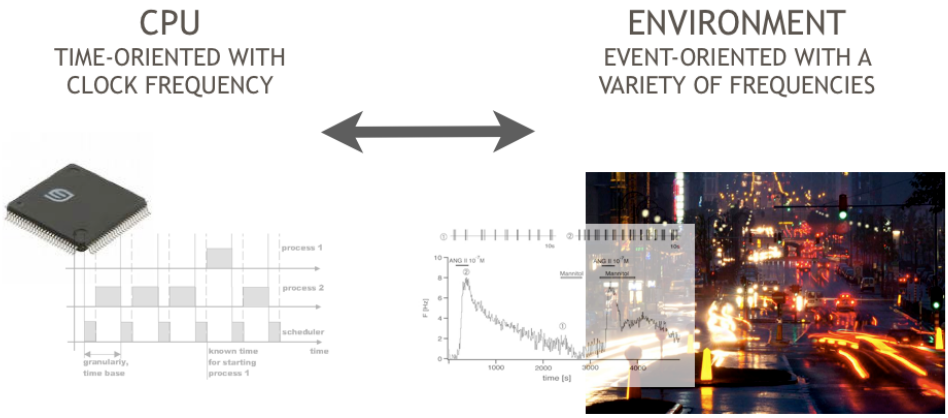
Architecture Design

IBM® Rational® Rhapsody® Advanced Training

WILLERT.



Architecture Design using UML and Rhapsody

Architecture Design aims at a large number of viewpoints of a system and the structural setup with regard to these viewpoints.
Traditional Architecture Design is primarily seen as the static partitioning of the system in logical components. But that is only one field.

Other fields are the presentation and structuring of different versions (historical development) and variants and their differences to one another or even miscellaneous operation modes..

Finally there is the perspective on the dynamic behavior, also know as runtime architecture.

All these fields can be addressed with UML and the respective diagrams. During this part of the training we will focus mainly on static and runtime architecture.

Especially the runtime architecture has effects on the type of modeling in UML and the therefor meaningful diagrams to be used.

Index:

Static and dynamic architecture design.

Runtime Architecture Design Pattern.

Decision in principle: Time-driven or event-driven.

State machines and events.

State machines and signals.

Background information regarding modeling based on state machines.

Background information regarding modeling based on activity diagrams and flow charts with Rhapsody.

Advantages & disadvantages.

Architecture Design Basics

Static and dynamic architecture design?

The so-called static architecture design is all about dividing the system into logical units. In UML this is done, inter alia, in form of different classes and objects.

The logical processes are modeled or programmed within these classes (objects) i.e. in form of state machines.

The question that remains is: how does the behavior of different classes affect each other when executed?

Dividing the computational power into specific dynamic shares of the classes is the dynamic architecture design or runtime architecture design.

Runtime Architecture Design Pattern

The main() Loop is one of the simplest methods to assign dynamic shares to the cpu. In this case the behavior of a class is executed by the cpu on after the other.

The disadvantage is that the periodicity is the same for all and from the point of view of fast reactions relatively slow. Sometimes it is desirable and necessary that single reactions are executed in less time than the periodicity of the the main() Loop allows. These would then be executed separately and simultaneously to the main() Loop, ie. ISR's (Interrupt Service Routines).

These can interrupt the dynamic execution of code in the main() Loop (Preemption).

From the viewpoint of time there are a several more possibilities to assign the execution of single dynamic shares from the static architecture to the computational power of the cpu. In other words: to allocate the computational power of the cpu to the respective dynamic shares of the architecture.

Hereby there are two fundamental viewpoints. The allocation is either done time- or event-oriented.

Decision in principle: Time or event

The main allocation of computational power is time-oriented (main() Loop, time-slicing ...). The cpu time is thereby divided into cycles. A cycle is allocated to every single Unit to be executed. The cycles can be of different thus response times that vary can be displayed.

The reuse of software components in such a system is difficult, as dynamic architecture must be ensured next to the static allocation of a system.

The alternative is an event-oriented dynamic allocation of components to be executed to the cpu.

The advantage of these systems is a more simple reuse, as the runtime architecture dynamically adapts to the occurrence of events. The disadvantage is a more dynamic behavior with poor deterministic behavior. (Predictability of temporal behavior.).

In UML it is important to design the modeling of runtime architecture in a way that corresponds to an event-oriented and time-oriented manner.

Why architecture design?

- REDUCTION OF COMPLEXITY (DIVIDE & CONQUER)
- RESOLVING DEPENDENCIES (TO THE HARDWARE, COMPONENTS AMONG THEMSELVES...)
- IMPROVEMENT OF QUALITY ATTRIBUTES IN GENERAL SOFTWARE (COMPREHENSIBILITY, CHANGEABILITY REUSABILITY, ROBUSTNESS)

Events and signals??

As the stimulant of our system and also as data flow within the system it is important regarding architecture to distinguish event oriented data flow from time-continuous (so-called signals) data flow.

The nature of events and signals differ significantly

Events

Must be classified as discrete from a temporal point of view. Thus, events are valid at a particular time and they represent this time, even though their lifetime may exist beyond.

Time sequences must be displayed with the appropriate number of events. Gaps in the range of values over time cannot be ruled out (Sampling theorem). Events may very well be used in event-oriented architecture, they also harmonize very well with bus systems for transmitting data.

Signals

From a temporal point of view signals are called continuously adjacent values. Easy comparable to a cable connected to a port to which an analogue signal is applied i.e. voltage. The signal represents this moment and is exactly valid in that moment. Signals can be used very well with time-oriented architecture. They can be retrieved at any moment in time und provide a valid value.

State machines and events.

The most used and common approach to the modeling of behavior in embedded systems is based in the use of UML state machines.

Now one has to know that the state machine assumes an event-driven System (by Harel).

In Figure No. I shows a typical behavior or a part off MMI in the form of brightness and volume control.

The transitions are implemented as events. The system has three buttons, one to switch from brightness to volume control and vice versa and one for increase and one reduction of brightness or volume.

When a button is pressed, then an event is generated and stored in an event queue.

Behind the scenes there is a scheduler running. The scheduler fetches on event after the other from the queue and activates the respective state machine that is waiting for precisely this event. The state machine executes the corresponding transition and performs the specified action in this state and passes control back to the scheduler. Similarly, all events are processed precisely in the same order they occur.

An important feature of this architecture: Temporal disruptions related to the state machine will not affect the logical behavior. This is illustrated much better by giving an example.

Lets suppose the system is in brightness state and we want to increase brightness. The event mode would be generated first and the minus event afterwards.

Even when the two buttons are pushed one after the other in a very short time and the system was busy at that very moment with the execution of another state machine, the events would be processed in the exact order as soon as computing time is allocated to our state machine.

Attention:

Timedriven runtime architecture with signals in the dataflow are not modelled very well on the basis of state machines.

Activity diagramms are much better suited for this purpose as orders can be precisely defined within these diagramms based on the history of signal changes.

Inefficient processing is the disadvantage. Activity diagramms are processed each time, state machines ony switch to the new state.

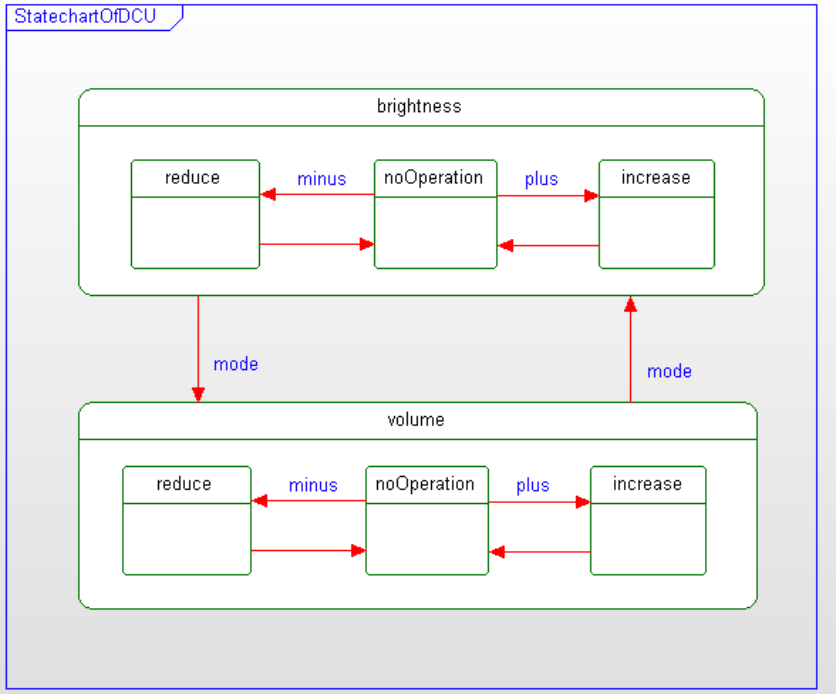
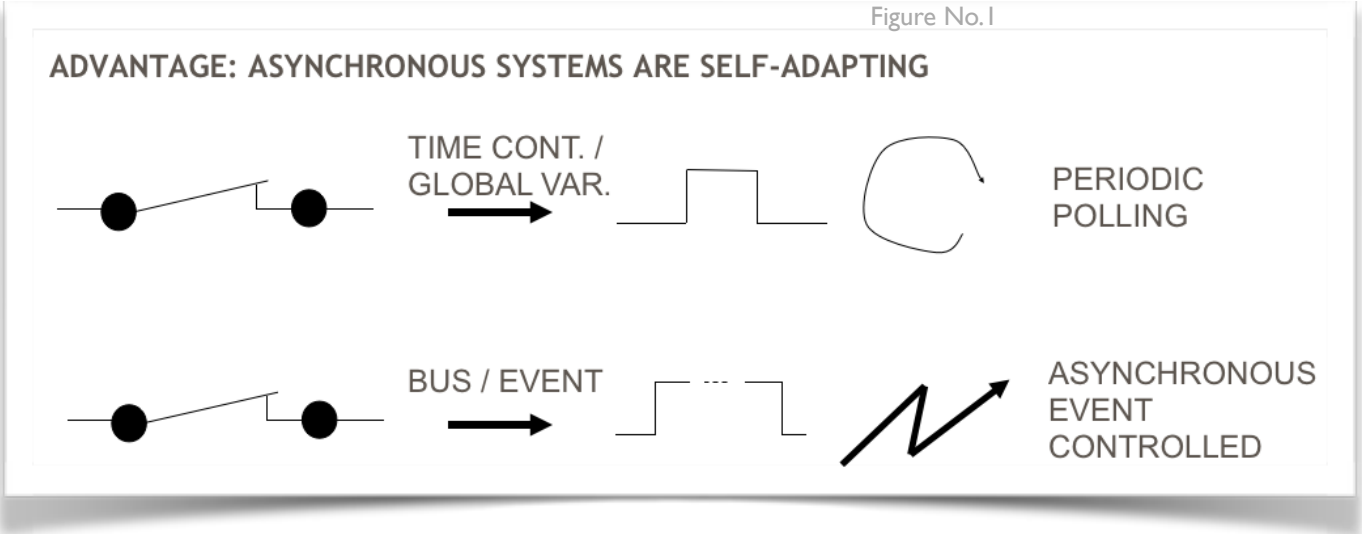


Figure No. I



State machines and signals

Lets assume we do not have buttons but we have switches. The nature of input changes from events to signals

Accordingly, we could model our state machine based on variables, which contents represents the position of the switch as shown in Figure No. 2.

Considering the behavior over time we encounter one problem. State machines by Harel, do not know an explicit indication of the order of transitions. Usually the order of events is specified in the event queue.

In this case, there is no queue. When relevant signals change during a period of active processing of the state machine to the next, it can lead to undefined states of decision.

If for example the operation mode is switched and the plus signal is queuing then the operating mode should be changed first and then the reaction on the plus signal should follow.

This order cannot be specified in state machines.

In practice, more complex situations arise in which orders are to be considered in logical decisions, even across multiple states.

Modeling on the basis of state machines is not well suited for timedriven signal-oriented systems. Here it is better to model on the basis of activity diagrams.

Attention:

Rhapsody code generation from activity diagramms is very very similar to state machine.

Sometimes it makes more sense to use flowcharts.

However, these should be integrated into classes in a different way.

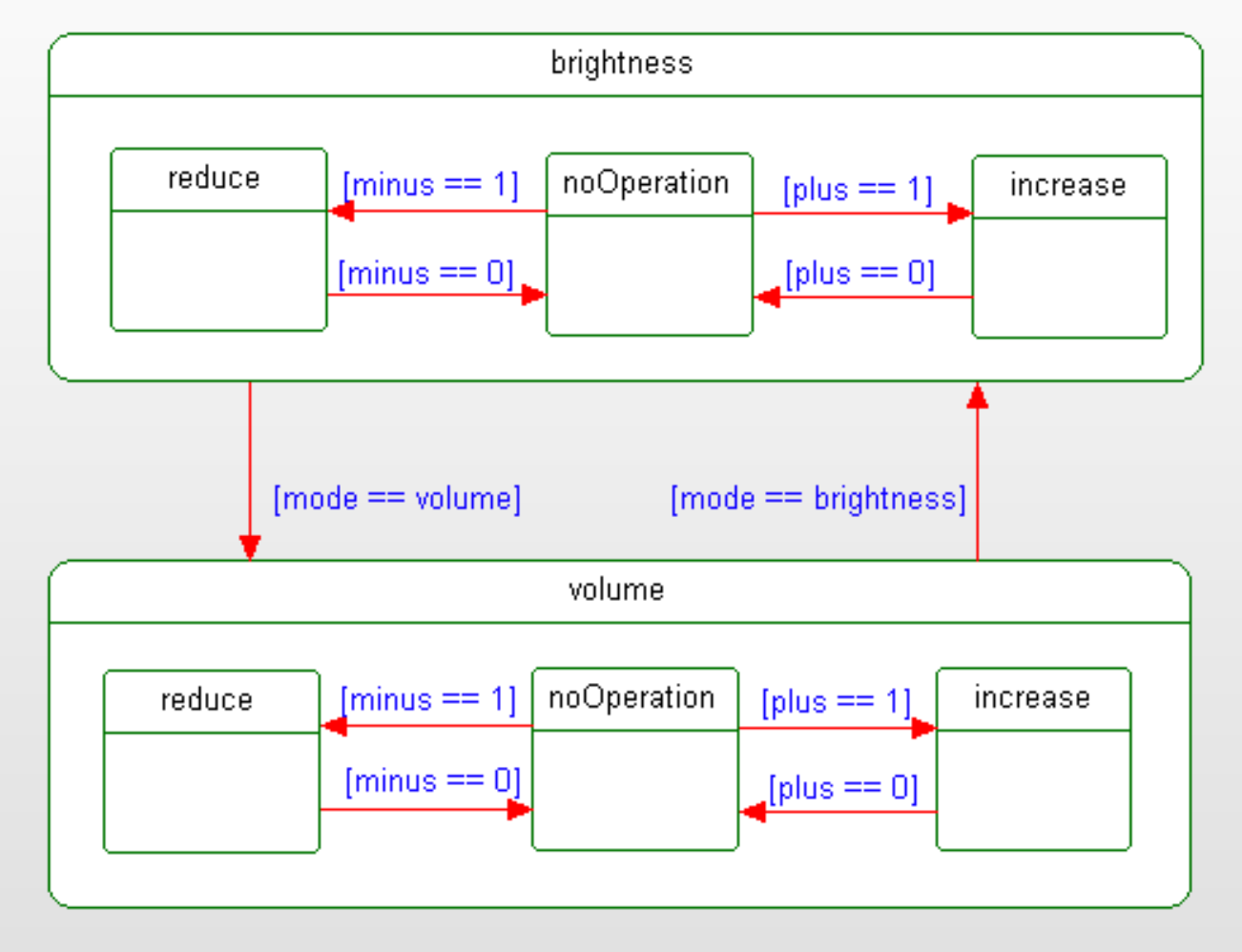


Figure No. 2

Background information regarding modeling based on state machines

Let's take another look at the state machine shown in figure No. 3. Viewing the diagram superficially one can say that all information that is necessary for execution of logic over time is included. But that is not quite true. The need for a queue running in the background which stores the precise order of events, is often overlooked.

The execution over time becomes distinct only with the information regarding the order of events (It must be assumed that the state machine does not possess the sole computing time and more than one change has occurred within one cycle.

Example: The system is in the state brightness, and the user wants to increase the volume, then the order of events is: 1. Mode 2. plus. If the order is changed, brightness will increase. When the buttons are pressed quickly in succession and the state machine is not active in between, the information regarding the order, will be required.

This information is missing in time-driven runtime designs based on signals. Therefore these have to be modeled in the state machine, additionally. The result is that each state has to be checked for current validity before a state change can be implemented within. Ultimately this leads to a flow chart. Modeling on the basis of notation of state machines leads to incomprehensible and unnecessarily complex state machines.

Attention:

There are three design patterns for runtime architecture and dataflow. When it comes to combination problems are inevitable. The same applies to certain combinations of modeling methods and the use of diagramms and architectural patterns.

Here are some combinations which DO NOT harmonize with each other.

Preemptive scheduling and synchronous data flow. (i.e. global variables or signals).

Timedriven runtime designs with asynchronous flow of data (Events, News).

Timedriven runtime designs and modeling in state machines.

Event driven runtime designs and modeling in sequence diagramms.

Background information regarding modeling based on activity diagrams and flow charts with Rhapsody

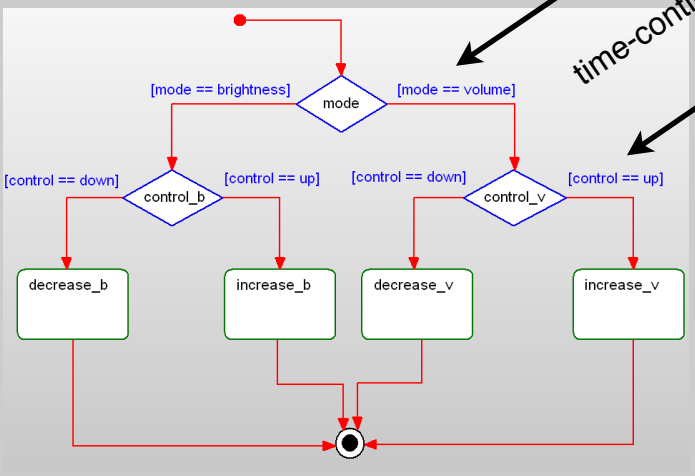
In Rhapsody code is generated from activity diagrams which are similar to state machines.

Rhapsody offers modeling on the basis of flowcharts. Code which is generated from flowcharts corresponds precisely to the requirements we need at this point and thus forms the desired alternative to state machines.

CPU

TIME-ORIENTED WITH CLOCK SPEED

called cyclically



time-continuous signal

Continuously adjacent signals

The condition of the environment is sampled with every pass. The result is slow response times.

```
while (1)
{
        if (Signal_1)
                do();
        else
                don‘t();

        if (Signal_2)
                do();
        else
                don‘t();

        if (Signal_3)
                do();
        else
                don‘t()
}
```

Figure No. 3

Pre-cons on modeling with state machines based on events?

The advantage of state machines is very fast reaction to changes within the system, due to the current state which is known and that only one transition has to take place. A change of state is performed directly after occurrence. Thus the system is executed in small steps which from a temporal point of view leads to more efficiency as with time-driven systems.

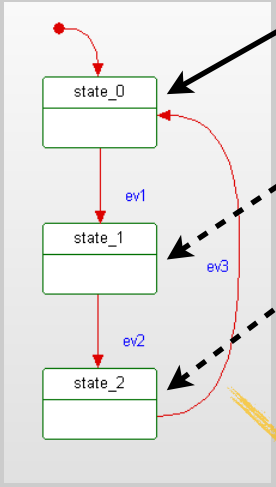
A disadvantage is less robustness to stimuli. When the system generates false events due to electromagnetic smog, the system responds to every event which can lead to an overload. A prerequisite for a robust system based on events is that it is not overloaded with meaningless events.

A time driven system does not react to state changes, but always in the same time intervals, regardless of stimuli. This is also the disadvantage. The system is subject to the sampling theorem. If stimuli with a higher frequency appear, the event driven system reacts correctly und adjusts itself to that frequency automatically, whereas the time driven system remains rigid and becomes inaccurate. Even changes may disappear.

CPU

TIME-ORIENTED WITH CLOCK FREQUENCY

called by event



Δ t

Δ t

EVENTS

THE SYSTEM KEEPS TRACK OF THE STATES OF THE ENVIRONMENT, INHERENTLY, AND CAN RESPOND TO A CHANGE (AN EVENT) IMMEDIATELY.

```
switch (rootState_active) {
      case state_0:
      {
         if(id == ev1_Default_id)
            {
               rootState_active = state_1;
              do();
            }
      }
      break;
      case state_1:
      {
         if(id == ev2_Default_id)
            {
               rootState_active = state_2;
              do();
            }
      }
      break;
      case state_2:
      {
         if(id == ev3_Default_id)
            {
                rootState_active = state_0;
              do();
            }
      }
      break;
}
```

UMLforum.de

Product:

IBM® RATIONAL® RHAPSODY®
START-UP TRAINING

Author:

ANDREAS WILLERT

Editor:

WILLERT SOFTWARE TOOLS GMBH

Hannoversche Straße 21
DE - 31675 Bückeburg
Phone: +49 5722 9678 - 60
info@willert.de
www.willert.de

Premier Business Partner