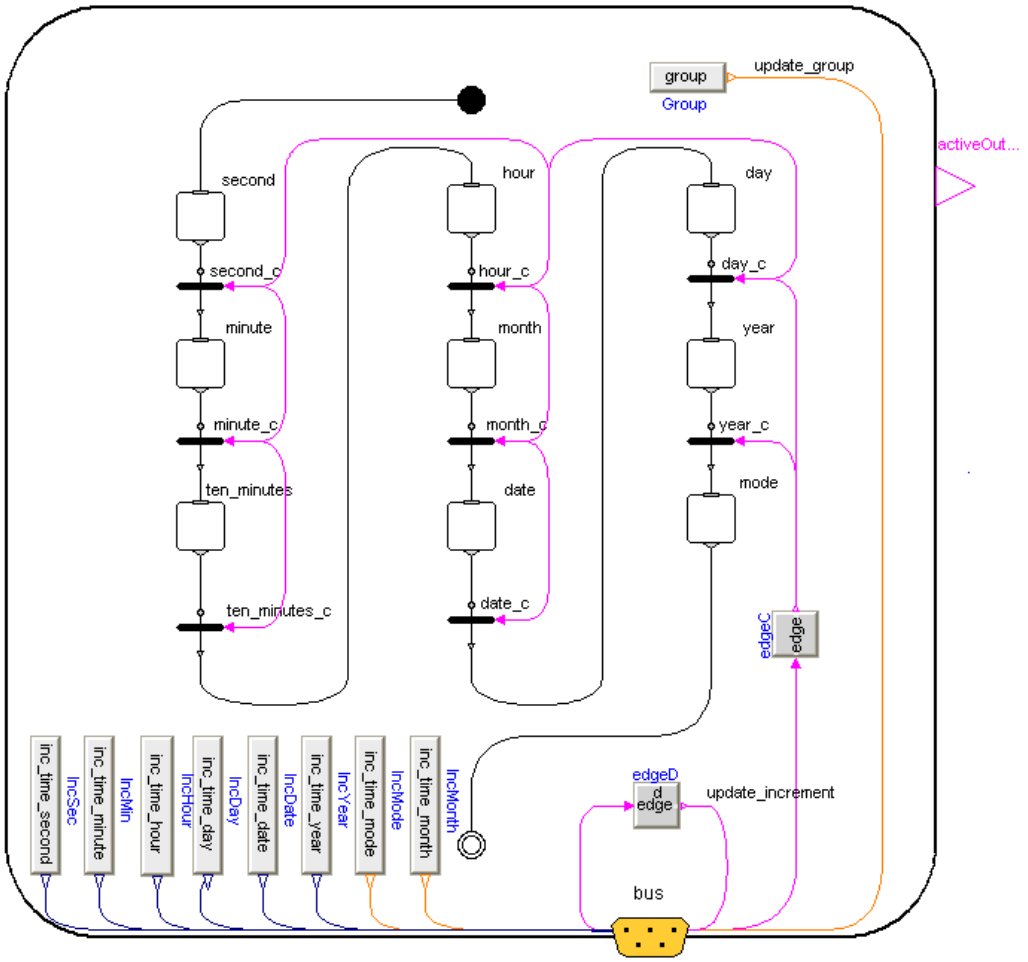

Figure 46: ModeGraph realisation of Update.

The last active quantity state will then once again be active and the time logic will receive the `update_increment` signal. Recall the when-clause triggered by this `update_increment`, in the code of the `Time` component. The conditional if statements in `update` are given as:

```
if second.active then
  inc_time_second = 1;
else
  inc_time_second = 0;
end if;

if minute.active then
  inc_time_minute = 1;
elseif ten_minutes.active then
  inc_time_minute = 10;
else
```

```
  inc_time_minute = 0;
end if;

if hour.active then
  inc_time_hour = 1;
else
  inc_time_hour = 0;
end if;

...

if mode.active then
  inc_time_mode = 1;
else
  inc_time_mode = 0;
end if;
```

Furthermore, a boolean variable, `group`, is used to indicate what should be shown on the display when in the different quantity states in `update`. The encoding of the different display modes of the `group` variable is:

- 1 – Hours, Minutes, and Seconds
- 2 – Month, Date, Day, Mode
- 3 – Year
- 0 – Error

And the conditional structure looks like:

```
//Choose display mode depending on active Mode
if second.active then
  group = 1;
elseif minute.active then
  group = 1;
elseif ten_minutes.active then
  group = 1;
elseif hour.active then
  group = 1;
elseif month.active then
  group = 2;
elseif date.active then
  group = 2;
elseif day.active then
  group = 2;
elseif mode.active then
  group = 2;
elseif year.active then
  group = 3;
else
  group = 0;
end if;
```

Depending on the active Mode in `displays`, the visual display window shows different time/date/alarm information. This is a suitable situation to once

again utilise the guaranteed mutual exclusivity between Modes. The six fields of the display window are named `digit1`, `digit2`, `digit3`, `digit4`, `small_digit` and `text_digit`, and represent the six fields in the display window as shown in Figure 47.

Figure 47: The display window of the watch and the corresponding variable names.

The boolean variables in the conditions of the if-structure are sent over the bus from the respective states. The conditional if-structure of the display window is given as:

```
if time_active then
/* Display Time */
  digit1 = if time_mode < 0.5 then
    div(mod(time_hour, 12), 10) else div(time_hour,10);
  digit2 = if time_mode < 0.5 then
    mod(mod(time_hour, 12), 10) else mod(time_hour,10);
  digit3 = div(time_minute, 10);
  digit4 = mod(time_minute, 10);
  small_digit = if time_mode < 0.5 then 12 else 24;
  if time_mode < 0.5 then
    if div(time_hour,12) < 0.5 then // 0 for AM
      text_digit = 8;
    else // 1 for PM
      text_digit = 9;
    end if;
  else
    text_digit = 10;
  end if;
elseif date_active then
/* Display Date */
  digit1 = div(time_date, 10);
  digit2 = mod(time_date, 10);
  digit3 = div(time_month, 10);
  digit4 = mod(time_month, 10);
  small_digit = -1;
  text_digit = integer(init_time_day);
elseif update_group == 1 then
/* Display update values of hour, minute and second */
  digit1 = div(init_time_hour, 10);
  digit2 = mod(init_time_hour, 10);
  digit3 = div(init_time_minute, 10);
  digit4 = mod(init_time_minute, 10);
  small_digit = init_time_second;
  text_digit = integer(init_time_day);
```

```
elseif update_group == 2 then
/* Display update values of month, date and day */
  digit1 = div(init_time_month, 10);
  digit2 = mod(init_time_month, 10);
  digit3 = div(init_time_date, 10);
  digit4 = mod(init_time_date, 10);
  small_digit = if time_mode < 0.5 then 12 else 24;
  text_digit = integer(init_time_day);
elseif update_group == 3 then
/* Display update value of year */
  digit1 = rem(div(init_time_year,1000), 10);
  digit2 = rem(div(init_time_year,100), 10);
  digit3 = rem(div(init_time_year,10), 10);
  digit4 = rem(div(init_time_year,1), 10);
  small_digit = if time_mode < 0.5 then 12 else 24;
  text_digit = integer(init_time_day);
elseif out_alarm1_active or out_update1_active then
/* Display Alarm1 Time */
  digit1 = if time_mode < 0.5 then
    div(mod(div(alarm1_time,60), 12), 10) else
    div(div(alarm1_time,60),10);
  digit2 = if time_mode < 0.5 then
    mod(mod(div(alarm1_time,60), 12), 10) else
    mod(div(alarm1_time,60),10);
  digit3 = div(mod(alarm1_time,60), 10);
  digit4 = mod(mod(alarm1_time,60), 10);
  small_digit = if time_mode < 0.5 then 12 else 24;
  text_digit = if time_mode < 0.5 and
    div(div(alarm1_time,60),12) < 0.5 then 8 else
  if time_mode < 0.5 and div(div(alarm1_time,60), 12) > 0.5
    then 9 else 10;
elseif out_alarm2_active or out_update2_active then
/* Display Alarm2 Time */
  digit1 = if time_mode < 0.5 then
    div(mod(div(alarm2_time,60), 12), 10) else
    div(div(alarm2_time,60),10);
  digit2 = if time_mode < 0.5 then
    mod(mod(div(alarm2_time,60), 12), 10) else
    mod(div(alarm2_time,60),10);
  digit3 = div(mod(alarm2_time,60), 10);
  digit4 = mod(mod(alarm2_time,60), 10);
  small_digit = if time_mode < 0.5 then 12 else 24;
  text_digit = if time_mode < 0.5 and
    div(div(alarm2_time,60), 12) < 0.5
      then 8 else if time_mode < 0.5 and
      div(div(alarm2_time,60), 12) > 0.5 then 9 else 10;
elseif stopwatch_active then
/* Display Stopwatch */
  digit2 = mod(sw_disp1, 10);
  digit1 = div(sw_disp1, 10);
  digit3 = div(sw_disp2, 10);
  digit4 = mod(sw_disp2, 10);
  small_digit = 0;
  text_digit = 11;

else
/* Otherwise show zeros (for example in wait and chime) */
  digit1 = 0;
```

```
    digit2 = 0;
    digit3 = 0;
    digit4 = 0;
    small_digit = 0;
    text_digit = if chime_active then 10 else 100;
  end if;
```

Recall the conditional definition of the boolean variable `group` in `update` (sent through the bus as `update_group`), and note how it is used in the structure above.

Out

The `Out` state comprises, for each alarm, a state Alarm X and a state Update X, where X is the number of the alarm, see Figure 48.

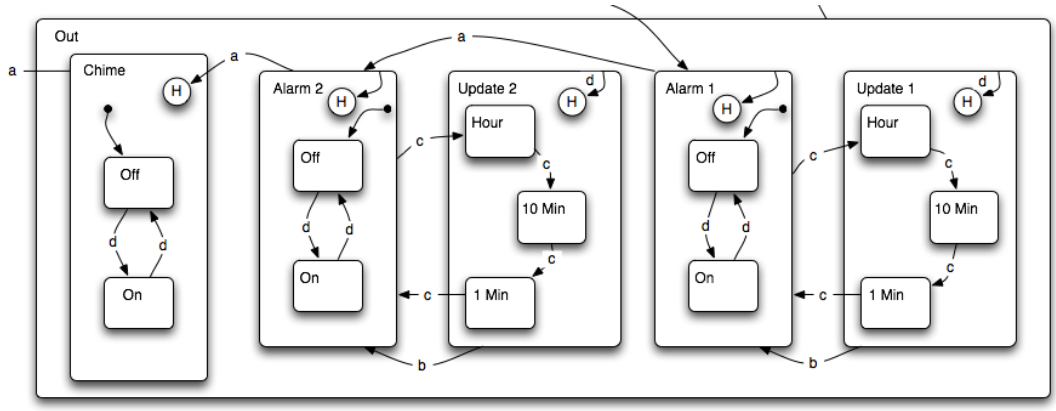

Figure 48: The superstate Out containing alarms and chime functionality from Harel's statechart wristwatch.

In Alarm X the alarm is turned on/off by pressing button D. Recall the Alarm X – Status parallel state, and its condition to be in Alarm X to allow an alarm status to be toggled. Additionally, if button C is pressed, an update of the alarm time is entered. Finally, a state Chime can be entered to toggle chime functionality on and off. The ModeGraph realisation of this update mechanism as well as the Chime state is very similar to the time update, hence no further details will be given about the realisation of them.

Stopwatch

Entering the Stopwatch state for the first time, a substate, Zero is entered. If button B is pressed, the parallel states Run and Display are entered, and the stopwatch starts running. Pressing button B will toggle the stopwatch on and off. However, when in state On, the user may press button D to display lap time. If button D is pressed when Run is in state Off, the stopwatch returns to state Zero, and the time is reset, see Figure 49.

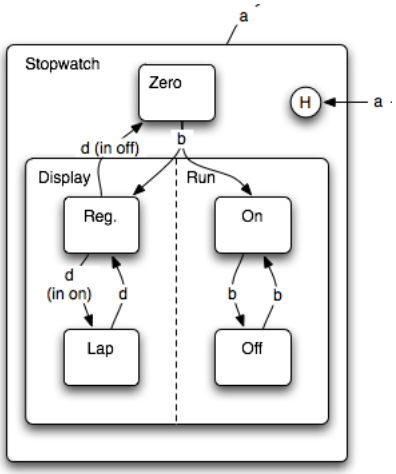

Figure 49: Harel's statechart description of Stopwatch.

The ModeGraph visual realisation of the stopwatch is rather straightforward and will not be discussed in detail. The code for the stopwatch logic can be viewed below (annotations and declarations of variables are left out to save space) and the ModeGraph realisation is shown in Figure 50 and in Figure 51.

```
algorithm
  //Increment time if run is on
  if run_on_active then
    sw_time :=mod(time - sw_time_stamp_on, 60);
    if display_lap_active then
      sw_disp1 :=sw_time_stamp_lap_secs;
      sw_disp2 :=sw_time_stamp_lap_fracs;
    else
      sw_disp1 :=integer(mod(sw_sum_secs + sw_time, 60));
      sw_disp2 :=integer(mod(sw_sum_fracs
        + mod(100*(sw_time), 100), 100));
    end if;
  //Set time to zero
  elseif zero_active then
    sw_time :=0;
    sw_disp1 :=0;
    sw_disp2 :=0;
  //Display freezed time
  else
    sw_time :=mod(sw_time + sw_time_stamp_off
      - sw_time_stamp_on, 60);
    sw_disp1 :=integer(sw_sum_secs);
    sw_disp2 :=integer(sw_sum_fracs);
  end if;

  //Zero sums
  when zero_active then
    sw_sum_secs :=0;
    sw_sum_fracs :=0;
  end when;

  //Timestamp when off and add to sums
  when run_off_active then
```

```
    sw_time_stamp_off :=time;
    sw_sum_secs :=mod(sw_sum_secs + sw_time_stamp_off
      - sw_time_stamp_on, 60);
    sw_sum_fracs :=mod(100*(sw_sum_fracs + sw_time_stamp_off
      - sw_time_stamp_on), 100);
  end when;

  //Freeze display time but continue counting in the background
  when display_lap_active then
    sw_time_stamp_lap_secs :=integer(mod(sw_sum_secs + time
      - sw_time_stamp_on, 60));
    sw_time_stamp_lap_fracs :=integer(mod(100*(sw_sum_fracs +
      time - sw_time_stamp_on), 100));
  end when;
equation

  //Timestamp when on
  when run_on_active then
    sw_time_stamp_on =  time;
  end when;
```

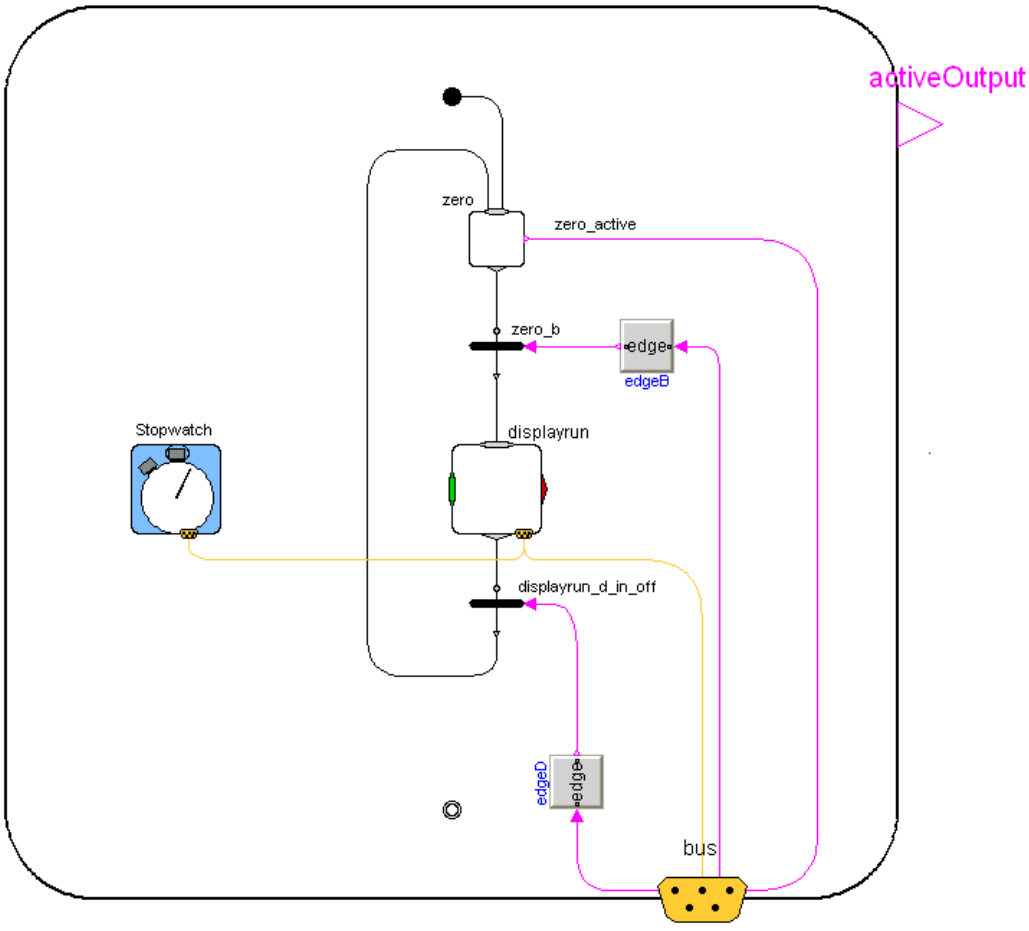

Figure 50: ModeGraph realisation of the state Stopwatch.

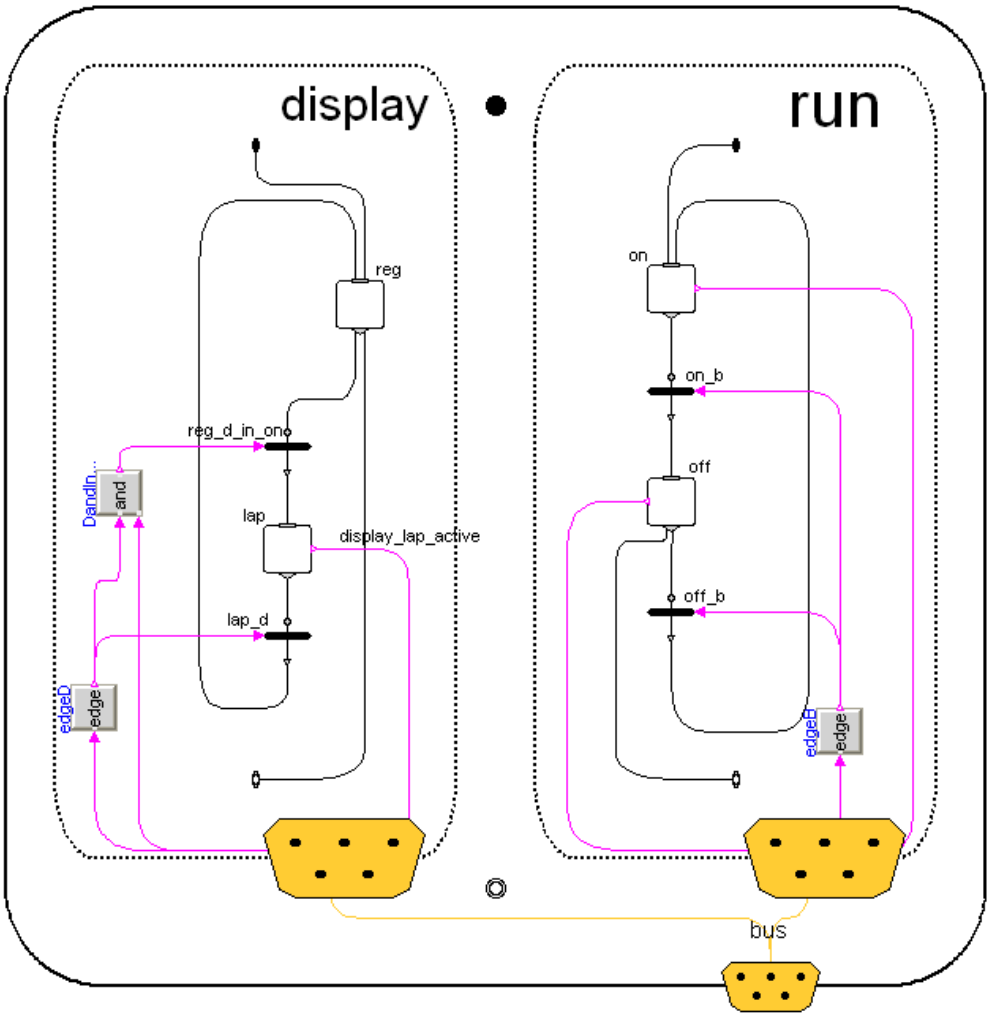

Figure 51: Different display modes of the Stopwatch realised in ModeGraph.

5. Generating AUTOSAR C-code

Assume that we have a Modelica model with a physical components and a controller part connected in a closed-loop system. It is common to simulate the entire system in this way, allowing the controller to be safely tuned. When satisfactory simulation results are acquired, the controller part can be put on a processor working on the real physical plant.

5.1 Exporting Dymola Components

In a closed-loop system modelled in Modelica, it is required to define which parts of the system that should be exported. It is needed to have a way to define a controller and its real-time properties. This is most conveniently accomplished by creating a superclass, Task, containing appropriate parameters that all models that should be exported inherit from. When translation of the system is carried out, Dymola will recognise the parts to export and treat them accordingly.

The parameters used in the Task superclass are:

```
partial block Task
  parameter String processor=""
    "Name of the processor which will run the task";
  parameter String task=""
    "Name of task in which the controller will be part";
  parameter Integer priority=1 "Priority of the task";
  parameter Real sampleTime "Sample time for task";
  parameter Real phase=0 "Phase of the task";
end Task;
```

These parameters are given default values that can be changed with modifiers to fit the controller that is inheriting from it. When the process is co-simulated, those values are used to integrate the C-code in a proper way.

5.2 AUTOSAR Compliant C-Code PI Controller

Assume a linear servo plant being controlled by a PI-controller in a closed-loop Modelica model as shown in Figure 52.

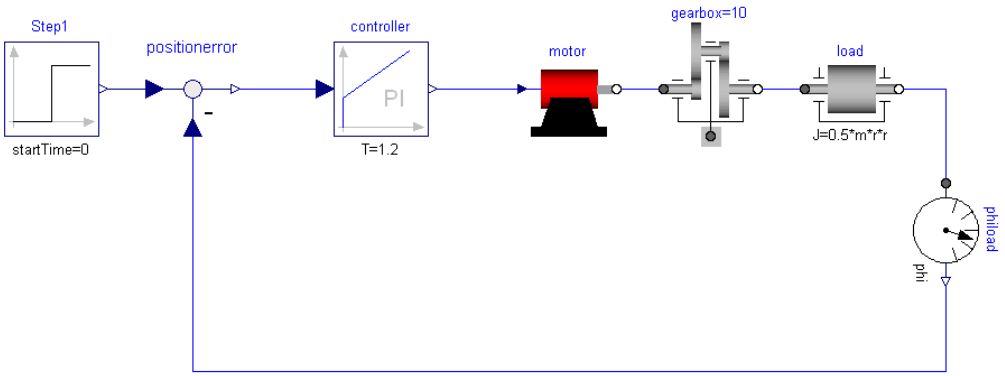

Figure 52: Closed-loop system containing a linear servo and a PI-controller.

This is a typical application example where it might be desirable to export the controller part to AUTOSAR-compliant native C-code that is eventually downloaded to an ECU. In practice, AUTOSAR-compliance means meeting specific design and naming standards of the generated C-code and an additional XML Software Component Description.

The generated C-code should, before being placed on the hardware, successfully be co-simulated with the Modelica plant. This procedure needs to be smooth and require a minimum of configuration of the co-simulation. Modifications to the original model will, however, unavoidably have to be made to adapt it to the co-simulation needs.

The co-simulation environment of choice is Extessy's platform EXITE ACE [14]. EXITE ACE makes it possible to simulate distributed systems with a variety of different tools, but also with microprocessors connected using interfaces like a Controller Area Network (CAN) bus [12].

PI Controller

The model to be exported is a modification of a standard PI-controller, taken from the Modelica.Blocks.Continuous library. The model inherits from a model, Task, that allows the Dymola translator to generate AUTOSAR-compliant C-code for this specific controller. The PI has one RealInput u and one RealOutput y with parameters:

- k = 7
- T_i = 1.2
- h = 0.1

Linear Servo Plant

The plant comprises an electric DC motor, a gearbox with a transmission ratio of 10 and a load $J = 0.5 \cdot m \cdot r^2$, where mass, m = 20 kg, and radius, r = 0.5 m. Finally a step is used as reference signal.

5.3 Structure of the Software Component Description

During generation of the AUTOSAR compliant C-code, an additional Software Component Description will also be generated. The Software Component Description is written in XML with a predefined AUTOSAR Document Type Definition (DTD) [11].

An explanation of the structure of the generated Software Component Description is presented below. An example model with a PI-controller connected to a linear servo will be used as an example. Note that the tags in the AUTOSAR Software Component Description XML file are capitalised and also hyphenated if the term contains multiple words.

XML Structure

The static part of the XML file defines the document being an AUTOSAR XML file and locates the DTD file and is thus always included.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AUTOSAR SYSTEM "AUTOSAR_MetaModel_v12342.dtd">
<AUTOSAR>
  <AR-PACKAGES>
    ...
  </AR-PACKAGES>
</AUTOSAR>
```

Furthermore, two ARPackages are defined. The first one is the exported model and the second one is a definition of the Modelica primitives Boolean, Integer and Real. String is left out at the moment, since it is not supported in the co-simulation environment. The second package is named *Modelica_Types* and is always included in the Component.

```
<AR-PACKAGE>
  <LONG-NAME>Modelica_Types</LONG-NAME>
  <SHORT-NAME>Modelica_Types</SHORT-NAME>
  <AR-ELEMENTS>
    <BOOLEAN-TYPE>
      ...
    </BOOLEAN-TYPE>
    <INTEGER-TYPE>
      ...
    </INTEGER-TYPE>
    <REAL-TYPE>
      <LONG-NAME>Real</LONG-NAME>
      <SHORT-NAME>Real</SHORT-NAME>
      <ENCODING>DOUBLE</ENCODING>
    </REAL-TYPE>
  </AR-ELEMENTS>
</AR-PACKAGE>
```

Assuming that the controller component is named *controller* in Dymola, the ARPackage containing the controller is named *Modelica_controller*.

```
<AR-PACKAGE>
  <LONG-NAME>Modelica_controller</LONG-NAME>
  <SHORT-NAME>Modelica_controller</SHORT-NAME>
  <AR-ELEMENTS>
      ...
  </AR-ELEMENTS>
```

```
</AR-PACKAGE>
```

The following definitions are then made within the Modelica_controller package:

- An Atomic Software Component is defined and named *Modelica_controller_Component*, also referring to the Dymola component name.

```
<ATOMIC-SOFTWARE-COMPONENT-TYPE>
  <LONG-NAME>Modelica_controller_Component</LONG-NAME>
  <SHORT-NAME>Modelica_controller_Component</SHORT-NAME>
    ...
</ATOMIC-SOFTWARE-COMPONENT-TYPE>
```

  Within the Atomic Software Component definition, the asynchronous sender-receiver communication ports are defined with PortPrototypes. In AUTOSAR, inputs and outputs are called RequirePorts (RPorts) and ProvidePorts (PPorts), respectively. The naming of the ports combines the component name and the Dymola name of the port, e.g., the output y of controller will be named *controller_y*.

```
<PORT-PROTOTYPES>
  <R-PORT-PROTOTYPE>
    ...
  </R-PORT-PROTOTYPE>
  <P-PORT-PROTOTYPE>
  <LONG-NAME>controller_y</LONG-NAME>
  <SHORT-NAME>controller_y</SHORT-NAME>
  <SENDER-RECEIVER-INTERFACE-TREF>
    /Modelica_controller/RealSignal
  </SENDER-RECEIVER-INTERFACE-TREF>
  </P-PORT-PROTOTYPE>
</PORT-PROTOTYPES>
```

- An Internal Behavior named *Modelica_controller_InternalBehavior* is defined and within the Internal Behavior a Runnable that lets the RTE know the name and existence of a runnable piece of code is defined. The Runnable is named *Modelica_controller_Runnable*. The reference ties it to the AtomicSoftwareComponent in question.

```
<INTERNAL-BEHAVIOR>
  <LONG-NAME>Modelica_controller_InternalBehavior</LONG-
  NAME>
  <SHORT-NAME>Modelica_controller_InternalBehavior</SHORT-
  NAME>
  <ATOMIC-SOFTWARE-COMPONENT-TYPE-REF>
  Modelica_controller_Component
  </ATOMIC-SOFTWARE-COMPONENT-TYPE-REF>
  <RUNNABLE-ENTITYS>
    <RUNNABLE-ENTITY>
      <LONG-NAME>Modelica_controller_Runnable</LONG-NAME>
      <SHORT-NAME>Modelica_controller_Runnable</SHORT-NAME>
    </RUNNABLE-ENTITY>
  </RUNNABLE-ENTITYS>
</INTERNAL-BEHAVIOR>
```

- SenderReceiverInterfaces corresponding to the three primitives are always defined here, named *BooleanSignal*, *IntegerSignal* and *RealSignal*.

```
<SENDER-RECEIVER-INTERFACE>
```

```
  <LONG-NAME>BooleanSignal</LONG-NAME>
  <SHORT-NAME>BooleanSignal</SHORT-NAME>
  ...
</SENDER-RECEIVER-INTERFACE>
<SENDER-RECEIVER-INTERFACE>
  <LONG-NAME>IntegerSignal</LONG-NAME>
  <SHORT-NAME>IntegerSignal</SHORT-NAME>
  ...
</SENDER-RECEIVER-INTERFACE>
<SENDER-RECEIVER-INTERFACE>
  <LONG-NAME>RealSignal</LONG-NAME>
  <SHORT-NAME>RealSignal</SHORT-NAME>
  <DATA-ELEMENT-PROTOTYPES>
    <DATA-ELEMENT-PROTOTYPE>
      <LONG-NAME>value</LONG-NAME>
      <SHORT-NAME>value</SHORT-NAME>
      <REAL-TYPE-TREF>/Modelica_Types/Real</REAL-TYPE-TREF>
      <INFO-TYPE>data</INFO-TYPE>
    </DATA-ELEMENT-PROTOTYPE>
  </DATA-ELEMENT-PROTOTYPES>
</SENDER-RECEIVER-INTERFACE>
```

The complete XML file of the PI example can be viewed in Appendix 9.1.

5.4 RTE Function Calls

The generated C-file will naturally come out very different depending on the actual model being exported. The important issue in this context is how communication with the RTE is carried out, and how this is related to the description file. The C-file is named after the runnable in the description file, which in the PI-example will be *Modelica_controller_Runnable*.

When data is sent and received, function calls to the RTE are made. The names of those are constructed as follows.

- Inputs values are read by calling:
  `Rte_IRead_<name of runnable>_<name of signal>_value()`
  Hence, in the PI-example, the input u is read with:
  `Rte_IRead_Modelica_controller_Runnable_controller_u_value()`
- Outputs are accordingly written:
  `Rte_IWrite_<name of runnable>_<name of signal>_value(output)`
  which for the output **y** in the PI-example will turn out:
  `Rte_IWrite_Modelica_controller_Runnable_controller_y_value(controller_y)`

The RTE makes sure that the function calls are carried out depending on how the rest of the system is configured and how ports are connected to each other. In the co-simulation environment RPorts can be connected to PPorts and vice versa. It is worth mentioning that the function calls above are macros that internally will execute operations on the respective runnables.

5.5 Extessy ACE Integration

It is a great advantage to be able to perform SIL-simulations of the exported code with the simulated plant before the code is downloaded to the hardware. This will ensure that the exported controller is coherent with the model and that chosen

processor parameters are reasonable and work with the simulated plant. To be able to achieve this, a co-simulation environment is required.

EXITE ACE [14] is a powerful co-simulation platform that in addition to Dymola provides support for multiple simulators, such as Simulink, ASCET, Rhapsody, Artisan Studio, and C/C++. ACE also supports import of AUTOSAR compliant C-code and Software Component Descriptions. These are rather unique features at the present, and the existing support for co-simulation with Dymola made it an easy choice of environment.

C-Code Export of a Modelica Controller

To illustrate ACE-integration and Dymola co-simulation of exported AUTOSAR C-code, the previously mentioned PI-controlled motor drive will be used as an example. Before exporting the controller, two XML files need to be present in /$DYMOLA/insert where $DYMOLA is the Dymola installation path:

- ModelicaElementTypesForAutosar.xml
- ModelicaSignalTypesForAutosar.xml.

After setting the TranslateToAutosar-option, translation of the motor drive model will generate a C-code file named Modelica_controller_Runnable.c and a SW-C description called Modelica_controller.xml, both located in the current working directory of Dymola.

Importing the AUTOSAR SW-C in EXITE ACE

Assume that the PI-controller is successfully exported as described above. Even though the export is automated, the system still needs to be set up and configured for co-simulation.

- Start EXITE ACE and create a new Project by right-clicking the Workspace view and choose New → Project.
- To import the AUTOSAR SW-C description, select File → Import → Autosar and find the xml file generated by Dymola. The Workspace View should now look something like:

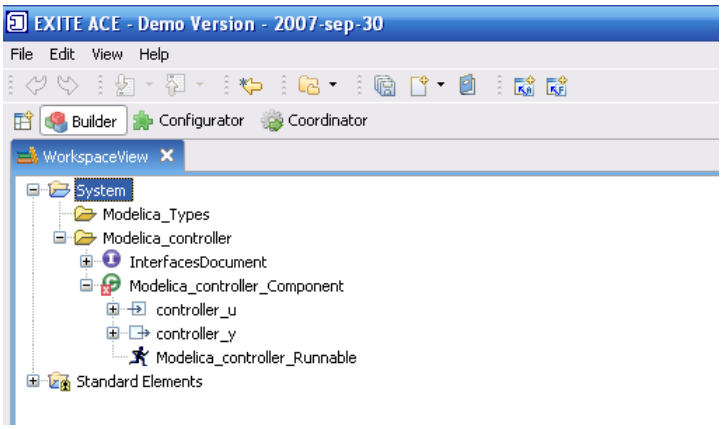

Figure 53: Imported AUTOSAR SW-C description.

- Open the Component by double-clicking Modelica_controller_Component. Right-click the Component icon and choose Add new model. Select AUTOSAR Model, and click Finish. The Component is now an AUTOSAR-Component.

- Right-click the Component view again and choose Generate adapter. This will create a directory, genPath_Modelica_controller_Component, where all AUTOSAR-related files are placed.
- Copy the generated C-file Modelica_controller_Runnable.c from the Dymola working directory to the recently created genPath directory. Additionally, two predefined files, Rte_Type.h and user_make.mk, need to be put in this directory.
- Right click AUTOSAR-Component and choose Generate Code and build DLL, see Figure 54. The AUTOSAR code should now compile without errors.

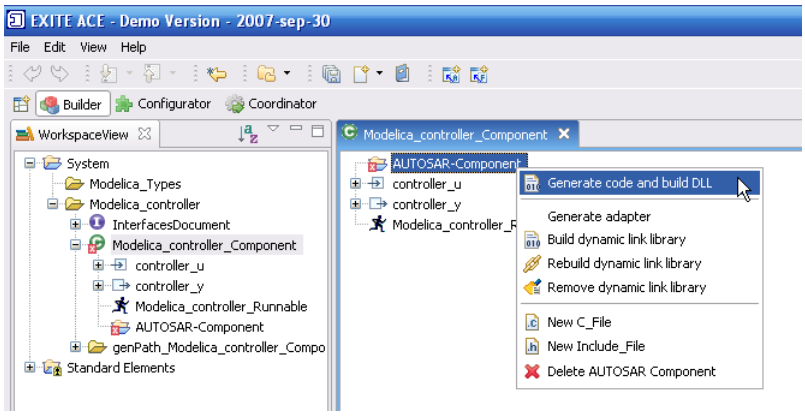

Figure 54: Generate AUTOSAR code and build the DLL.

- Create a new directory at root package level called Modelica_Process.
- Right-click the Modelica_Process directory and choose New → Component and then select Dymola Model to add a new Dymola Component.
- Open the Component view, right-click and choose New Port. Add one require port, u, and one provide port, y. Both should have Referenced Interface set to RealSignal, see Figure 55.

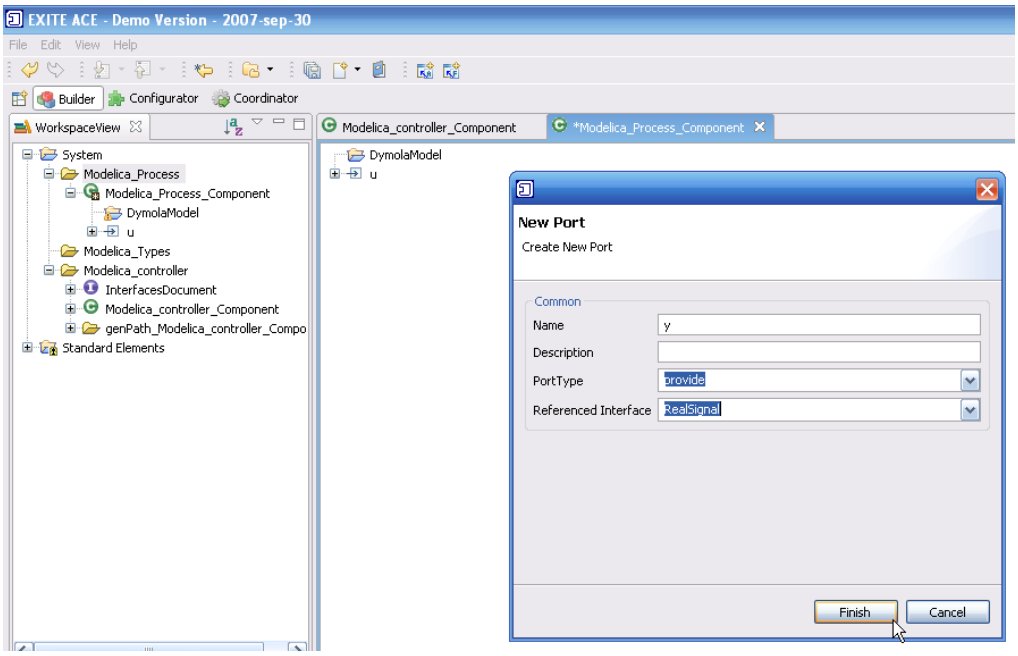

Figure 55: Add provide port y to the Modelica process.

- Right click in the Component view and choose Create Dymola model and name it Process.mo.
- In Dymola, open the EXITE ACE library located in /$EXITE-ACE/lib/modelica/EXITE_ACELibrary.mo. Then continue by opening Process.mo and create a new model in the package and name it Modelica_Process. Use the original motor drive model and copy it to the Modelica_Process model and remove the PI-controller. Instead place and connect the u and y Modelica port components that are located in the Process package to provide a communication interface for EXITE ACE.

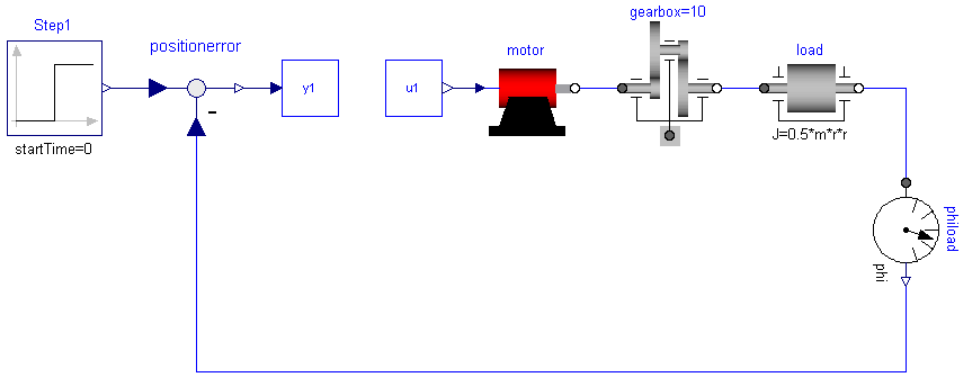

Figure 56: Modelica process model with ports enabling communication with EXITE ACE.

- Translate and save the model. Remember to remove the TranslateToAutosar option.
- Right click the WorkSpaceView tab in EXITE ACE and choose New → Composite.
- Open the Composite and drag both the Controller and Process Components inside the Composite view. They should both have Sample time = 0.01 and Start time = 0.0, see Figure 57. Be careful not to replace the Composite Table with one of the Components.

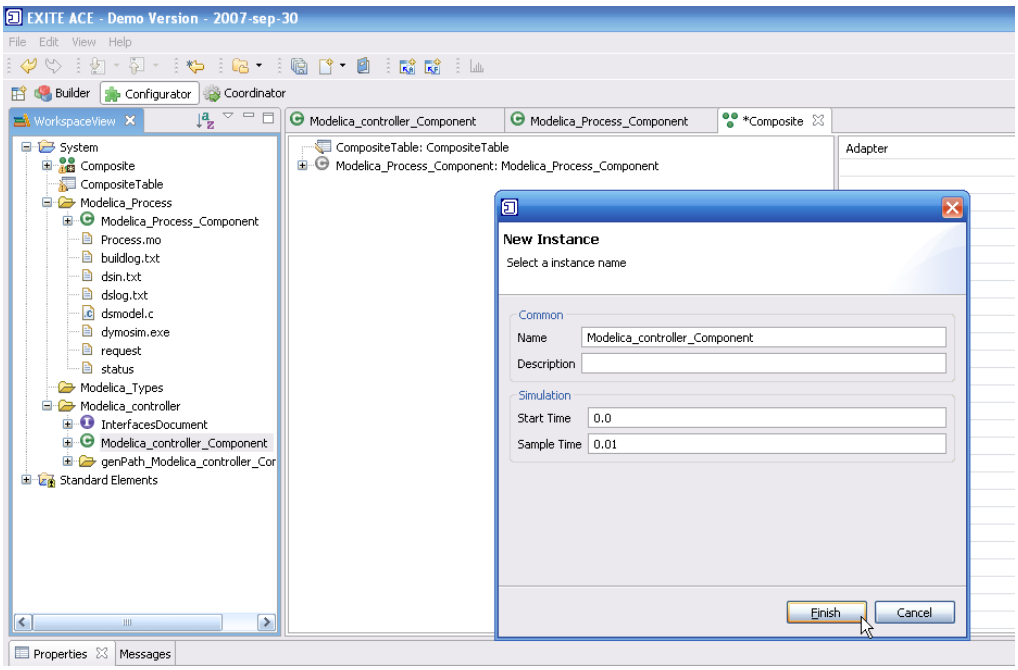

Figure 57: Modelica_controller_Component added to the Composite.

- Similarly, drag the runnable to the Controller Components instance in the Composite view. This will create a new Sample Rate. Start and Sample time is the same as above.
- Create two ident adapters by right-clicking the Composite view and choose New Adapter → Ident. Connect the controller inports to the process outports by dragging them from the Composite view to the respective ident connections, see Figure 58.

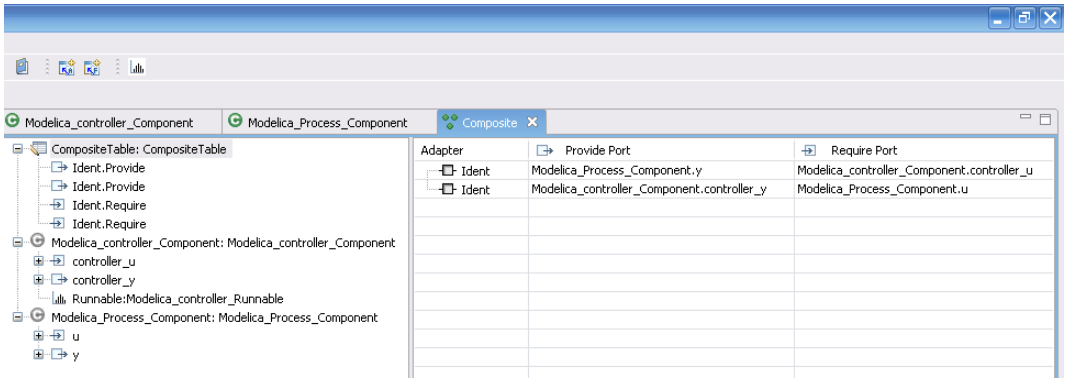

Figure 58: Composite with two components, one runnable and two idents.

- Right-click root package and choose new → Simulation Parameters. Use the present Composite as the target, add a suitable name and press Finish.
- Double-click the Simulation Parameters and click the allocation tab. Create a new host, by right clicking the window above allocation tab and choosing New Host Node. Name it localhost. Right click localhost in the window to the right, and choose New Process.
- Switch back to the Simulation tab. Tick the Realtime box and set Stop time to 30.

- Enter Simulation → Setup in Dymola. Set Stop time to 30, and set interval length to 0.01. Choose the Realtime tab and tick the Synchronize with realtime box.
- Now simulation is ready to commence. Start the simulation first in ACE and then in Dymola. Open a plot window in Dymola and something like in Figure 59 will be shown.

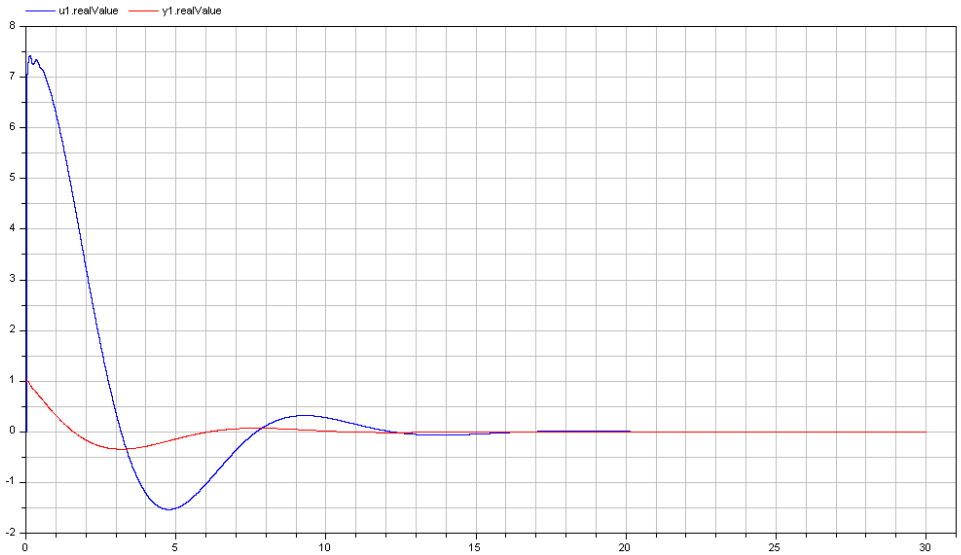

Figure 59: Error signal (blue) and control signal (red) in co-simulation.

6. Results

In this thesis the ModeGraph library has been introduced. The motivation for ModeGraph originates in the inadequacy of StateGraph in terms of implementing Statechart-oriented reactive systems. 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 from 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.
Additionally, an interface for code-generation compatible with the AUTOSAR standards has been presented. The new features introduce the possibility of exporting a controller part of a Modelica model (that inherits from Task) to C-code that is directly integrateable with either HIL/SIL-simulation or to the actual hardware process. This provides enhanced flexibility by offering an abstraction of the communication layer, i.e. communication between exported components are handled by an underlying RTE and is independent of the physical placement.

7. Future Work

The ModeGraph library is currently at a prototype stage. The basic semantics and rules for conditional execution have been defined. However, there are yet many refinements that need to be investigated and defined. An important area is analysis of unsafe graphs. Although this topic has unfortunately landed outside the scope of this thesis, ModeGraph has a solid foundation to carry out such analysis by forcing strict hierarchical instantiation of composite states.

Having introduced conditional execution leaves a vast number of cases to be researched in terms of sorting of equations and handling of conditionally reset variables. It is also worth mentioning that the Dymola support of conditional execution is merely at a prototype level at the moment. The Mode could successfully be accepted as a built-in class in Modelica, and gathering of assignments conditionally assigned with respect to Modes should be automatically gathered into efficient code.

A topic that should not be forgotten is facilitation of usage. Powerful GUI-support of all the features presented in this thesis would greatly enhance the user experience of graphical modelling. Making it possible to assign a value by opening the dialogue of a Mode is absolutely necessary to make the feature accessible without too much insight into the specifics of Mode-Automata.

More benchmarks, especially in the hybrid context would be beneficial to investigate the capacity of ModeGraph and its capabilities.

When it comes to generating AUTOSAR-compliant C-code, only a minor feasibility case-study has been researched and tested. Results confirm that C-code can successfully be exported that is easily integrated with either a co-simulation environment or a physical plant. However, this field is huge. Assume for instance that Harel's Wristwatch is to be divided and placed on three different microprocessors with one or several AUTOSAR-interfaced RTEs. The mere thought makes it easy to realise that there is a lot more work to be done.

8. References

[1] Charles André
I3S Laboratory – UMR 6070 University of Nice-Sophia Antipolis / CNRS
Semantics of S.S.M (Safe State Machine)
2003

[2] Karl-Erik Årzén, Rasmus Olsson, Johan Åkesson
Lund Institute of Technology, Sweden
Grafchart for Procedural Operator Support Tasks
15th Triennial World Congress, Barcelona, Spain, 2002

[3] Karl-Erik Årzén
Department of Automatic Control, Lund Institute of Technology, Sweden
JGrafChart
http://www.control.lth.se/~karlerik/Grafchart/JGrafchart.html

[4] Hilding Elmqvist
SattControl AB, S-205 22 Malmö, Sweden
Cooperated Distributed Control Objects
IFAC Symposium on Distributed Intelligence Systems
August 13-15, 1991, Arlington, Virginia, USA

[5] David Harel
Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel
Statecharts: A Visual Formalism for Complex Systems
Science of Computer Programming 8 (1987) 231-274

[6] Florence Maraninchi and Yann Rémond
VERIMAG - Centre Equation, 2 Av. de Vignate - F38610 GIERES
Mode-Automata: About Modes and States for Reactive Systems
http://www.imag.fr/VERIMAG/PEOPLE/Florence.Maraninchi
1998

[7] F. Maraninchi and Y. Rémond
Mode-Automata: a new Domain-Specific Construct for the Development of Safe Critical Systems
http://www-verimag.imag.fr/%7Emaraninx/SCP2002.html
2003

[8] Martin Otter, Karl-Erik Årzén, Isolde Dressler
DLR Oberpfaffenhofen, Germany; Lund Institute of Technology, Sweden
StateGraph - A Modelica Library for Hierarchical State Machines
Proceedings of the 4th International Modelica Conference, 2005

[9] AFCET
J. Automatique et Informatique Industrielle
Normalisation de la representation du cahier des charges d'un automatisme logique
1977

[10] AUTOSAR GbR
AUTOSAR Technical Overview 2.0.1
http://www.autosar.org

[11] AUTOSAR_MetaModel_v12342.dtd

[12] CAN in Automation
Controller Area Network
http://www.can-cia.org/cia/

[13] Dynasim AB
Dymola
http://www.dynasim.se

[14] Extessy AG
EXITE ACE
http://www.extessy.com

[15] Modelica Association
Modelica
http://www.modelica.org

9. Appendix

9.1 AUTOSAR SW-C description

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE AUTOSAR SYSTEM "AUTOSAR_MetaModel_v12342.dtd">
<AUTOSAR>
  <AR-PACKAGES>
    <AR-PACKAGE>
      <LONG-NAME>Modelica_controller</LONG-NAME>
      <SHORT-NAME>Modelica_controller</SHORT-NAME>
      <AR-ELEMENTS>
        <ATOMIC-SOFTWARE-COMPONENT-TYPE>
          <LONG-NAME>Modelica_controller_Component</LONG-NAME>
          <SHORT-NAME>Modelica_controller_Component</SHORT-NAME>
          <PORT-PROTOTYPES>
            <R-PORT-PROTOTYPE>
              <LONG-NAME>controller_u</LONG-NAME>
              <SHORT-NAME>controller_u</SHORT-NAME>
              <SENDER-RECEIVER-INTERFACE-TREF>
                  /Modelica_controller/RealSignal
              </SENDER-RECEIVER-INTERFACE-TREF>
            </R-PORT-PROTOTYPE>
            <P-PORT-PROTOTYPE>
              <LONG-NAME>controller_y</LONG-NAME>
              <SHORT-NAME>controller_y</SHORT-NAME>
              <SENDER-RECEIVER-INTERFACE-TREF>
                  /Modelica_controller/RealSignal
              </SENDER-RECEIVER-INTERFACE-TREF>
            </P-PORT-PROTOTYPE>
          </PORT-PROTOTYPES>
        </ATOMIC-SOFTWARE-COMPONENT-TYPE>
        <INTERNAL-BEHAVIOR>
          <LONG-NAME>
            Modelica_controller_InternalBehavior
          </LONG-NAME>
          <SHORT-NAME>
            Modelica_controller_InternalBehavior
          </SHORT-NAME>
          <ATOMIC-SOFTWARE-COMPONENT-TYPE-REF>
            Modelica_controller_Component
          </ATOMIC-SOFTWARE-COMPONENT-TYPE-REF>
          <RUNNABLE-ENTITYS>
            <RUNNABLE-ENTITY>
              <LONG-NAME>
                  Modelica_controller_Runnable
              </LONG-NAME>
              <SHORT-NAME>
                  Modelica_controller_Runnable
              </SHORT-NAME>
            </RUNNABLE-ENTITY>
          </RUNNABLE-ENTITYS>
        </INTERNAL-BEHAVIOR>
        <SENDER-RECEIVER-INTERFACE>
```

```
      <LONG-NAME>BooleanSignal</LONG-NAME>
      <SHORT-NAME>BooleanSignal</SHORT-NAME>
      <DATA-ELEMENT-PROTOTYPES>
        <DATA-ELEMENT-PROTOTYPE>
          <LONG-NAME>value</LONG-NAME>
          <SHORT-NAME>value</SHORT-NAME>
          <REAL-TYPE-TREF>
              /Modelica_Types/Boolean
          </REAL-TYPE-TREF>
          <INFO-TYPE>data</INFO-TYPE>
        </DATA-ELEMENT-PROTOTYPE>
      </DATA-ELEMENT-PROTOTYPES>
    </SENDER-RECEIVER-INTERFACE>
    <SENDER-RECEIVER-INTERFACE>
      <LONG-NAME>IntegerSignal</LONG-NAME>
      <SHORT-NAME>IntegerSignal</SHORT-NAME>
      <DATA-ELEMENT-PROTOTYPES>
        <DATA-ELEMENT-PROTOTYPE>
          <LONG-NAME>value</LONG-NAME>
          <SHORT-NAME>value</SHORT-NAME>
          <REAL-TYPE-TREF>
              /Modelica_Types/Integer
          </REAL-TYPE-TREF>
          <INFO-TYPE>data</INFO-TYPE>
        </DATA-ELEMENT-PROTOTYPE>
      </DATA-ELEMENT-PROTOTYPES>
    </SENDER-RECEIVER-INTERFACE>
    <SENDER-RECEIVER-INTERFACE>
      <LONG-NAME>RealSignal</LONG-NAME>
      <SHORT-NAME>RealSignal</SHORT-NAME>
      <DATA-ELEMENT-PROTOTYPES>
        <DATA-ELEMENT-PROTOTYPE>
          <LONG-NAME>value</LONG-NAME>
          <SHORT-NAME>value</SHORT-NAME>
          <REAL-TYPE-TREF>
              /Modelica_Types/Real
          </REAL-TYPE-TREF>
          <INFO-TYPE>data</INFO-TYPE>
        </DATA-ELEMENT-PROTOTYPE>
      </DATA-ELEMENT-PROTOTYPES>
    </SENDER-RECEIVER-INTERFACE>
  </AR-ELEMENTS>
</AR-PACKAGE>
<AR-PACKAGE>
  <LONG-NAME>Modelica_Types</LONG-NAME>
  <SHORT-NAME>Modelica_Types</SHORT-NAME>
  <AR-ELEMENTS>
    <BOOLEAN-TYPE>
      <LONG-NAME>Boolean</LONG-NAME>
      <SHORT-NAME>Boolean</SHORT-NAME>
    </BOOLEAN-TYPE>
    <INTEGER-TYPE>
      <LONG-NAME>Integer</LONG-NAME>
      <SHORT-NAME>Integer</SHORT-NAME>
      <LOWER-LIMIT>
        <INTERVAL-TYPE>CLOSED</INTERVAL-TYPE>
        <VALUE>-2147483648</VALUE>
      </LOWER-LIMIT>
      <UPPER-LIMIT>
```

```
            <INTERVAL-TYPE>CLOSED</INTERVAL-TYPE>
            <VALUE>2147483647</VALUE>
          </UPPER-LIMIT>
        </INTEGER-TYPE>
        <REAL-TYPE>
          <LONG-NAME>Real</LONG-NAME>
          <SHORT-NAME>Real</SHORT-NAME>
          <ENCODING>DOUBLE</ENCODING>
        </REAL-TYPE>
      </AR-ELEMENTS>
    </AR-PACKAGE>
  </AR-PACKAGES>
</AUTOSAR>
```

9.2 Harel's Wristwatch complete statecharts

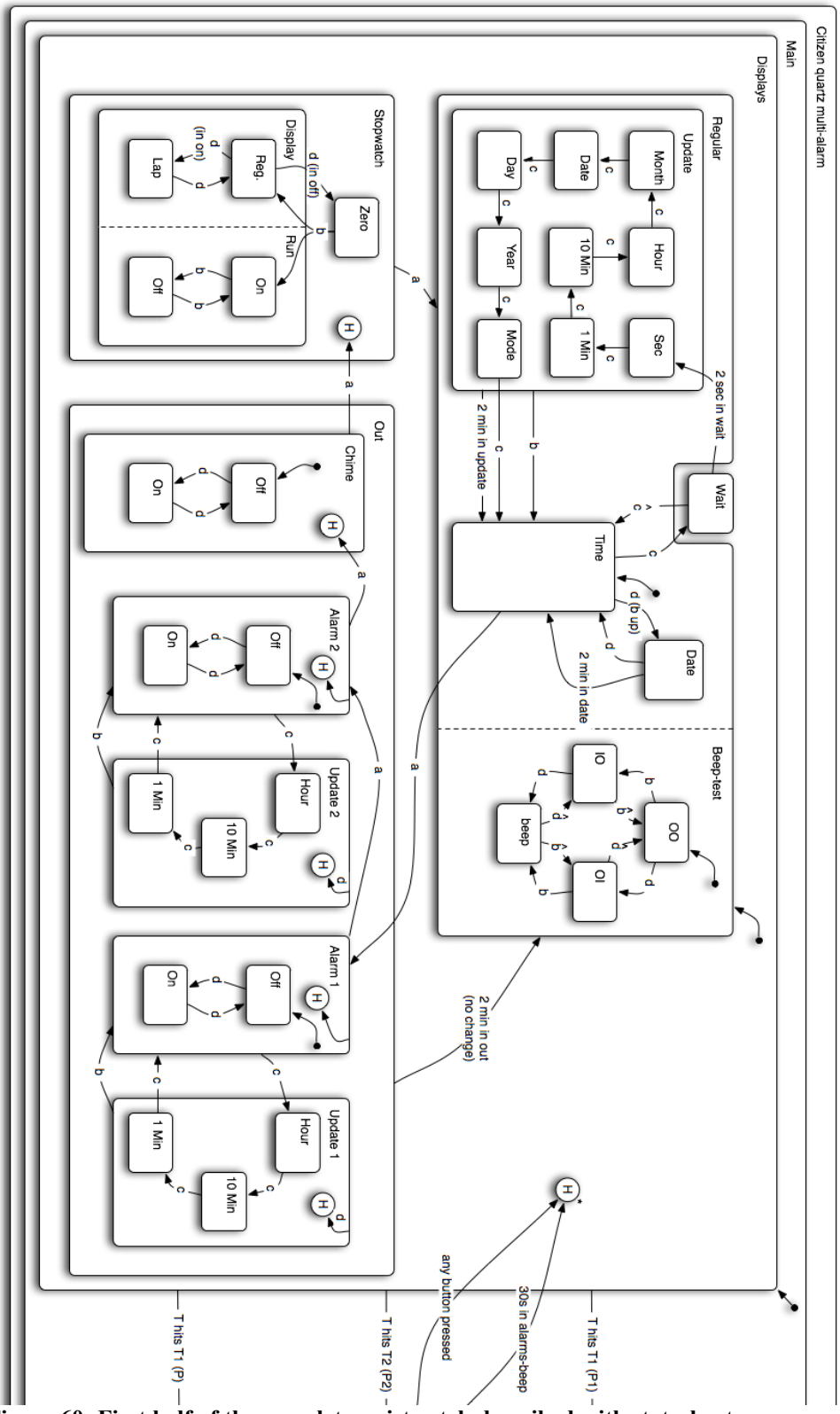

Figure 60: First half of the complete wristwatch described with statecharts.

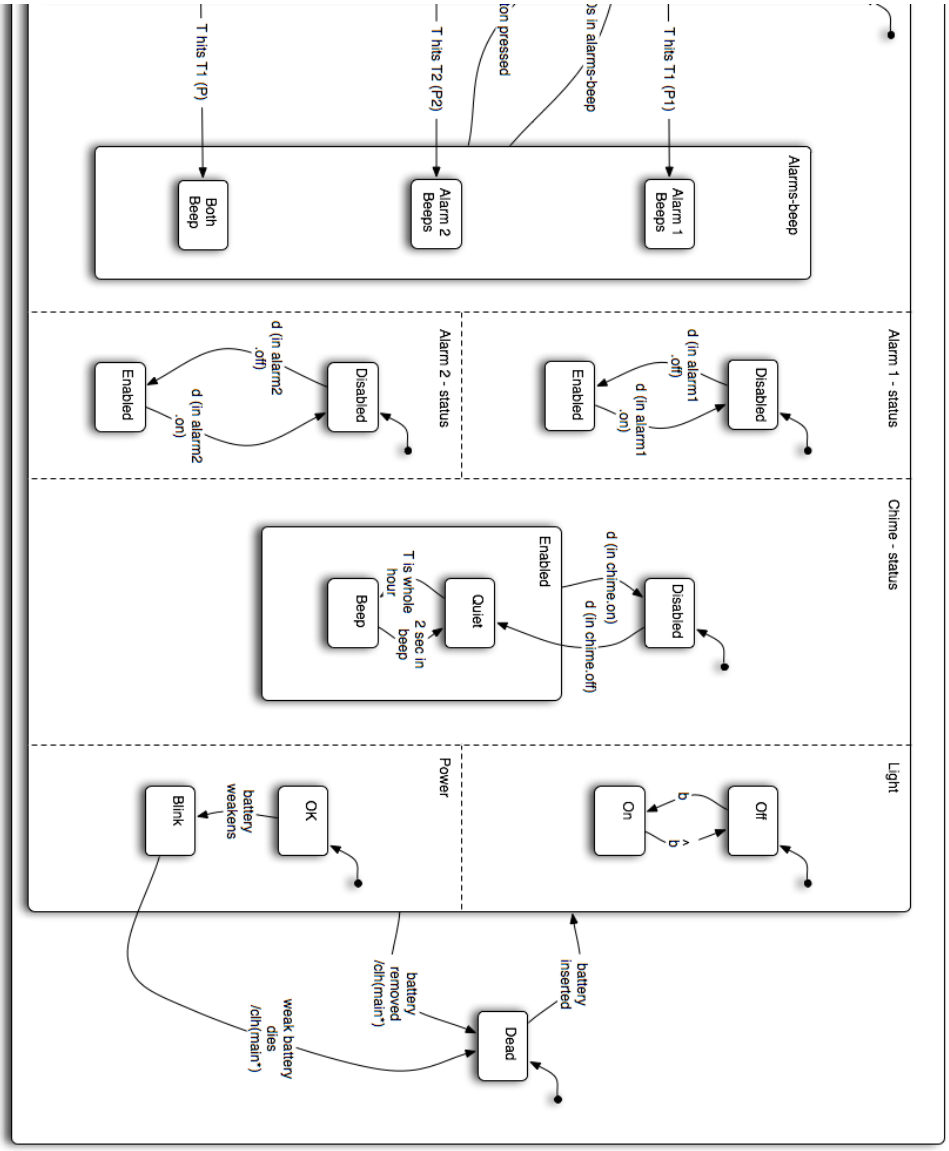

Figure 61: Second half of the complete wristwatch described with statecharts.