Design of Real-Time Software

Part 2: Real-time Software Design

Onno van Roosmalen
O.S.vanRoosmalen@chello.nl

Version July 2016

Contents

- Introduction
- Abstraction levels
- Real-time design issues
- Workshop preparation

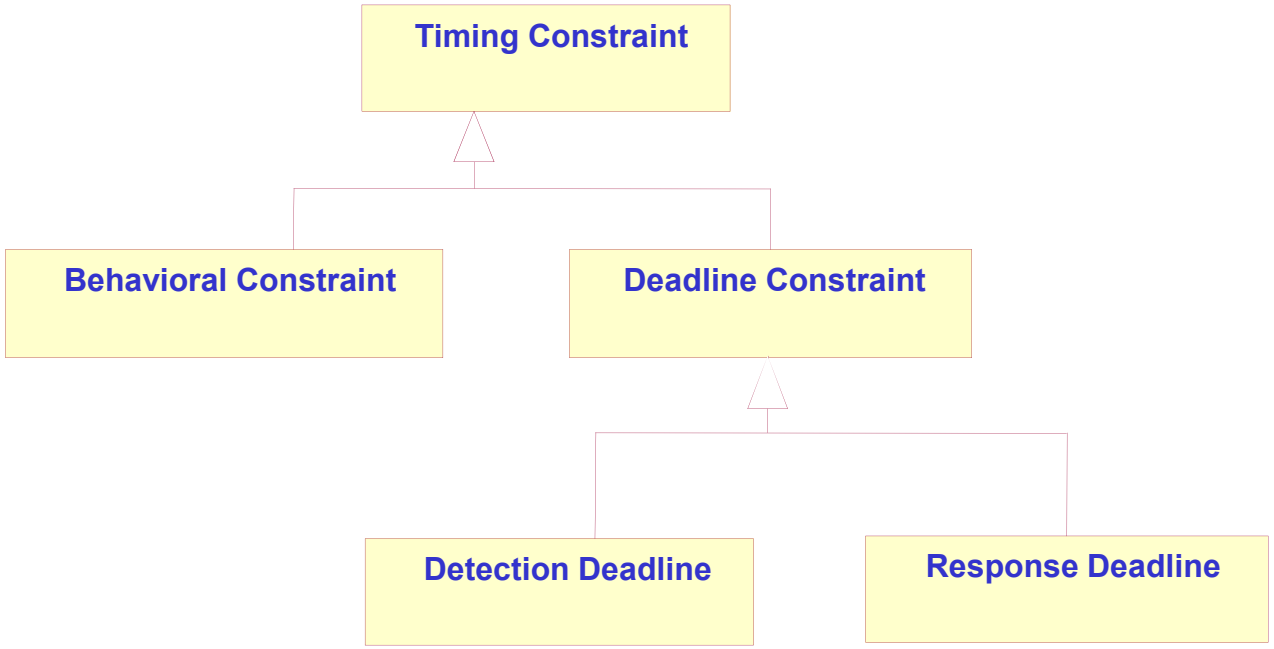
Contents

- Introduction

Timeliness as a requirement

- Two types of timing constraints:
  - Behavioral constraints:
    - Example: start closing the door after 20 seconds
  - Deadline constraints:
    - Switch off the pump within 10 ms.
- Behavioral constraints are functional requirements
  - Their realization can be localized in software implementation
- Deadline constraints are (equivalent to) non-functional requirements
  - Their realization is emergent from all software implementation aspects

Types of timing constraints



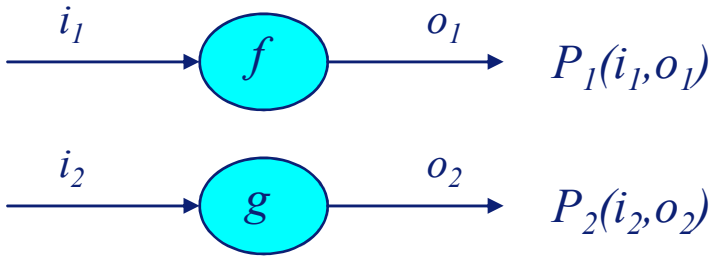
Composability

- A computer system is a layered structure that provides the system developer with convenient abstractions at each boundary
  - This helps to deal with complexity

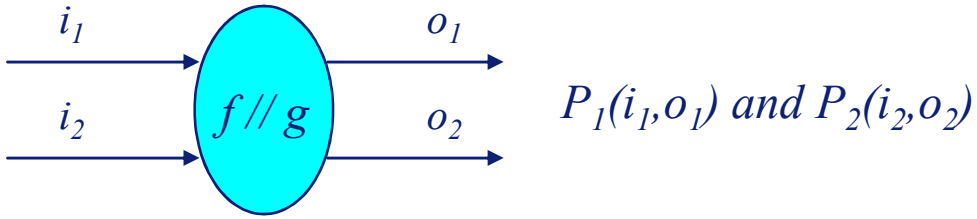
- It is necessary to offer primitives at each boundary that behave in a composable manner
  - This means that primitives can be combined on the basis of their interface specification
  - It is not necessary to consider their implementation

Composability

- Consider two transformations



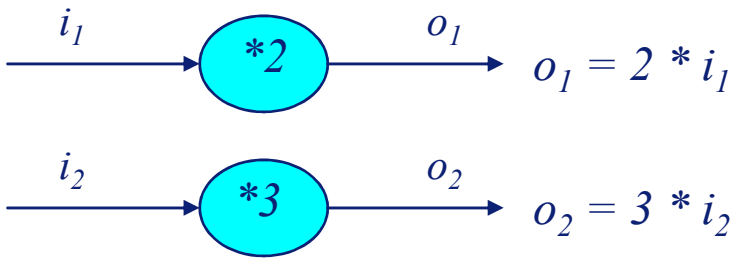
- Composability means that we can combine them



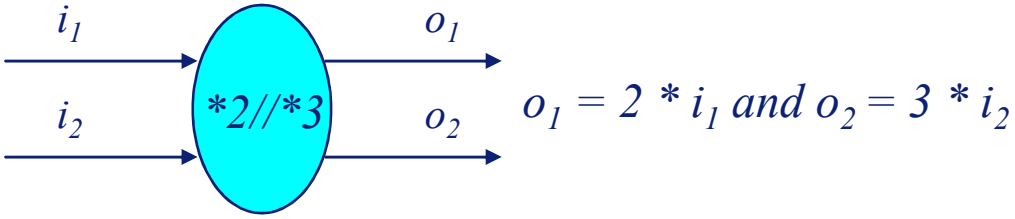
- This should work independently of the “context”, i.e.
  - the platform and
  - presence of other components

Example

- Consider the following two transformations



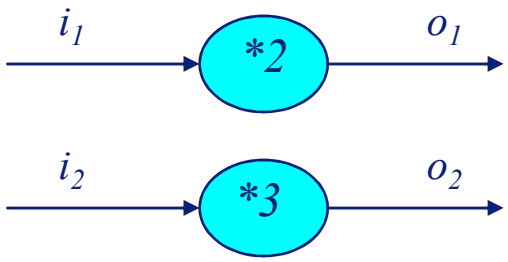
- We can implement them in such a way that they can be combined straightforwardly



- This works independently of the “context”

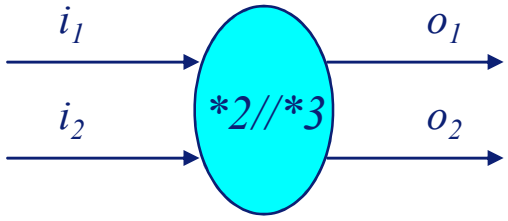
Time is a problem

- When time is part of the specification of *f* and *g*'s behavior, this is not easy to achieve
  - because *f* and *g* may share execution resources.



i_1 at $t_1 \rightarrow (o_1 = 2 * i_1)$ after $(t_1 + \delta)$

i_2 at $t_2 \rightarrow (o_2 = 3 * i_2)$ after $(t_2 + \delta)$



i_1 at $t_1 \rightarrow (o_1 = 2 * i_1)$ after $(t_1 + \delta)$

and

i_2 at $t_2 \rightarrow (o_2 = 3 * i_2)$ after $(t_2 + \delta)$

Essential Problem with Real-time

- Software systems cannot be fully analyzed for their timing properties until sufficient internal properties of ALL constituent components are known
  - This is usually not the case until the end of the system's development
  - Example: RMA
    - We need info on WCET's, resource sharing
    - We can "guesstimate" these beforehand and use them as "budgets" to be checked later.

- This may lead to late redesign cycles if the resulting system is not meeting its timing requirements
  - The expense of redesign increases rapidly with system complexity and becomes quickly prohibitive

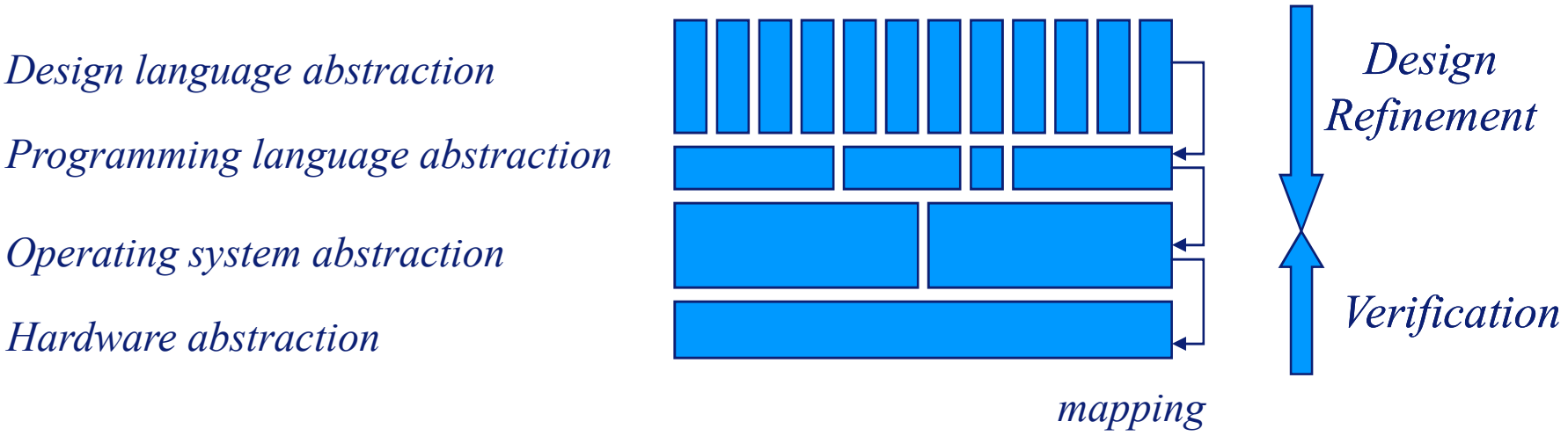
Raising the level of abstraction

- Languages with which we design/implement tend to become more abstract
  - For software development productivity
  - For system complexity
  - For software maintainability

- However, for scheduling and performance analysis we need simple and/or well understood (low-level) machine language mapping
  - For computation of execution times
  - For predictability of task interactions

Layers of Abstraction in Design

- There are different units of concurrency in each abstraction layer
  - More abstract layers have more concurrency
  - Concurrency is the norm rather then the exception
- The mapping must be made explicit, either in the design refinement or the verification process



Raising the Level of Complexity

1) Systems tend to be more complex
2) In each abstraction layer more complexity I,s introduced
   - For performance
   - For more convenient abstractions
   - Example:
     - Hardware: caching, pipelining
     - Operating system: virtual memory
     - Programming language: garbage collection
     - Design Language: classes, objects

- All these factors make analysis more difficult

Solution: use heuristics

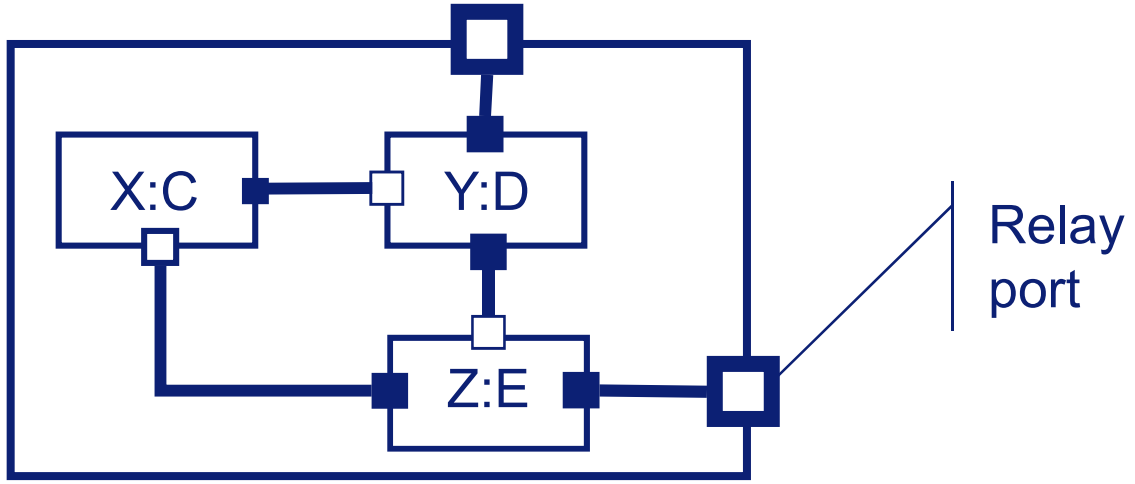
- In RMA we have to analyze the internal properties of the various concurrent components
  - These properties may be context dependent (e.g. execution times, priorities)
- In real-time system design, choices with respect to the concurrent entities and their properties become crucial to enable verification of timing specifications
  - (i.e. the mapping of concurrency primitives of tasks)
- This course is intended to provide insight in the design process and possible heuristics to guide it

Design languages

- There are many design methods, two principle categories
  - Data-flow methods
  - Object-oriented (OO) methods
- There are many (old) data-flow methods, with slight differences in notations and semantics
    - Ward/Mellor; Yourdon; Hatley-Pirbhai; Codarts; ….
    - Some variants called “Structured Analysis/Structured Design”
- Most relevant OO methods now covered by UML 2
  - The only design language with a standard specification
  - Accepted by a large fraction of the software community
  - The most notable RT variants (now covered by UML 2):
    - ROOM; Octopus; Rhapsody; …

Composite Structure Modeling in ROOM

- ROOM and Rhapsody introduced variants of UML 2 composite structures



OO Design Heuristics

- Software objects represent domain objects
  - Software objects keep relevant state of domain objects
    - The state information is obtained through sensors that are at the bottom of an aggregation hierarchy
    - Sensors are I/O objects
    - There always is some latency in the measurement process. This should not lead to too inconsistent a view on the domain objects
  - Their behavior controls the state changes of domain object
    - The control is exercised through actuator objects that are at the bottom of an aggregation hierarchy
    - Actuators are I/O objects
  - A software object provides “intelligence” to the corresponding domain object

UML 2 diagrams

- The distinction between programming language and design language is disappearing
  - Rhapsody and ROOM use “real-time” UML 2 models that have executable semantics (examples of model driven development)
- Platform independence of program/design is achieved through virtual machine layer
  - non-real-time semantics is (often) the same on different VM’s
  - verifying timeliness requires detailed knowledge of VM properties

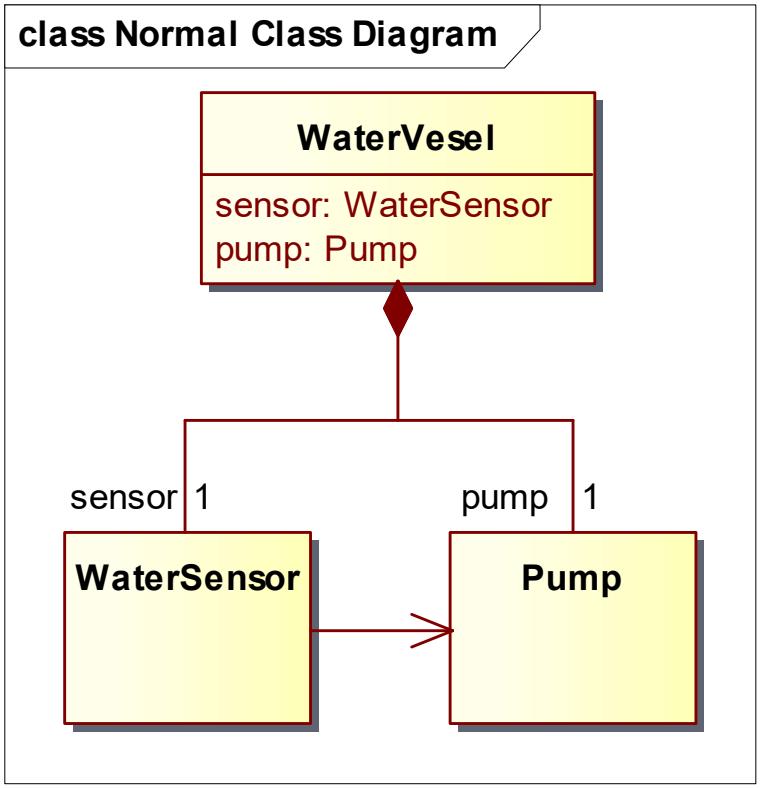
Object Structuring Heuristics

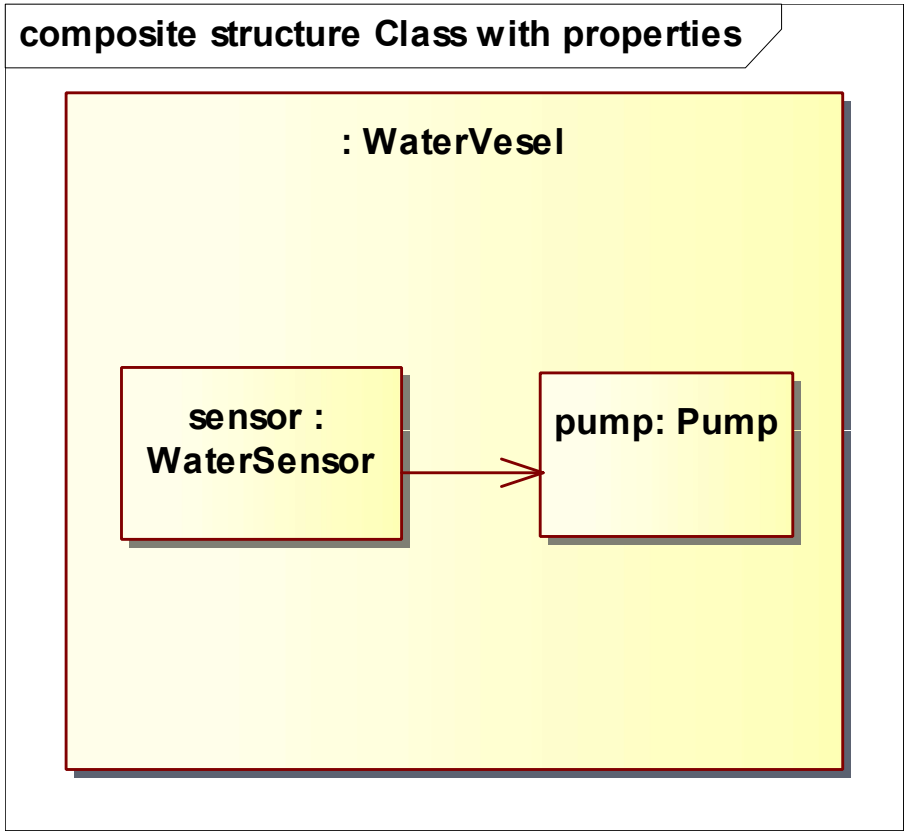
- Model a system as a hierarchical aggregation/ composition
- Every monitored and/or controlled domain object appears in the model
  - I/O Object heuristic
  - Usually leaf objects in an aggregation/composition
- Objects encapsulate their status/values relevant to the control problem
  - object capture both monitoring and control aspects

Rationale for UML Composite Structure modeling

- The process view of architecture is linked to objects rather than classes
  - Number of active objects determines number of threads
  - Resources are instances: objects , not classes, are accessed under mutual exclusion.
- Consequence:
  - Analyzing responsiveness, schedulability, performance requires diagrams that express the sharing of object.
  - Composite Structure diagrams exploit the static relationships between objects (composite aggregation)

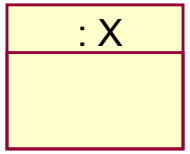
Composite Aggregation





Design Language (UML Composite Structures)

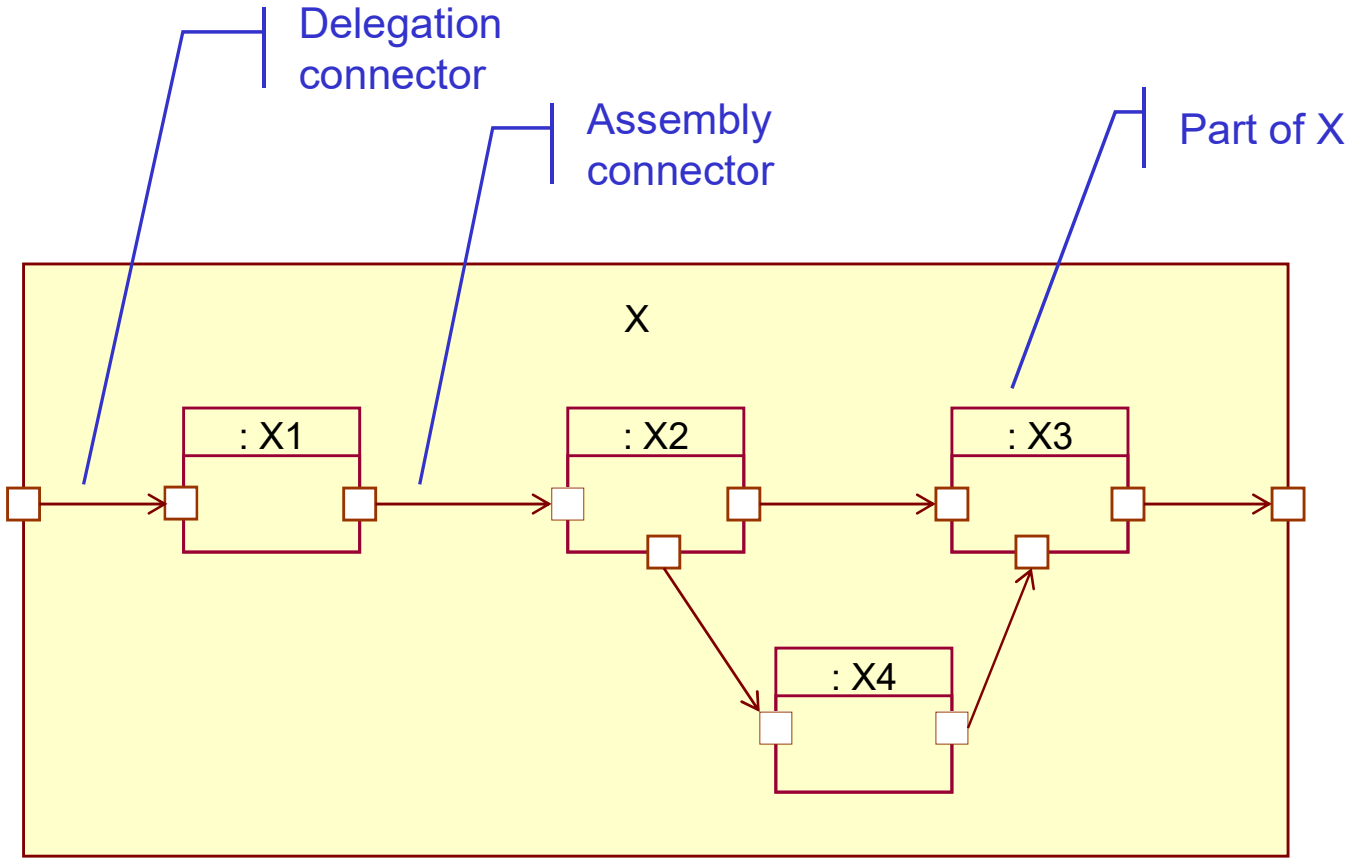
- Part classified as X



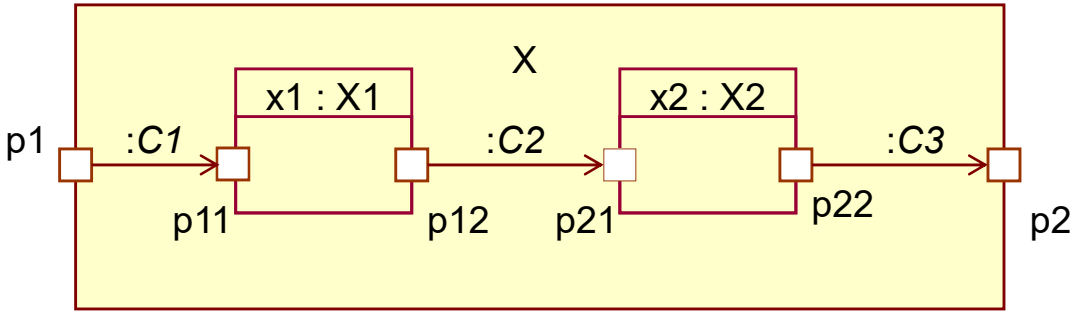
- Port p classified as PortX

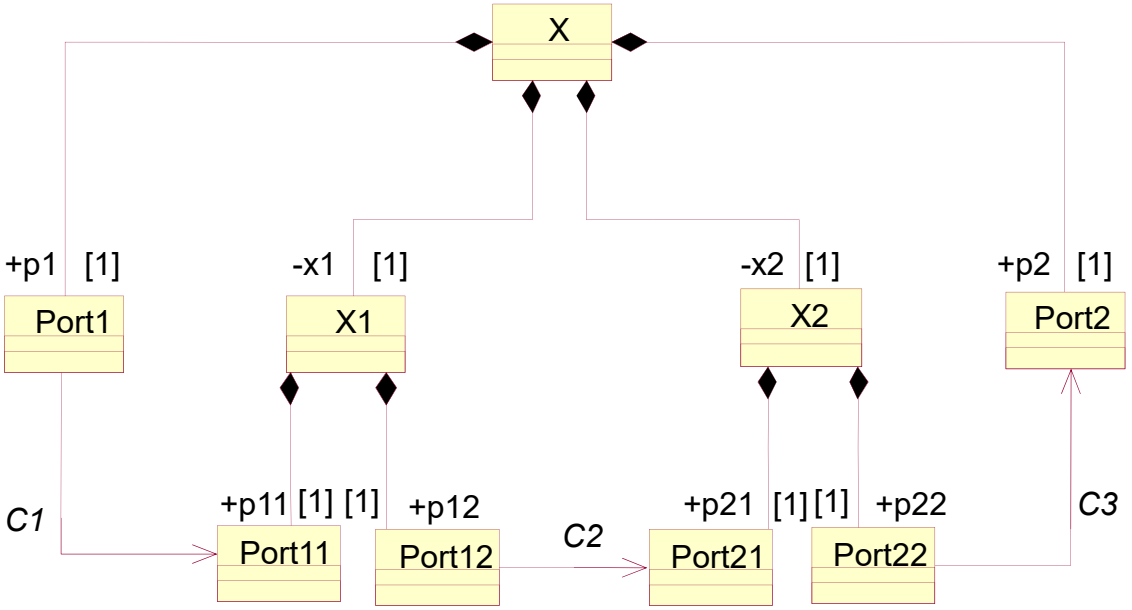
- Connector of type *C*

Modeling Composite Structures



Relation with standard structure diagram in UML



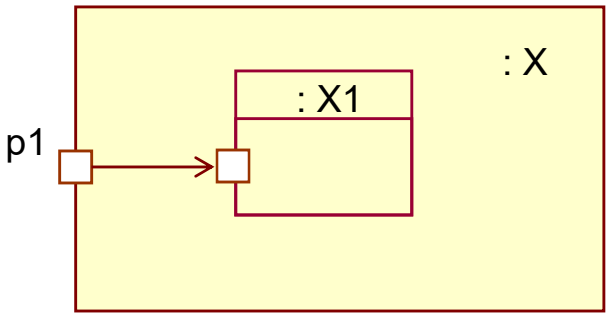


Purpose of Ports

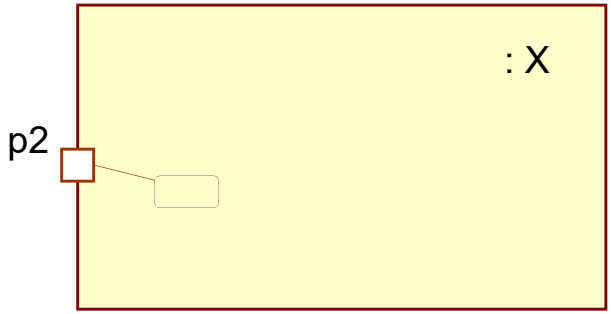
- A port is a static adapter (as in the “Adapter” design pattern)
  - To adapt the communication signature and mechanism on the inside from the outside.
  - It decouples the communicating partners (they only depend on the port types).
  - Ports represent the interfaces of the parts.
  - Ports implement the interfaces by delegating to the proper behavior of their part
  - Parts are substitutable if they have the same ports (interfaces: signature and specification)

Delegation of communication

- Delegation to a part :X1 of the part :X (using a delegation connector)



- Delegation directly to the part :X's behavior (called a behavior port)



Communication between Ports

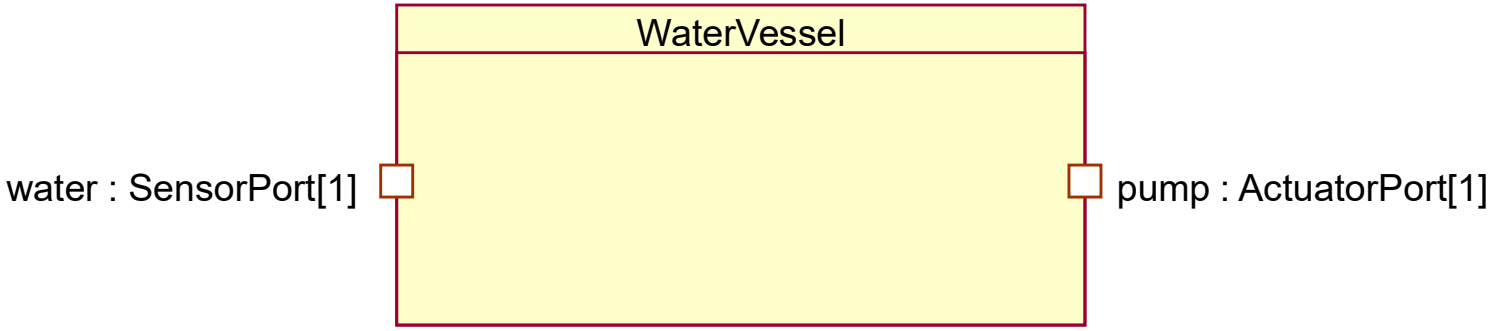
- Call-return
  - Koala
- Buffered
  - Infinite buffers (asynchronous): Room, Rhapsody
  - No buffers (rendez-vous, synchronous) : POOSL
  - When a part uses buffered message communication, it must be active

Water Vessel Example

- The earlier mentioned example of the water vessel
- A single polled detector yields the height of the water
- Design a control program

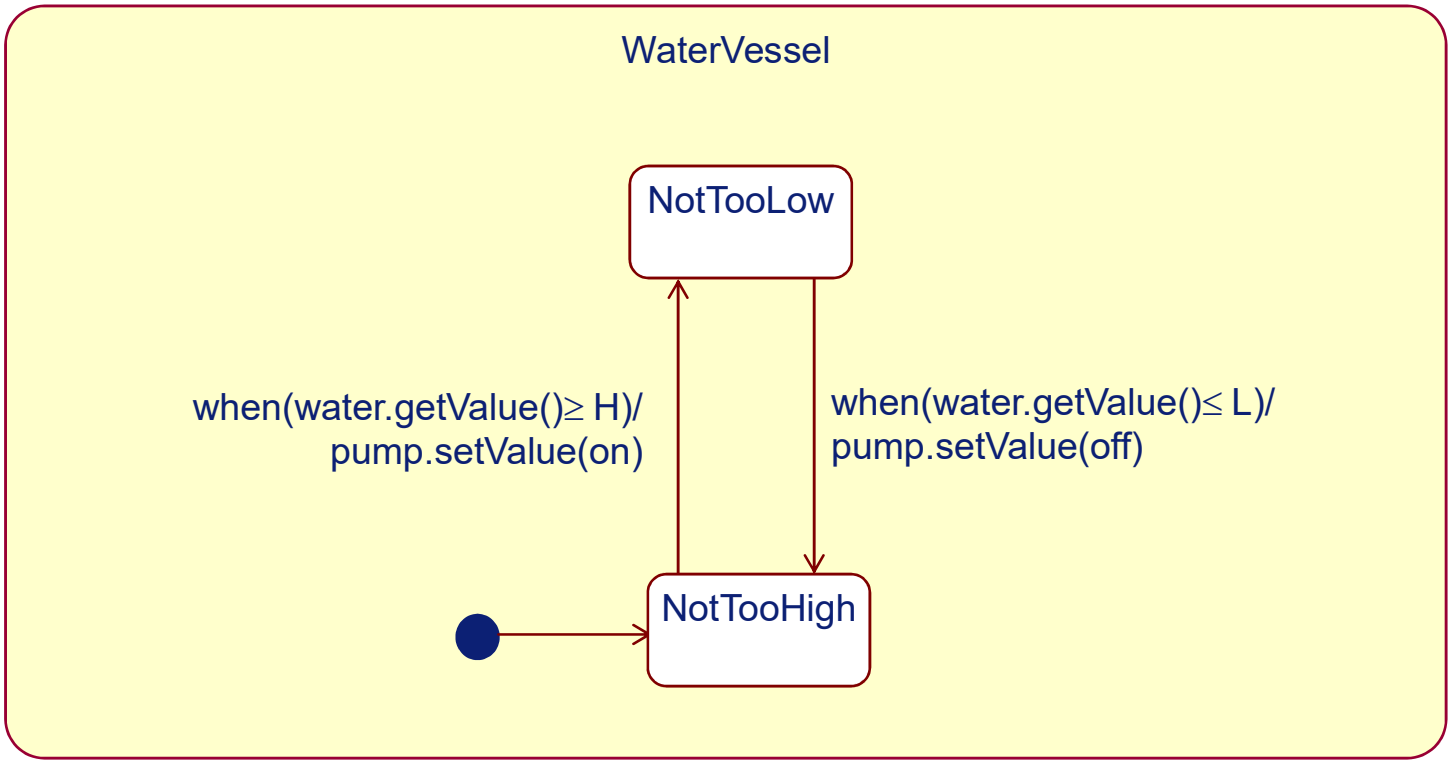
Design

- Static structure (Interface)



Design

- Behavior



Language Mapping

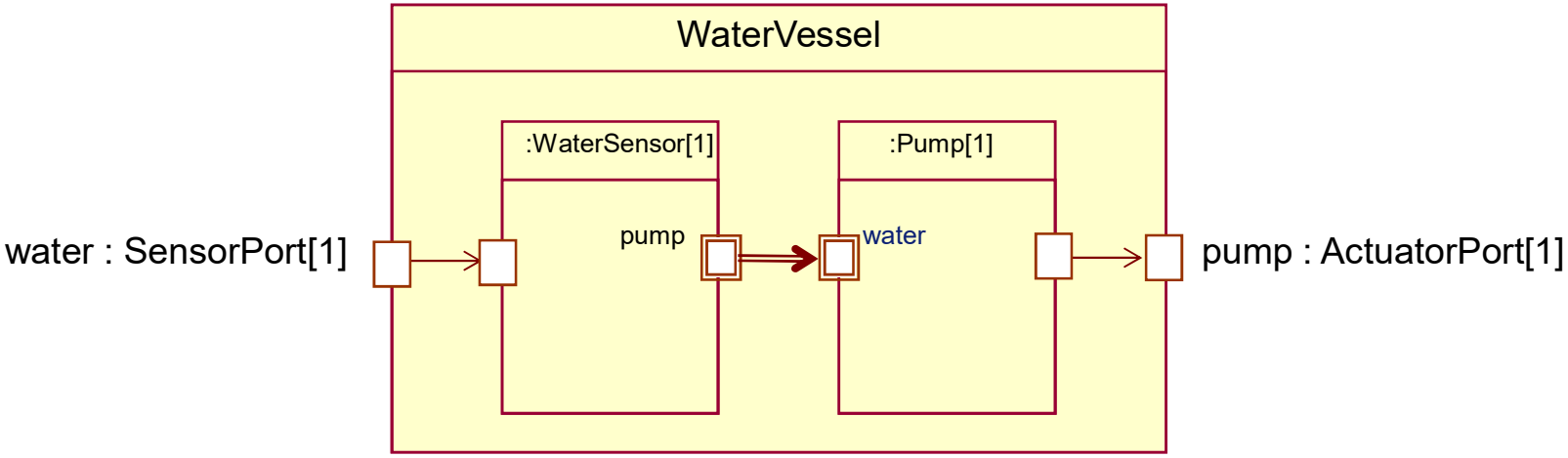
```
class WaterVessel extends Thread {
        private SensorPort water;
        private ActuatorPort pump;
        private State state;

        public void run() {
                state = NotTooHigh;
                while (true) {
                        Value x = water.getValue();
                        if ((x >= H) && (state==NotTooHigh)) {
                                state = NotTooLow;
                                pump.setValue(on);
                        } else if ((x <= L) && (state==NotTooLow)) {
                                state = NotTooHigh;
                                pump.setValue(off);
                        }

                 }
        }
}
```

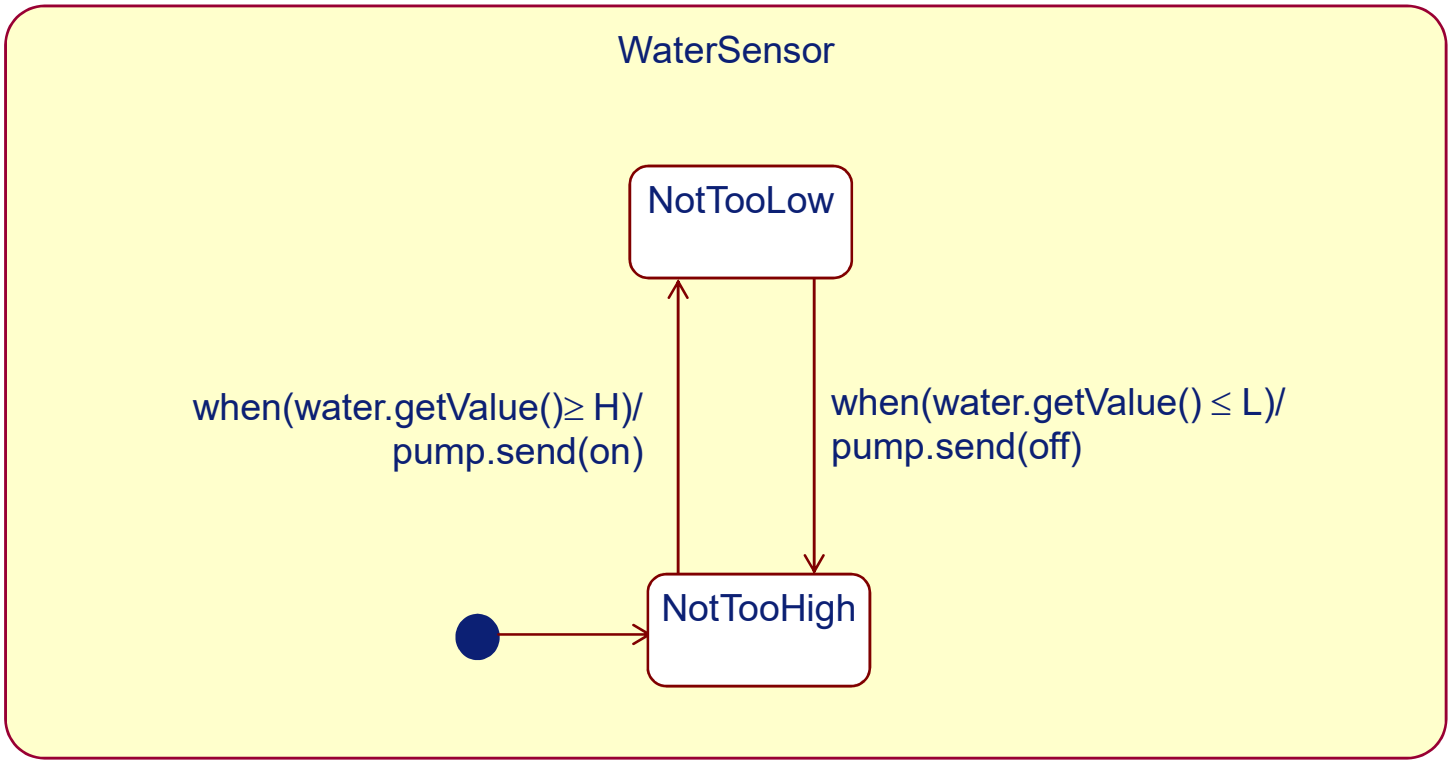
Design

- Structure using I/O object heuristic



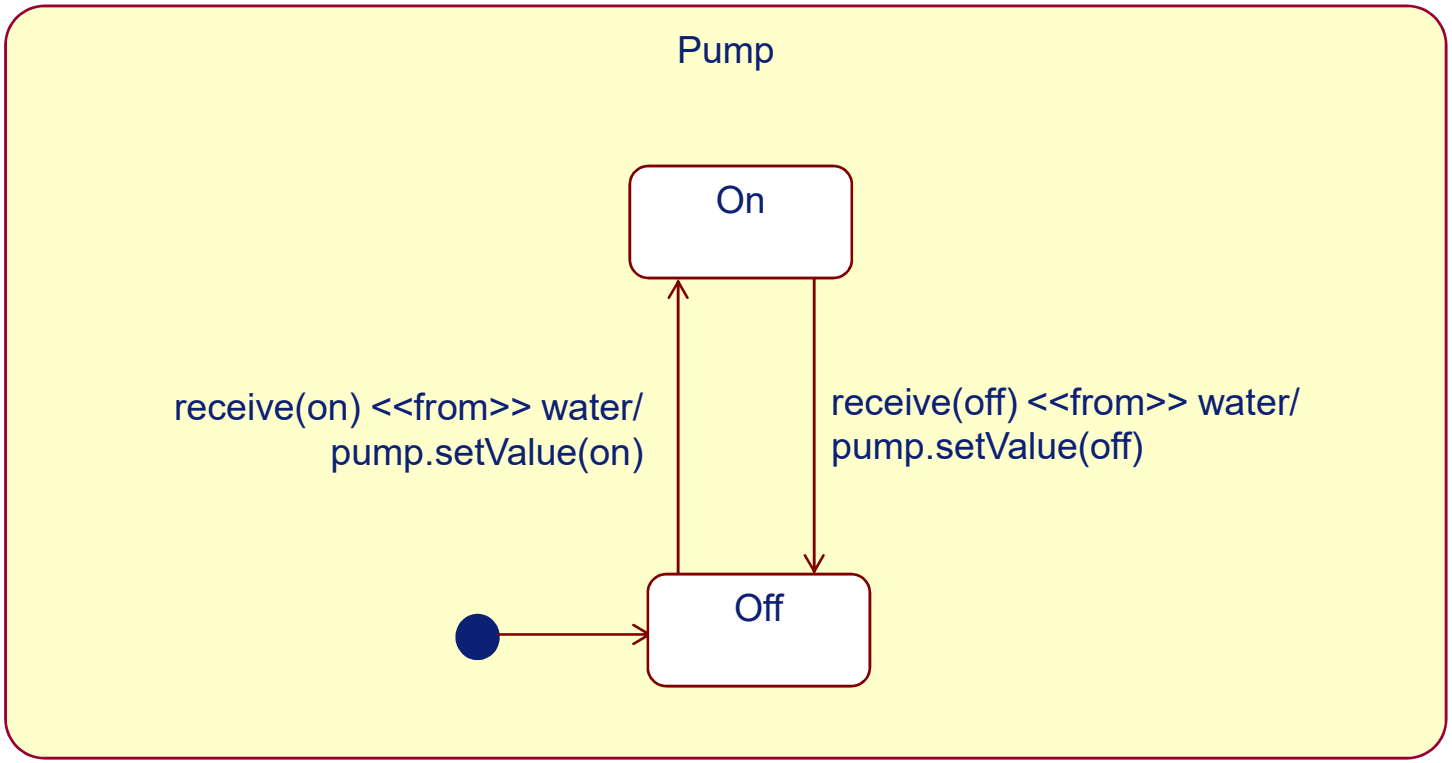
Design

- Behavior of WaterSensor



Design

- Behavior of Pump



Language Mapping for WaterSensor

```
class WaterSensor extends Thread {
        private SensorPort water;
        private SendPort pump;
        private State state;

        public void run() {
                state = NotTooHigh;
                while (true) {
                        Value x = water.getValue();
                        if ((x >= H) && (state==NotTooHigh)) {
                                state = NotTooLow;
                                pump.send(on);
                        } else if ((x <= L) && (state==NotTooLow)) {
                                state = NotTooHigh;
                                pump.send(off);
                        }

                 }
        }
}
```

Language Mapping for Pump

```
class Pump extends Thread {
        private ReceivePort water;
        private ActuatorPort pump;
        private State state;

        public void run() {
                state = Off;
                while (true) {
                        x=water.receive()
                        if ((x ==off) && (state==On)) {
                                state = Off;
                                pump.setValue(off)
                        } else if ((x ==on) && (state==Off)) {
                                state = On;
                                pump.setValue(on);
                        }

                 }
        }
}
```

Action Sequencing Rule (using semaphores)

Given: - collection of tasks executing actions *A, B*
- a *synchronization condition (requirement)*

$$\#A \leq \#B + i$$

for non-negative constant *i* .

Solution: introduce a semaphore *s*, $s_0 = i$ and make replacements:

$$A \rightarrow P(s); A$$
$$B \rightarrow B; V(s)$$

Action Sequencing Rule (using messages)

Given: - collection of tasks executing actions *A, B*
- a *sequencing condition (requirement)*

$$\#A \leq \#B + i$$

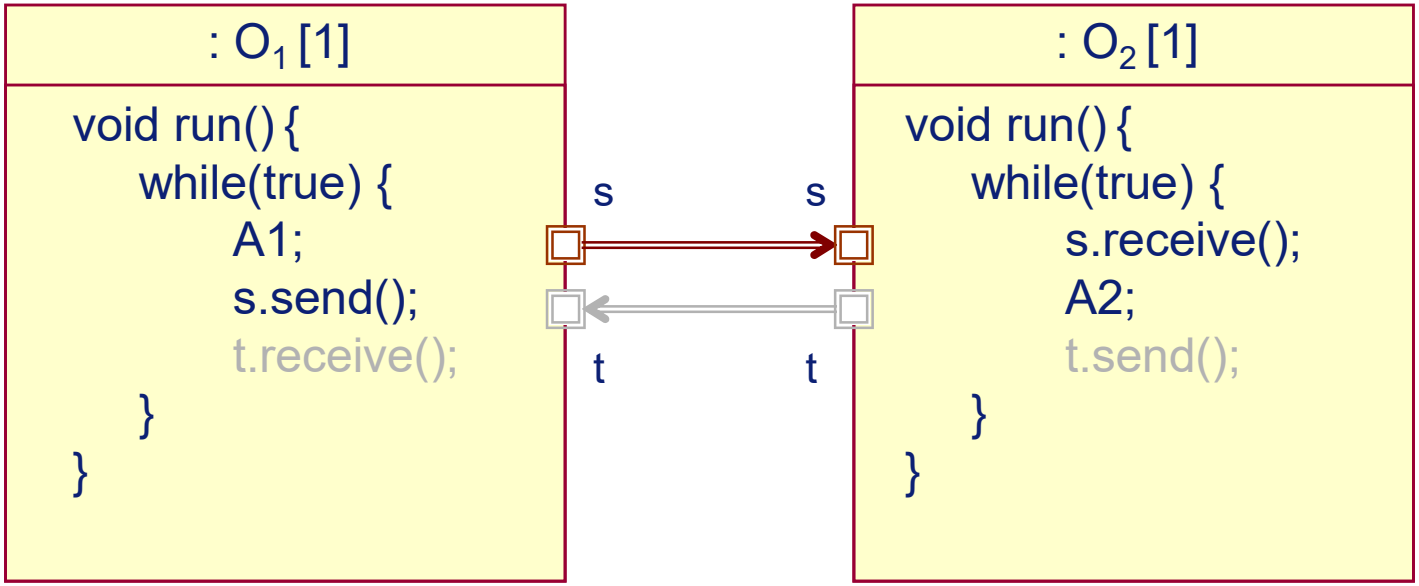
for non-negative constant *i* .

Solution: introduce a infinitely buffered channel *c*, $c_0 = i$ and make replacements:

$$A \rightarrow \textit{receive(c,-); A}$$
$$B \rightarrow \textit{B; send(c,-)}$$

Action Sequencing

- Splitting the water vessel into two parts is a form of action synchronization.
- We don’t need to synchronize back (why?)



Thread Re-factoring Heuristics

- Behavior diagrams (e.g. behavioral state machines) will imply some “natural” concurrency
  - Through their implementation language mapping.
- During design the diagrams can be transformed to re-factor the concurrent thread using so called cohesion arguments
  - Temporal cohesion
  - Sequential cohesion
  - Control cohesion
  - Functional cohesion
- The cohesion principles employ our action synchronization principle discussed earlier
- This re-factoring can reduce or increase the number of threads

Threading Cohesion Arguments

- Temporal cohesion:
  - If two or more threads have the same period or are released by the same event, they can be merged.
- Sequential cohesion:
  - If two or more threads are synchronized such that they carry out their actions in sequential fashion these threads can be merged
- Functional cohesion:
  - If two or more threads are synchronized such that they carry out their actions in a mutually exclusive fashion these threads can be merged

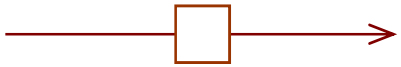
Thread inversion

- All threads of an application could in principle be mapped to one (this is what a scheduler usually does, not the concurrency designer)
- This is extreme, but sometimes overhead due to an abundance of threads must be reduced
- The process of systematically reducing the number of threads using cohesion principles is called thread inversion
  - Multiple instance thread inversion (example: buttons in lift control)
  - Sequential thread inversion (example: sequences of data transformations must be applied one at the time and in fixed order, e.g. a repetitive fetch, decode, execute cycle)
  - Temporal thread inversion

Real-time System Types

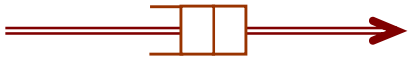
- End-to-end timing constraints
  - Example: Water Vessel
  - Appropriate model: use channels with blocking receive
  - A single task is carried by multiple active objects / multiple threads
- Local timing constraints
  - Example: apparatus with remote control
  - The remote control protocol is time constraint
  - The end-to-end behavior is not constrained
  - Each object is a single task.
  - Communicate between objects blocking free.

Inter-object Communication Options



- Shared Variables

- Buffered communication with active put() and get()



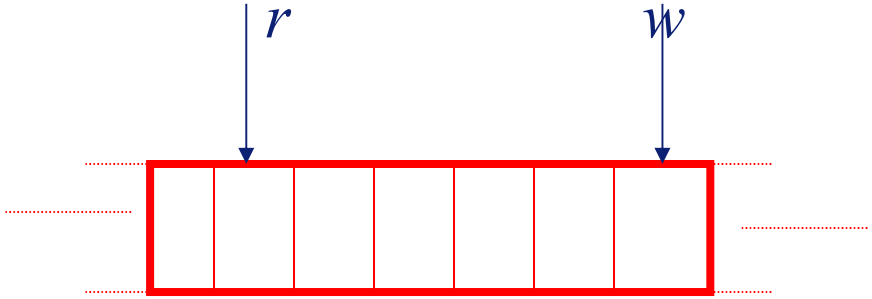
- Buffered communication with active send() and blocking receive()

Non-Blocking Communication

- Shared variables:
  - Safe when only one process writes the variable
  - Assume read and write are atomic actions.
- Buffered communication with active put() and get()
  - Safe (use e.g. a circular array implementation, see next slide)
  - Buffer size is adjusted according to needs of application

Using arrays

Consider an infinite array as an implementation of a queue. Variables *r* and *w* denote read- and write positions respectively (initially 0).




```
Type Buffer {
        Element[] queue;
        int r, w;
        void put(Element element);
        Element get();
        boolean isEmpty();
        void flush();
}
```

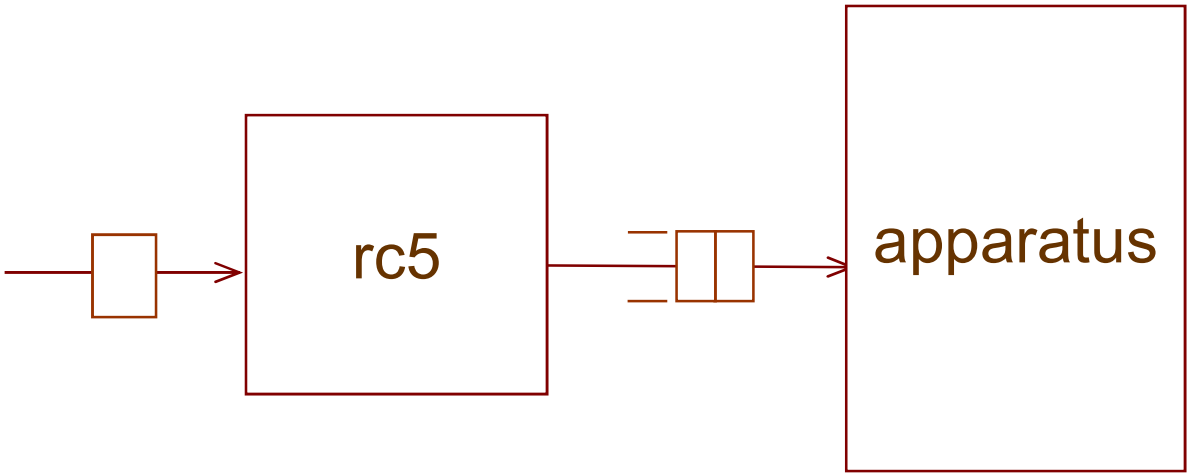
No Blocking

```
void put(Element element) {
        queue[w] = element;
        w = w +1;
}
Element get() {
        result = queue[r];
        r = r +1;
        return result;
}
boolean isEmpty() {
        return r==w;
}
void flush() {
        r=w;
}
```

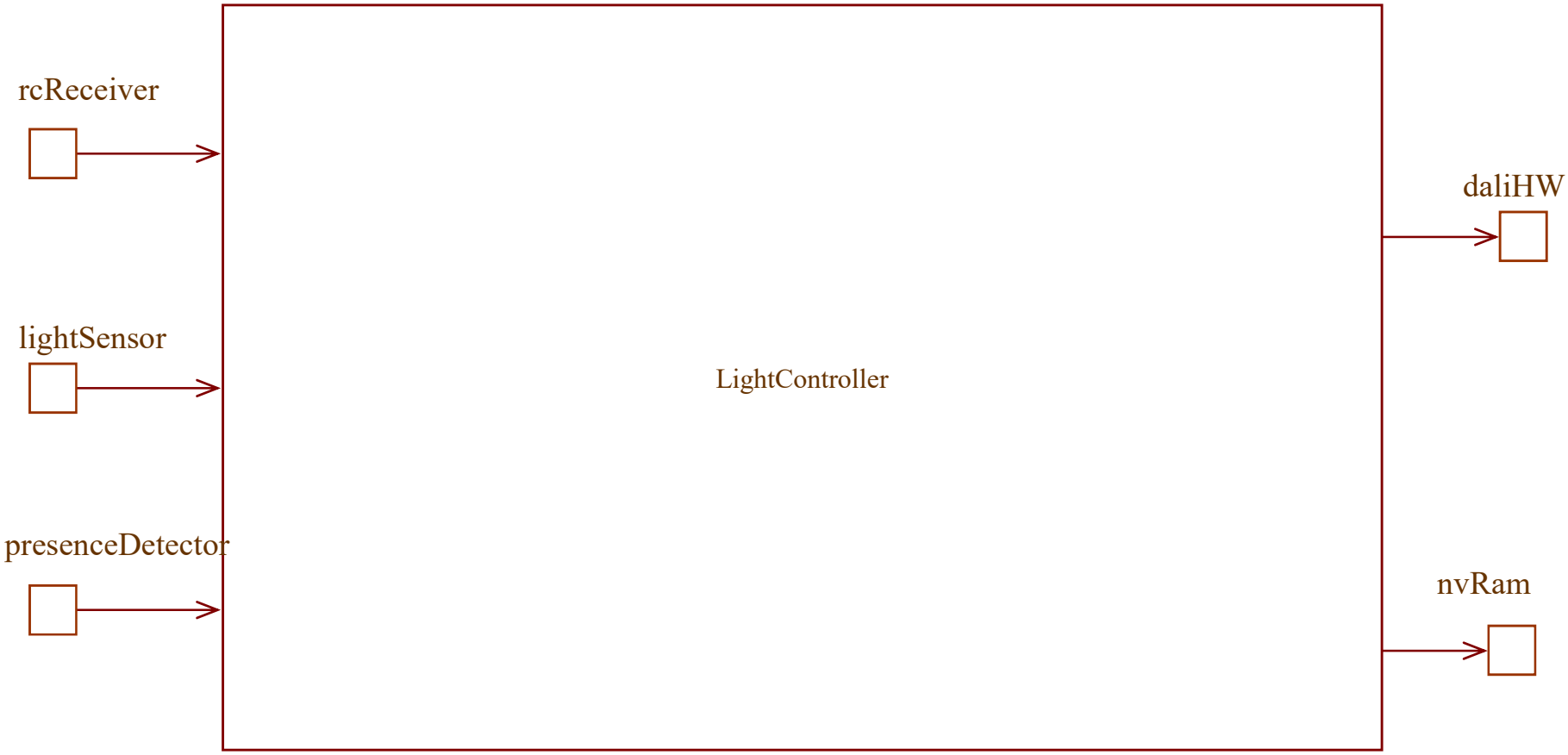
The array is never accessed by both producer and consumer at the same array element simultaneously.

Example Communication

- The rc5 object polls the input variable.
- For reliable interpretation of the bit stream, timing must be kept.
- The rc5 object delivers a stream of commands to the apparatus (non-lossy).



Light Controller



Functional Description

The Light Controller, connected to DALI ballasts, can switch lamps on or off and regulates the light level in a small room. When a person enters the room, the presence detector signals the controller and the lights should be switched on. If after 15 minutes no presence is detected, the lights are switched off.

With the remote control, that uses the RC5 protocol, the light level can be dimmed when the lamp has been switched on due to presence signal.The light sensor measures the light level and gives this information to the controller which determines the new DALI output value. A provided function *calculate(lightLevel, rcCommand)* delivers the appropriate dali command, given a running average of the light measurement (called lightLevel) and the command from the remote control (called rcCommand). Every command passed to the dali is also stored in NVRAM to be able to retrieve the last given command if necessary.

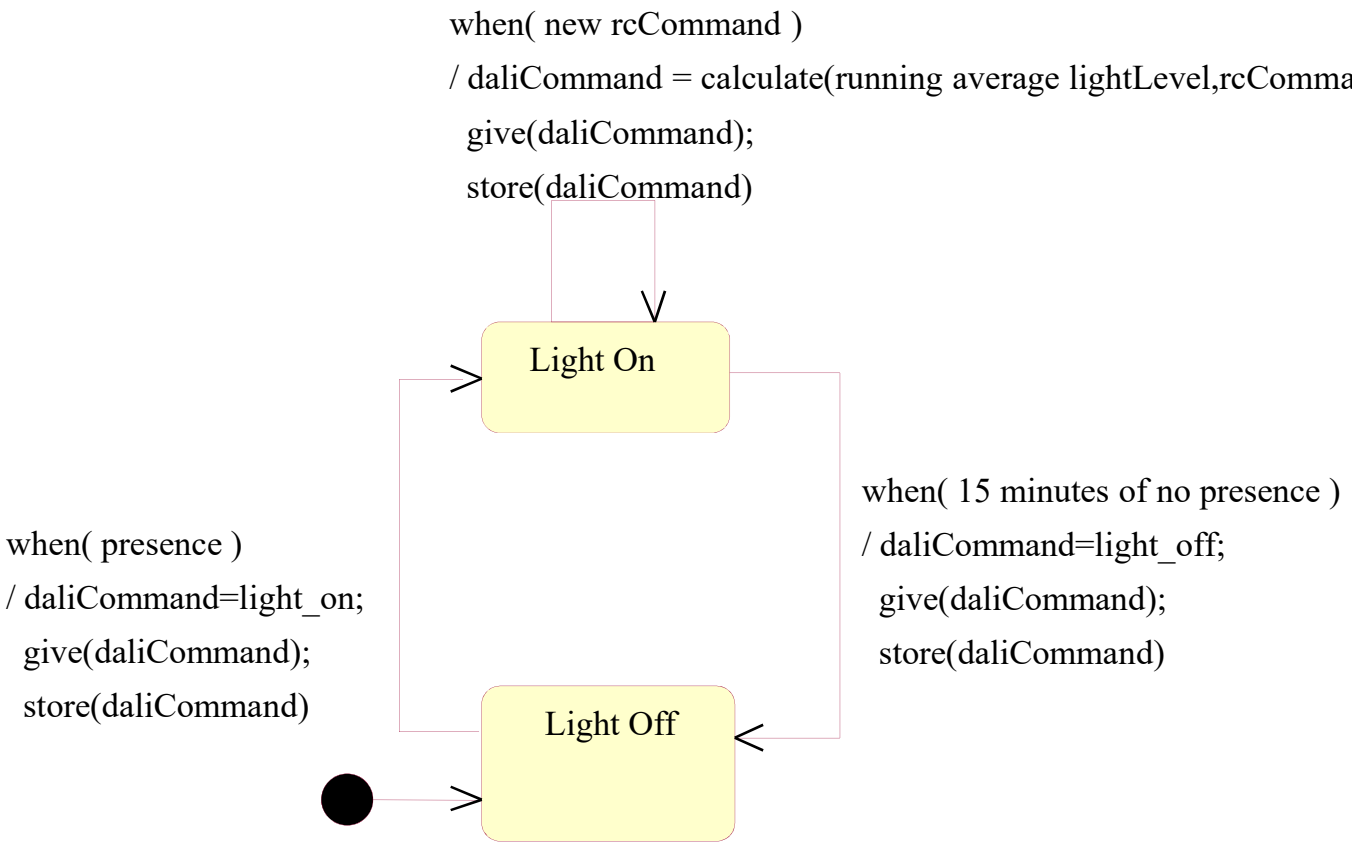
Assignment 1

- Make a normal class diagram for the light controller
  - Use the I/O Object heuristic.
- Draw the corresponding Composite Structure diagram.
  - What connectors would you introduce?
  - Do you have all the parts of the controller?

Step 2

- Give a behavior specification for the light controller as a whole in the form of a state machine diagram.
  - What are the events to respond to?
  - What are the responses required?
  - How do they depend on the state of the room and the instructions given by the remote control
  - Use the same kinds of abstractions as employed in the textual description

Behavior Abstract State Machine



Presence Detection

- The presence detector is polled.
- When no presence is detected for 15 minutes the lights should be switched off.
- During presence (and before the lights are switched off) the lights should be adjusted according to the remote control commands.

Step 3

- Make a model for timing the presence.
  - If nobody has been present for more than 15 minutes the result is true otherwise false.
  - There is a tick event from the system clock.
- Make a state machine diagram
- Sketch an implementation

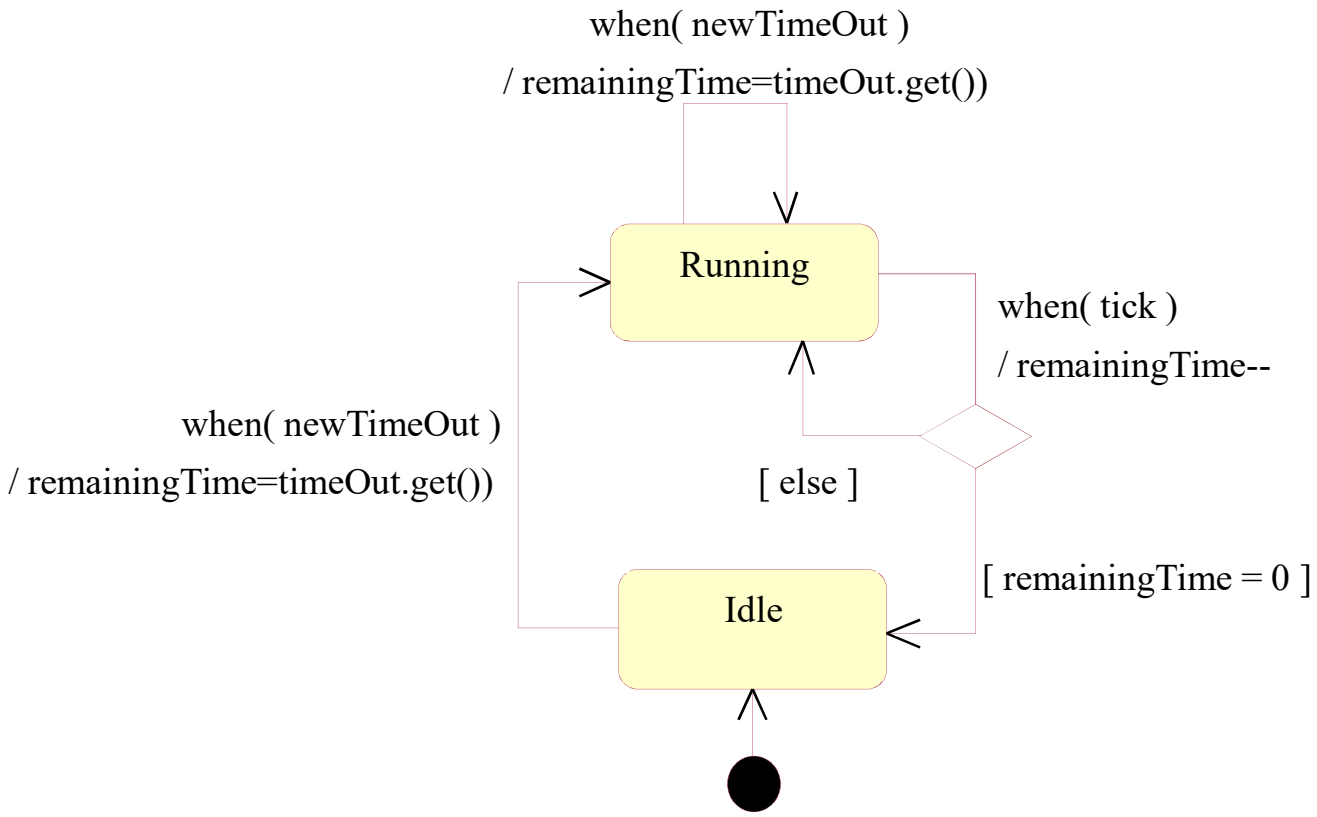
Timer Diagram

- Structure Diagram

  - The pd sends a message with the time to count down to the timer via the newTimeOut buffer.
  - When the timer process sees the message it will load the new time in the shared variable remainingTime and starts counting down.
  - When remainingTime is zero it will remain zero.
  - The timer-user process can inspect the remainingTime to see if time has expired.

Timer Diagram

- State Machine Diagram



Timer Process

P_{Timer} (T)::=

```
while(waitForNextPeriod(T)) {
    if(!timeOut.isEmpty()) {
        remainingTime=timeOut.get()
    }
    if(remainingTime != 0) {
        remainingTime=remainingTime-1;
    }
}
```

Presence Detection Process

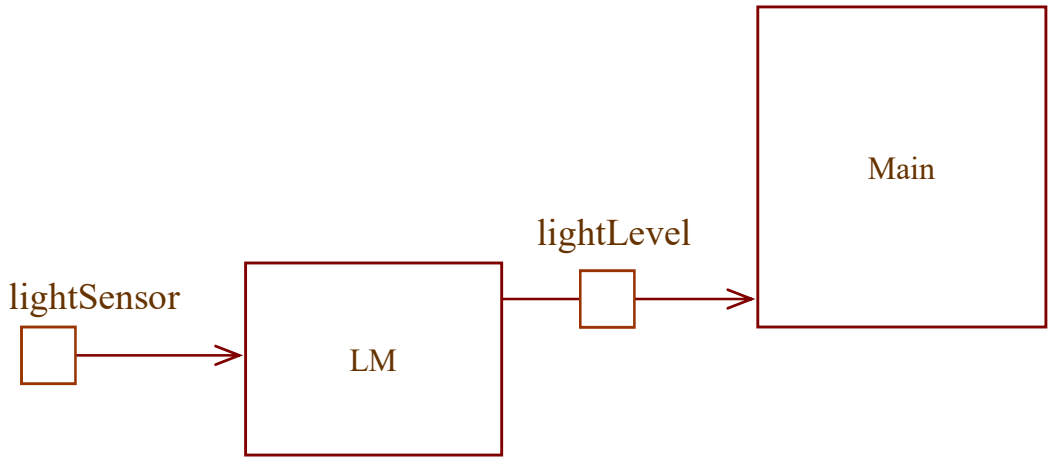
```
P_PD (time , T)::=
        while(waitForNextPeriod(T)) {
            presence = presenceDetector
            if (presence) {
                    newTimeOut.put(time)
            }
        }
```

- The process resets the timer when the detector observes presence.

Step 4

- Sketch an implementation for the light measurement object

Process Design: Starting Point



Light Measurement Process

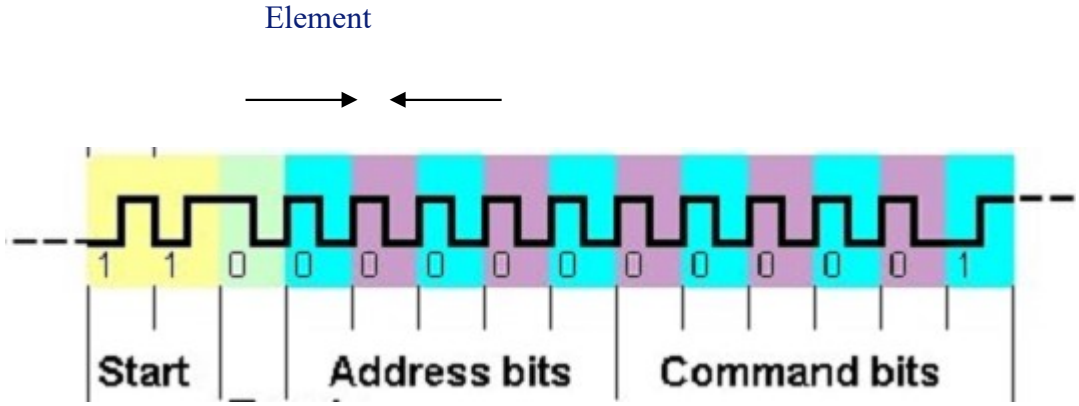
P_{LM} (n, T)::=

```
while(waitForNextPeriod(T)) {
    i=(i+1) mod n;
    result = result - value[i];
    value[i] = lightSensor;
    result = result + value[i];
    lightLevel = result;
}
```

RC5 Protocol

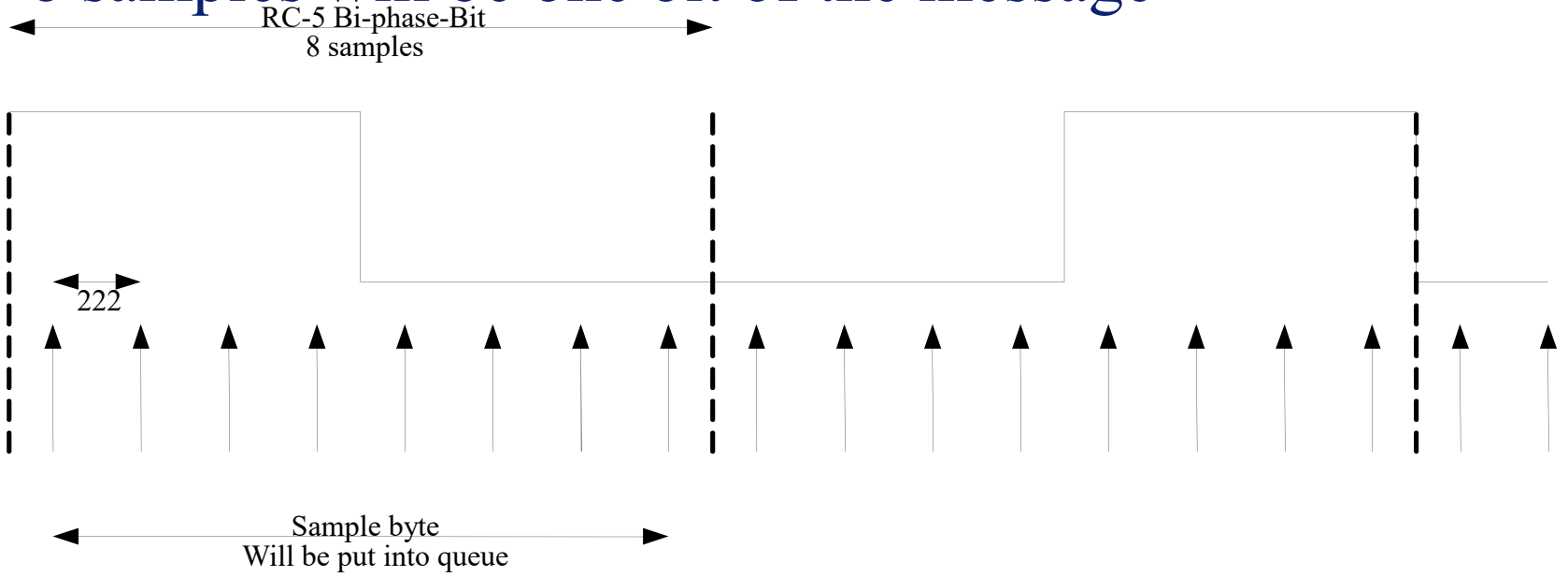
- RC5 protocol
  - Start/stop and command bits are passed as two elements
- Elements are passed by setting the line high/low
- Although elements are 888μs, the elements require a sampling period of 222 μs.
- A remote control command is received in 25ms
- Remote control commands are separated by 89 ms (a repetition time of 114ms).

RC Receiver Signal



RC-5 Polling

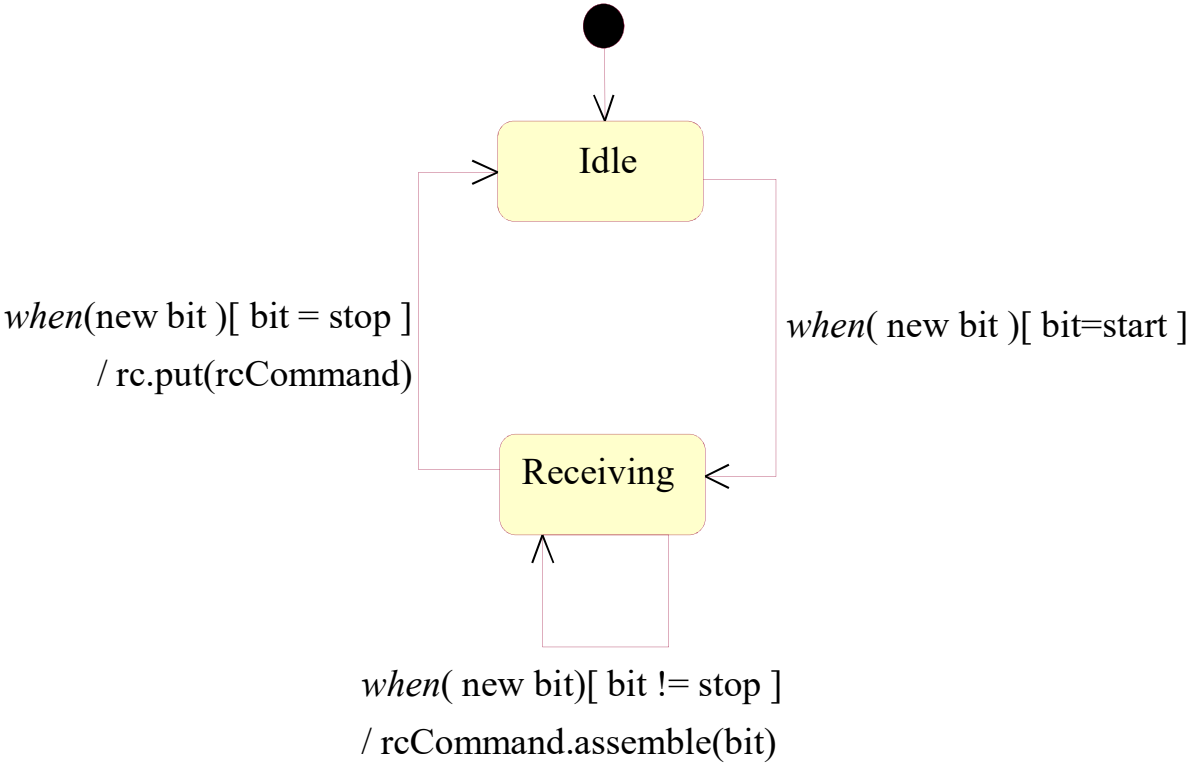
- Every 222us
- Shift one sample into a sample byte
- Every 8 samples post sample byte into queue
- 8 samples will be one bit of the message



Step 5

- Make a state machine diagram for the RC5 object
  - Make the following abstraction:
    - On polling bit may be received (event is *when*(new bit)).
    - The first bit must be a start-bit.
    - Until a stop bit is received, bits are assembled into a rcCommand (use rcCommand.assemble(bit)).
- Sketch an implementation

Abstract State Machine for RC5 Process



Remote Control Process

P_{RC5} (T)::=

```
state = Idle;
while(waitForNextPeriod(T)) {
    bit = rcReceiver
     if(state == Idle) {
            if(bit == start) {
                state = Receiving;
     } else if(state == Receiving) {
            if(element == stop) {
                state = Idle;
                rc.put(rcCommand)
            } else {
                rcCommand.assemble(bit);
            }
     }
}
```

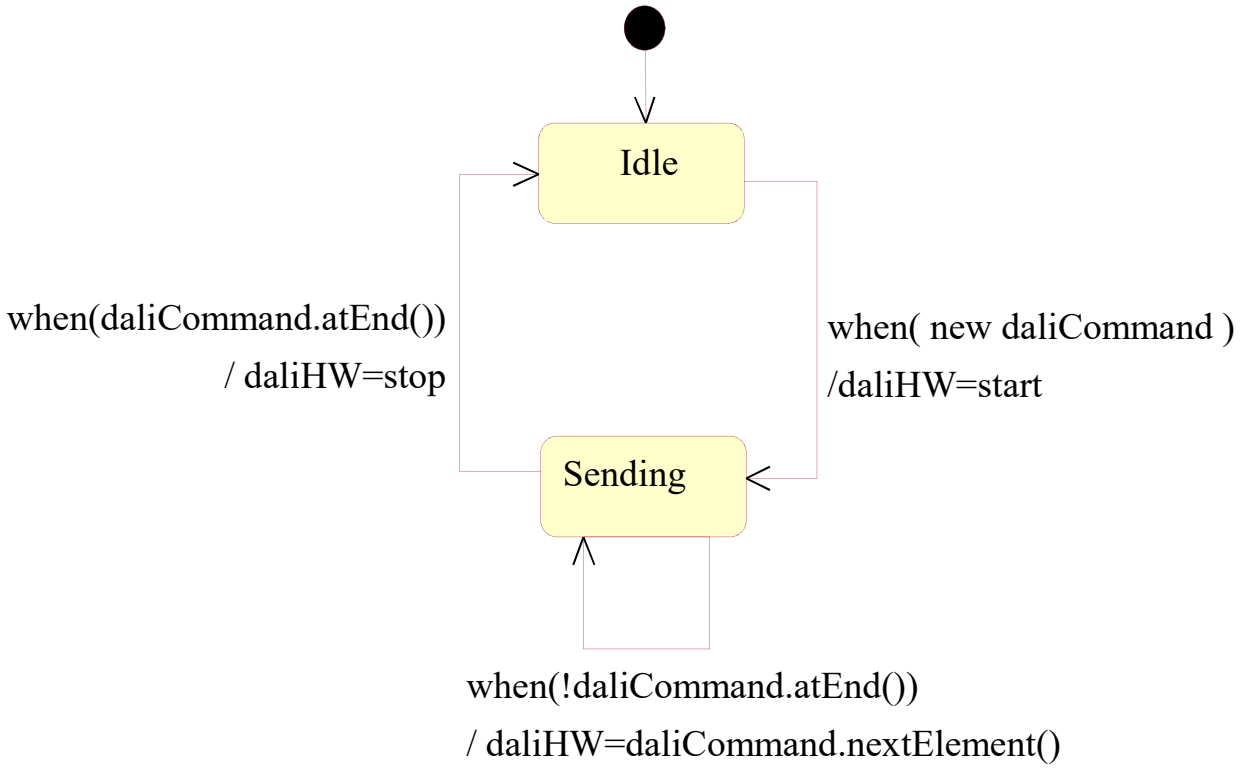
Dali Protocol

- The Dali protocol is very similar to the RC5 protocol
  - Start/stop and command bits are passed
- Elements are passed by setting the output high/low
- The moments of setting high and low outputs must be within a 10% tolerance of the periodicity of the signal to be understood by the receiving hardware.
- The period of the generated elements is 416 μs
- The tolerance is therefore 41 μs
- The distance between provided dali commands must be at least 25 ms

Step 6

- Make a state machine diagram for the dali object
- Sketch an implementation

Dali State Machine



Dali Implementation

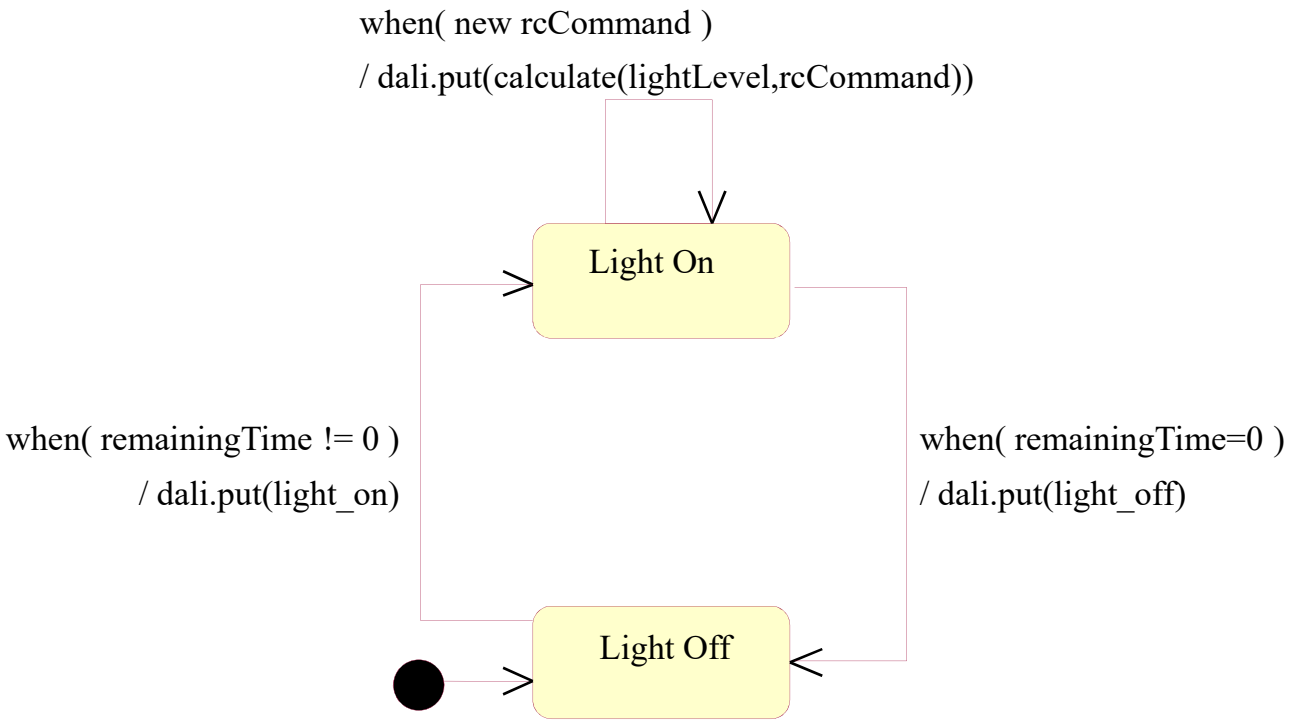
P_{Dali} (T) ::=

```
state= Idle;
while(waitForNextPeriod(T)) {
    if(state == Idle) {
        if( !dali.isEmpty() ) {
            daliHW=start;
            daliCommand = dali.get();
            memory.put(daliCommand, address);
            state=Sending;
        }
    } else if (state == Sending) {
        if( daliCommand.atEnd() ) {
            daliHW=stop;
            state=Idle;
        } else {
            daliHW=daliCommand.nextElement() );
        }
    }
}
```

Step 7

- Make a state machine diagram for the control of the LightController object
- Sketch an implementation

Main Abstract State Machine



Control Object

P_{Main} (T)::=

```
state = LightOff;
while( waitForNextPeriod(T) ) {
        if( state == LightOn ) {
            if( remainingTime == 0 ) {
                    state = LightOff;
                    dali.put(light_off);
                    rc.flush();
            }
        } else if (state == LightOff ) {
            if( remainingTime != 0 ) {
                    state = LightOn;
                    dali.put(light_on);
            } else rc.flush();
        }
        if (state == LightOn) {
            if( ! rc.isEmpty() ) {
                    rcCommand = rc.get();
                    daliCommand = calculate(lightLevel, rcCommand);
                    dali.put(daliCommand);
            }
        }
}
```

Assignment 8

- Compute the response times of the various tasks given the following numbers

Timings

Process	Ci (µs)	Ti (µs)	Di (µs)	Priority
Dali	100	416	10% of Ti = 41 µs ?	1
RC5	70	222	=Ti	2
Timer	30	250	=Ti	3
LM	30	250	=Ti	4
I2C	30	500	=Ti	5
PD	40	10000	=Ti	6
Main	400	40000	=Ti	7

Schedulability Formula

- From earlier schedulability theory (see also Burns and Wellings, 1996)
  - Recursive relation for the response time of a task

$$r_i = \sum_{j=1}^{i-1} C_j \left\lceil \frac{r_i}{T_j} \right\rceil + C_i + \max_{j=i+1}^{m} C_{Mine,j}$$

Blocking

Workshop assumptions

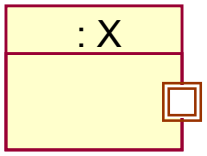
- Asynchronous communication (infinite buffers)
  - All parts communicate passing messages via channels
  - Only the interfaces corresponding to polled sensors and actuators are simple ports
- Composite structures with simple language mapping (C++ and Java – like)
- Behavioral State Machine modeling

Design Rules

- Parts are either
  - composition of other parts or
  - a sequential state machines
- So “leaf” parts are sequential state machines

Refined Design Language

- Active Part



- Port with infinite buffers

- Directed connector passing messages

- Writable device register

- Readable device register

Event Syntax

- Events
  - Message arrival on a receivePort. The message denotes the event. Messages are values.
    - Notation: *m <<from>> port [m=0]*
  - Change of a sensor value. The condition on the port's value denotes the event
    - *Notation: when(port.getValue() > 0*)

Contents

- Workshop

Case: mine pump

- The mine pump system is intended for regulating the ground-water level in a mine through the programmed control of a water pump
- The pump must be on when the water is too high (deadline *dh*)
- The pump must be off when the water is too low (deadline *dl*)
- This pump must not be operating if the concentration of methane gas is too high in the mine, because of the risk of explosions (deadline *dc*)
- When the pump is broken (no flow while pump is on) it must be switched off (deadline *df*)

Sensors and actuators

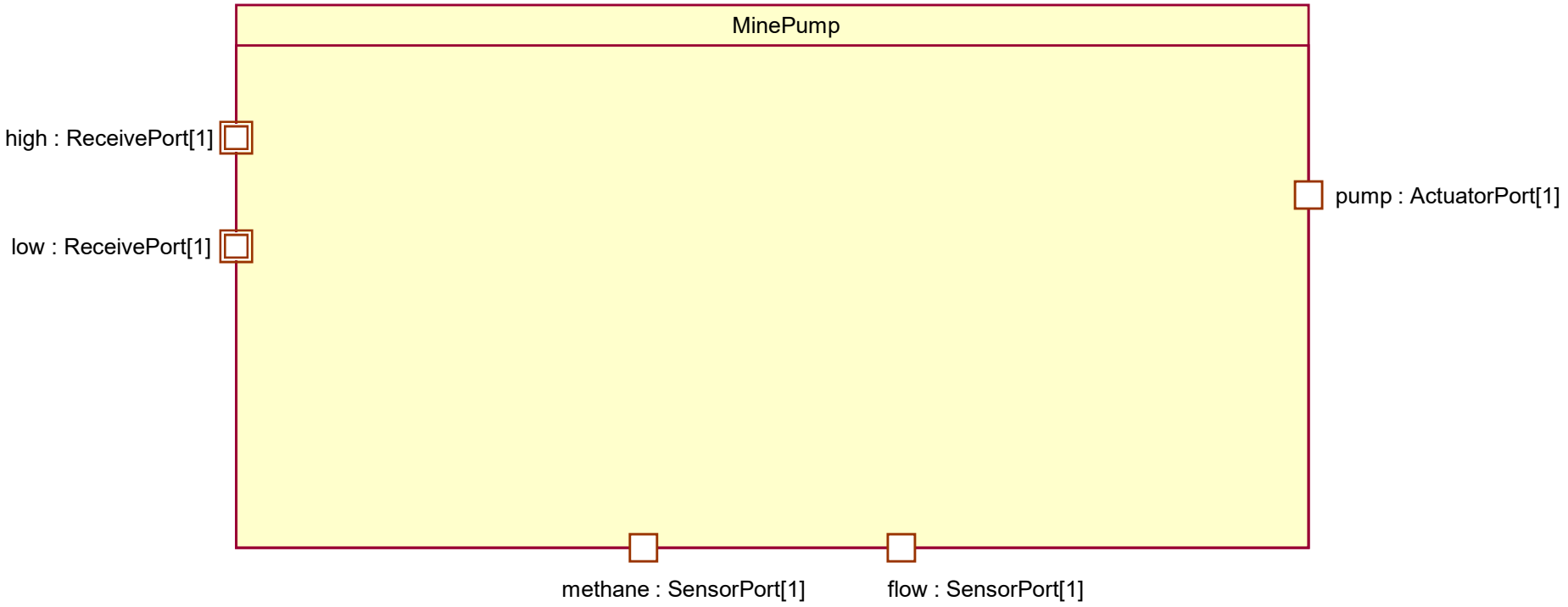
- The high and low water situations are made known to the system through messages
  - The environment sends messages
    - send(high) when water becomes high
    - send(low) when water becomes low
- The water level requires a long time to become low from high and high from low

- The other sensors and actuators are polled.

Deadlines

- dh = dl = 10 s
- dc = 30 ms
- df = 100 ms
- dlh = 100 s

System boundary



Workshop

- Make an analysis model (class diagram and state charts)
- Design a composite structure using heuristics
- Give the behavior of each of the objects in the design.
- Implement the program
- Indicate all periods and deadlines
- Define an execution model for on-line preemptive scheduling of this program, assign program fragments to tasks
- Analyze the schedulability