Embedded Systems

WS 08/09

Maritta Heisel

Maritta.Heisel(AT)uni-duisburg-essen.de

Denis.Hatebur(AT)uni-duisburg-essen.de

University Duisburg-Essen – Faculty of Engineering
Department of Computer Science
Workgroup Software Engineering

Content of lecture

Lecture

- Characteristics of embedded systems
- Development process for embedded systems
- Notations to be used in the development process
- If we have time: safety and security aspects of embedded systems, fault tolerance

Practical part of the course

- Development of a simple embedded system according to the development process

Organizational issues of the course

- Lecture: Tuesday 12–14, room BA 143
- Exercises and practical training: Tuesday, 14–16, room BA 143
  beginning: Oct. 21, 2008
- Course material will be published under
  http://swe.uni-duisburg-essen.de/

Organizational issues of the lab I

- Set up groups of at least 3 and at most 5 students.
- Announce your group until 2008-10-19⇒ 1 email per group with names and matr.-numbers (denis.hatebur@uni-due.de)!
- Work on tasks and submit the group solution **and all previous solutions in one .pdf-file** until following Sunday 23:59. The email must include names and matr.-no of all members. The .pdf-file should include only the number of the group.
- If more than two solutions are submitted too late, the whole group will not pass the lab.
- All tasks must be processed.

Organizational issues of the lab II

- To pass you have to attend all labs and submit all solutions in time (max. 2 exceptions).
- Everyone has to present the group solution at least (!) once.
- It must be indicated in the mail who performed the tasks and who performed the validation.
- All solutions will be published on the web.

Literature I

- Alan Burns and Andy Wellings: *Real-Time Systems and Programming Languages*.
  Pearson Education, 2001.
- Denis Hatebur: *A Pattern- and Component-Based Process for Embedded Systems Development*.
  University Duisburg–Essen, 2006,
  http://swe.uni-duisburg-essen.de/intern/dpes.pdf
- David E. Simon: *An Embedded Software Primer*.
  Addison-Wesley 2004.
- Ahmad Ibrahim: *Fuzzy Logic for Embedded Systems Applications (Embedded Technology)*, 2003

Literature II

- Manfred Broy and Wolfgang Pree: *Ein Wegweiser für Forschung und Lehre im Software-Engineering eingebetteter Systeme*,
  Informatik Spektrum, 18/2003, Volume 18.
- Michael Jackson: *Problem Frames. Analyzing and structuring software development problems.*
  Addison Wesley, 2001.
- Michael Jackson. Problems and requirements. In *Proceedings of the IEEE Second International Symposium on Requirements Engineering*. ACM Press, 1995.
- Michael Jackson and Pamela Zave. Deriving specifications from requirements: an example. In *Proceedings 17th Int. Conf. on Software Engineering, Seattle, USA*, S. 15–24. ACM Press, 1995.

Literature III

- Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, January 1997. Available at http://www.research.att.com/~pamela/ori.html#fre
- UML Superstructure Specification, v2.0 (709 Pages, 5.4 MB) http://www.omg.org/docs/formal/05-07-04.pdf
- Laurent Doldi: *UML 2.0 Illustrated.* TMSO, 2003. http://www.tmso-systems.com
- M. Jeckle, C. Rupp, J. Hahn, B. Zengler, S. Queins: *UML 2 glasklar.* Hanser, 2004.

Some definitions of embedded systems

- Embedded systems are computer-based systems being part of products other than a computer (Broy and Pree, after Simon)
- Embedded systems are information technology systems embedded in an electro-mechanical environment. (Borusan and Weber)
- . . . applications whose prime function is *not* that of information processing, but which nevertheless require information processing in order to carry out their prime function. (Ahmad Ibrahim)

About 99% of the worldwide production of microprocessors is used in embedded systems (Burns and Wellings).

Typical tasks of embedded systems (Burns and Wellings)

Process Control The computer interacts with its environment using sensors and actuators.
It controls the operation of the sensors and actuators to ensure that correct plant operations are performed at appropriate times.
Where necessary, analogue to digital (and vice versa) converters must be inserted between the controlled process and the computer.

Manufacturing The physical system consists of a variety of mechanical devices – such as machine tools, manipulators and conveyor belts – all of which need to be controlled and coordinated by the computer.

Application domains of embedded systems

- automotive
- aviation and space technology
- medical technology
- traffic guidance technology
- industrial automation
- telecommunications
- business
- entertainment
- household

Examples for embedded systems

- anti-lock braking system (ABS)
- smartcard
- washing machine
- traffic light
- temperature control unit
- elevator control unit
- ...

Characteristics of embedded systems I

- specialized for a particular purpose
- limited amount of resources (memory, power)
- high number of copies
- combination of hardware and software
- often security or safety critical
- connected via bus systems to other information technology systems
- larger embedded systems are often configurable
- faults in embedded software are expensive

Characteristics of embedded systems II

Embedded Software ...

- is usually reactive or continuous
- works on hardware with limited resources
- often has to fulfill safety or security requirements
- often fulfills timing requirements
- performs several tasks on one hardware

Embedded vs. real-time systems

Often used synonymously.
In contrast, we consider real-time systems to be a special kind of embedded systems:

> *A real-time system is any information processing activity of a system which has to respond to externally generated input stimuli within a finite and specified delay. (Burns and Wellings)*

Real-time does not mean to be very fast. But if a real-time system does not react within the specified delay, this is considered to be a system fault.
Hard real-time system: delay in reaction may cause danger to life of people or assets.

Overview of development process (DePES)

Overview of development process (DePES) I

1. Describe system in use
2. Describe system to be built
3. Decompose problem
4. Derive a machine behavior specification for each subproblem
5. Design global system architecture
6. Derive specifications for all components of the global system architecture
7. Design an architecture for all programmable components of the global system architecture that will be implemented in software

Overview of development process (DePES) II

8. Specify the behavior of all components of all software architectures, using sequence diagrams
9. Specify the software components of all software architectures as state machines
10. Implement software components and test environment
11. Integrate and test software components
12. Integrate and test hardware and software

Phase 1: Describe system in use

input:	informal description of the task	natural language
output:	context diagram of system in use	Jackson without machine domain
	shortcomings	natural language
	domain knowledge D ($F \wedge A$)	natural language, (HTA, state machines)
	glossary with definitions and designations	natural language
	list of possible development alternatives	natural language
validation:	all domains and phenomena in the context diagram must be described.	
	the context diagram must contain all domains necessary to describe the shortcomings.	
	shortcomings must be stated using elements of the domain knowledge description.	
	the glossary contains the notions used in D.	
	each entry in the list of possible development alternatives must consider at least one of the shortcomings.	

Phase 2: Describe system to be built

input:	all results of Phase 1	Jackson/ natural language
output:	system mission statement	natural language
	selected development alternatives	natural language
	context diagram of system to be built	ext. Jackson
	changed domain knowledge D $(F \wedge A)$	natural language, (HTA, state machines)
	initial set of requirements R_{init}	natural language
	requirements R to be implemented	natural language
validation:	only the limited set of operators is applied on the context diagram of system in use to derive the context diagram of system to be built	
	system mission statement must address the shortcomings or refer to domain knowledge of the system in use	
	domains and phenomena in the context diagram and in R and D must be consistent	
	R must be a subset of R_{init}	
	changes in the domain knowledge must be justified by the requirements	
	$D \wedge R$ are non-contradictory	

Phase 3: Decompose problem

input:	requirements R to be implemented of Phase 2	natural language
	domain knowledge D of Phase 2	natural language
	context diagram of Phase 2	ext. Jackson
output:	set of problem diagrams with associated set of requirements	Jackson with dot-notation
	expression of the subproblem relationships	grammar
validation:	consistent with context diagram of Phase 2	
	requirements R of Phase 2 must be treated in at least one subproblem	

Phase 4: Derive a machine behavior specification for each subproblem P_i

input:	requirements R from Phase 2	natural language
	domain knowledge D from Phase 2	natural language
	problem diagram for P_i from Phase 3	Jackson with dot-notation
output:	specification S_{P_i} of machine to construct	natural language
	sequences of interactions with annotated states for the domains in the environment, expressing R_{P_i} and D_{P_i}	sequence diagrams with annotated states
	sequences of interactions on initialization	sequence diagram with annotated states
validation:	$D \wedge S_{P_i}$ are non-contradictory	
	$D \wedge S_{P_i} \Longrightarrow R_{P_i}$	
	all requirements must be captured	
	in the sequence diagrams refined phenomena of the problem diagrams are used as signals	
	direction of signals must be consistent with control of shared phenomena	
	signals must connect domains as connected in problem diagram	
	the relationships of Phase 3 must be consistent with the states	

Phase 5: Design global system architecture I

input:	context diagram from Phase 2	ext. Jackson
	problem diagrams from Phase 3	Jackson with dot-notation
	sequences of interactions between machine and environment of all subproblems from Phase 4	sequence diagrams
	expression of the subproblem relationships from Phase 3	grammar
output:	system architecture	composite structure diagram
	perhaps subcomponents (recursively)	composite structure diagrams
	purpose of each component	natural language
	specification of external interfaces	interface classes
	specification of interfaces between the components	interface classes
	technical description of hardware interfaces	natural language, figures
	expression of the subproblem relationships for all components	grammars

Phase 5: Design global system architecture II

validation:	all machine interfaces of the problem diagrams must be captured	
	the signals in the sequence diagrams must be the same as the signals in the external interfaces	
	to each programmable component at least one problem diagram must be associated	
	each problem diagram must be associated to at least one component	
	all domains in the problem diagrams being part of the machine must be associated to a component	
	each machine domain in the context diagram must occur in the architecture	
	purpose must be consistent with the associated requirements	
	the grammar for each component must describe a subset of the grammar in Phase 3	

Phase 6: Derive specifications for all components of the global system architecture

For each subproblem:

input:	architecture from Phase 5	composite structure diagrams
	interface specifications from Phase 5	interface classes
	subcomponents (if defined) from Phase 5	composite structure diagrams
	sequences of interactions from Phase 4	sequence diagrams with annotated states or existing technical documentation
output:	interface behavior of all components (test specification)	sequence diagrams with annotated states
validation:	sequence diagrams together must describe the same interface behavior as in Phase 4	
	all signals in the interface classes of Phase 5 must be used in at least one sequence diagram	
	direction of signals must be consistent with the required and provided interfaces of Phase 5	
	signals must connect components as connected in the system architecture of Phase 5	
	it must be possible to map the new states to the states of Phase 4	

Phase 7: Design a software architecture for all components of the global system architecture

input:	global system architecture from Phase 5	composite structure diagram
	problem diagrams from Phase 3	Jackson with dot-notation
	interface specifications from Phase 5	interfaces classes
	relationships between subproblems specified in Phase 5	grammars
	possibly reusable components from other projects (Phase 9)	active or passive classes with interface classes
	machine behavior specifications from Phase 4	sequence diagrams with annotated states
output:	layered software architecture for each subproblem	composite structure diagrams
	merged layered software architecture (with subcomponents)	composite structure diagrams
	purpose of each software component	natural language
	specification of interfaces between software components	interface classes
validation:	if no instantiation of architectural patterns: consistent with problem diagram	
	signals of Phase 4 sequence diagrams are interfaces of the application layer	
	direction of all signals consistent to each other and input	
	external interfaces must be consistent with the interfaces of the system architecture developed in Phase 5	

Phase 8: Specify the behavior of all components of all software architectures, using sequence diagrams

For each subproblem:

input:	software architectures from Phase 7	composite structure diagrams
	interface specifications from Phase 7	interface classes
	system behavior from Phase 4	sequence diagrams with annotated states
	interface behavior of all programmable components from Phase 6	sequence diagrams with annotated states
output:	interface behavior of all software components (test specification)	sequence diagrams with annotated states
validation:	all sequence diagrams together must describe the same interface behavior as in Phase 6	
	all signals in the interfaces classes of Phase 7 must be used in at least one sequence diagram	
	direction of signals must be consistent with the required and provided interfaces of Phase 7	
	signals must connect components as connected in the software architecture of Phase 7	
	it must be possible to map any new states to the states of Phase 6	

Phase 9: Specify the software components of all software architectures as state machines

input:	interface behavior from Phase 8	sequence diagrams with annotated states
	relationships between subproblems specified in Phase 5	grammars
output:	component overview description with references to interface classes	class diagram with ports, sockets and lollipops
	data types and operations defined using pre- and postconditions	class diagrams formulas or natural language
	state machines	state machine diagrams
	invariants	formulas or natural language
validation:	consistent with interface behavior from Phase 8	
	completeness of state machines (implies error-cases for user-interaction)	
	a class must be active if it contains an active class or a timer	

Phase 10: Implement software components and test environment

input:	software component behavior from Phase 8	sequence diagrams with annotated states
	specification of merged components of Phase 9	*different notations*
output:	test software for software components	programming language or test language
	implemented software components	programming language
validation:	run tests	test results

Phase 11: Integrate and test software components

input:	global software architecture from Phase 7	composite structure diagrams
	software behavior from Phase 6	sequence diagrams with annotated states
	implemented software components from Phase 10	programming language
output:	implemented software	programming language
	test software for integrated software	programming language or test language
validation:	run tests	test results

Phase 12: Integrate and test hardware and software

input:	system architecture from Phase 5	composite structure diagram
	system specifications from Phase 4	sequence diagrams with annotated states
	expression of the subproblem relationships from Phase 3	grammar
	implemented software from Phase 11	programming language
output:	integrated system	machine
	acceptance test cases	test system and/or test plans
validation:	run tests	test results

Phase 1: Describe system in use

1. Describe system in use
2. Describe system to be built
3. Decompose problem
4. Derive machine behavior specification for each subproblem
5. Design global system architecture
6. Derive specifications for all components of the global system architecture
7. Design a software architecture for all components of the global system architecture that should be implemented in software
8. Specify the behavior of all components of all software architectures, using sequence diagrams

...

Phase 1: Describe system in use

input:	informal description of the task	natural language
output:	context diagram of system in use	Jackson without machine domain
	shortcomings	natural language
	domain knowledge D ($F \wedge A$)	natural language, (HTA, state machines)
	glossary with definitions and designations	natural language
	list of possible development alternatives	natural language
validation:	all domains and phenomena in the context diagram must be described.	
	the context diagram must contain all domains necessary to describe the shortcomings.	
	shortcomings must be stated using elements of the domain knowledge description.	
	the glossary contains the notions used in D.	
	each entry in the list of possible development alternatives must consider at least one of the shortcomings.	

Notations and concepts

- Terminology
- Context diagrams
- Requirements
- Domain knowledge
- Glossary
- Specification

Terminology: System, machine and environment I

- A system consists of a machine and its environment.
- System is a recursive term: A system can consist of other systems.
- The machine is the system to be built.
- Each machine acts in an environment

Terminology: System, machine and environment II

Goal: Design of a system with specified characteristics
Example: An elevator should enable persons in a building to get from one floor to another.
Components of the system:

- environment: part of "real world" relevant for the problem
  Example: floors, persons, cage, doors, engine, buttons, sensors, ...
- machine: controlling software and suitable hardware

Properties of the environment are fixed. We have to build the machine, so that it realizes the desired properties of the system.

Terminology: Phenomena I

State and behavior of the environment can be described by phenomena. Examples:

- **Elevator**
  Person presses button, expects, that the elevator arrives.
- **Bank**
  Client gives withdrawal instruction, expects a withdrawal.

Machine can interact with the environment by

- observing certain phenomena (input) ($\rightarrow$ sensors)
- causing certain phenomena (output) ($\rightarrow$ actuators)

Terminology: Phenomena II

Phenomena

- are actions/events/operations that occur in the environment
- are important for expressing **statements**
- can be observed or controlled by the environment or the machine, respectively

Examples:

- waiting in front of the elevator
- pressing the button inside the elevator
- elevator door closes

Terminology: Control of phenomena

1. Controlled by the environment, not observable by the machine
   Example: *waiting in front of the elevator*
2. Controlled by the environment, observable by the machine
   Example: *pressing the button inside the elevator*
3. Controlled by the machine, observable by the environment
   Example: *elevator door closes*

The category "controlled by the machine, not observable by the environment" is not considered in this phase, since internal phenomena of the machine do not belong to the requirements.

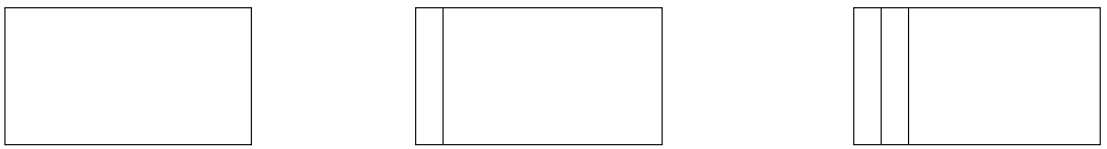
Context diagrams

- Distinction between environment and machine
- Representation of the connections between environment and machine
- Structuring the environment into a machine and (usually several) problem domains
- Connections between domains
- Represent the world, when the machine is in operation

Context diagrams – Notation

$$\begin{aligned} \text{Domains} &\mathrel{\widehat{=}} \text{Rectangles} \\ \text{Interfaces} &\mathrel{\widehat{=}} \text{Lines} \end{aligned}$$

Types of domains:



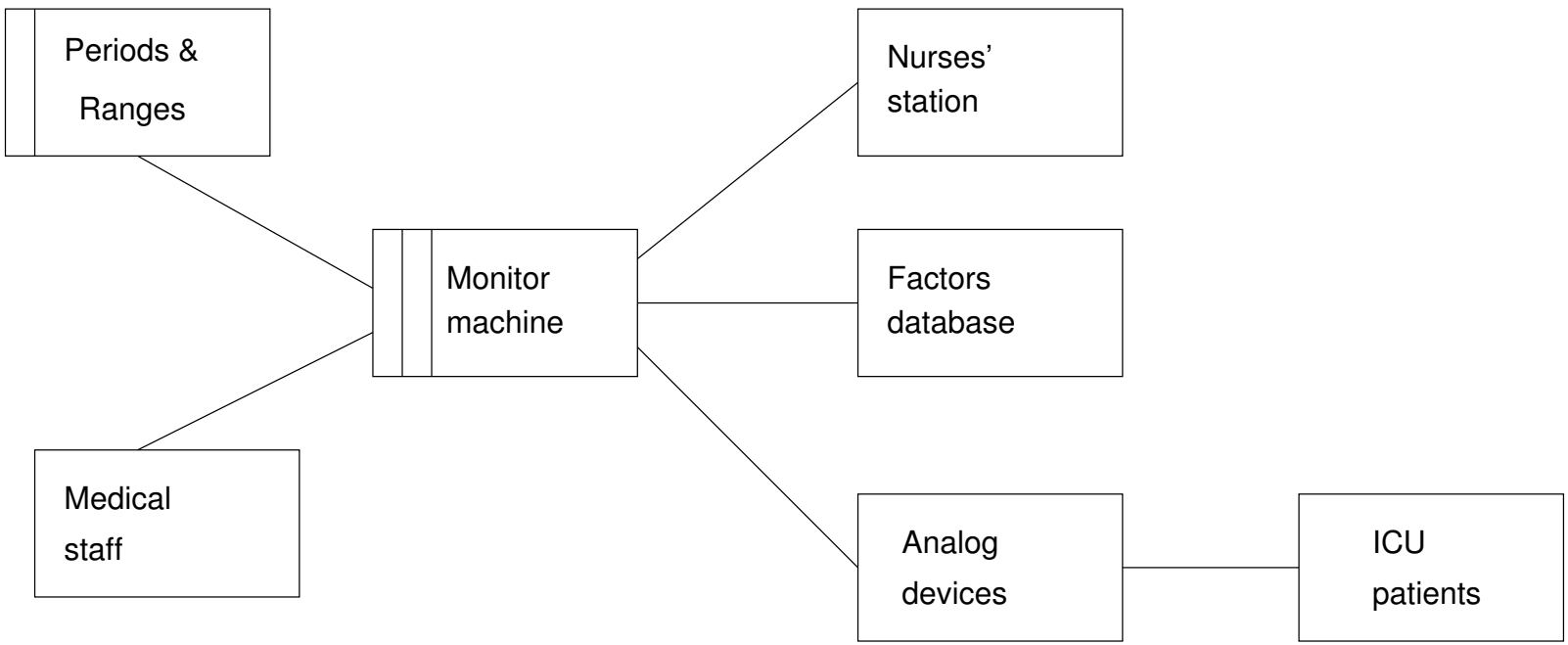
given domain | designed domain | machine domain

Interfaces between domains $\mathrel{\widehat{=}}$ *shared phenomena*
e.g., driving a nail with a hammer

Context diagram example: patient monitoring system

A patient monitoring program is required for the intensive-care unit of a hospital. Each patient is monitored by an analog device which measures factors such as pulse, temperature, blood pressure, and skin resistance.
The program reads these factors on a periodic basis (specified for each patient) and stores the factors in a database. For each patient, safe ranges for each factor are also specified by medical staff.
If a factor falls outside a patient's safe range, or if an analog device fails, then the nurses' station is notified.

Related (provisional) Context Diagram



Context Diagram of a Problem

- forms the basis for structuring and analyzing the problem
- shows all domains that need to be taken into consideration
- everything that does not appear in the context diagram, is not considered

$\Longrightarrow$ careful selection of domains necessary!

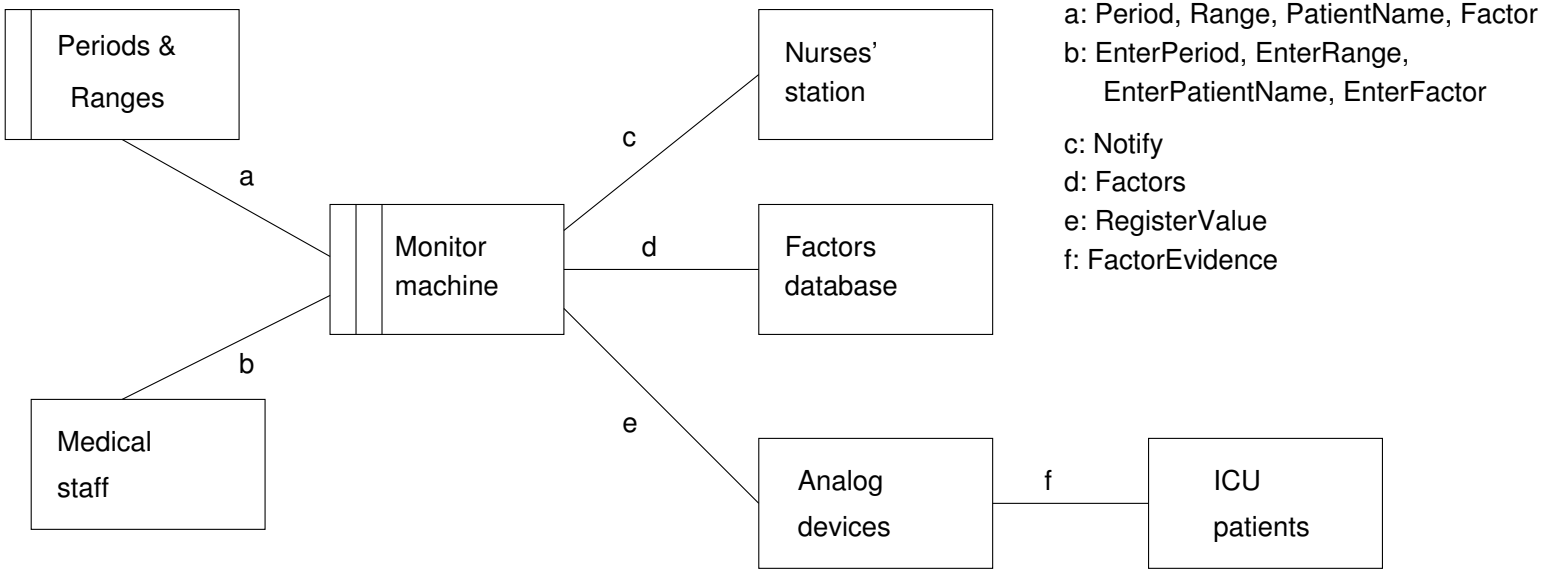
Example: different possibilities for the database domain

- Factor database as *given* domain
  $\Longrightarrow$ it already exists, does not have to be designed
- If it were part of the task:

  Only sensible, if the database is also used by other systems.
- Otherwise: Database as part of the machine that is to be constructed, *no separate domain* in the context diagram

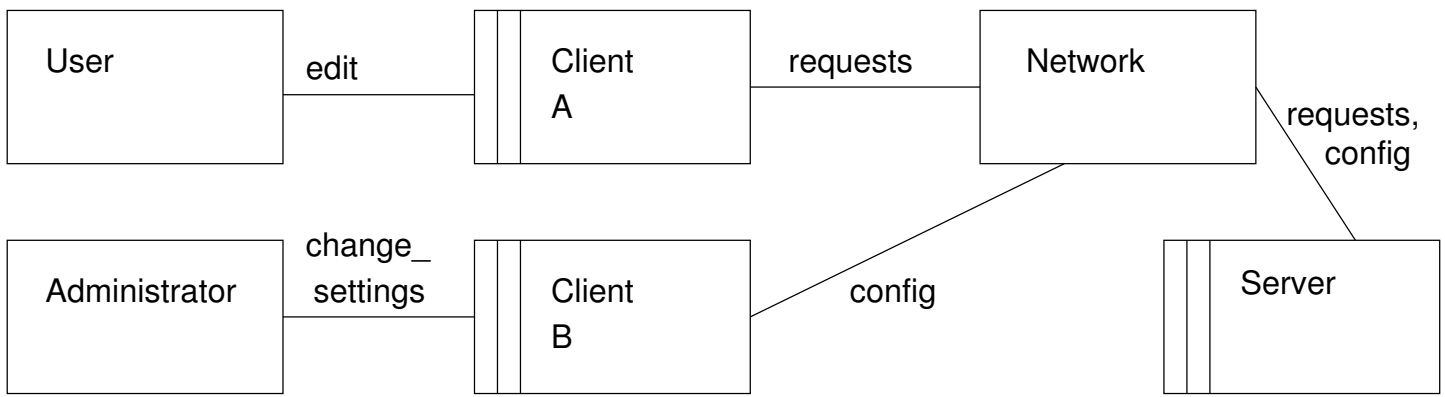
Context Diagram – Complete Notation

Write down shared phenomena at the connecting lines



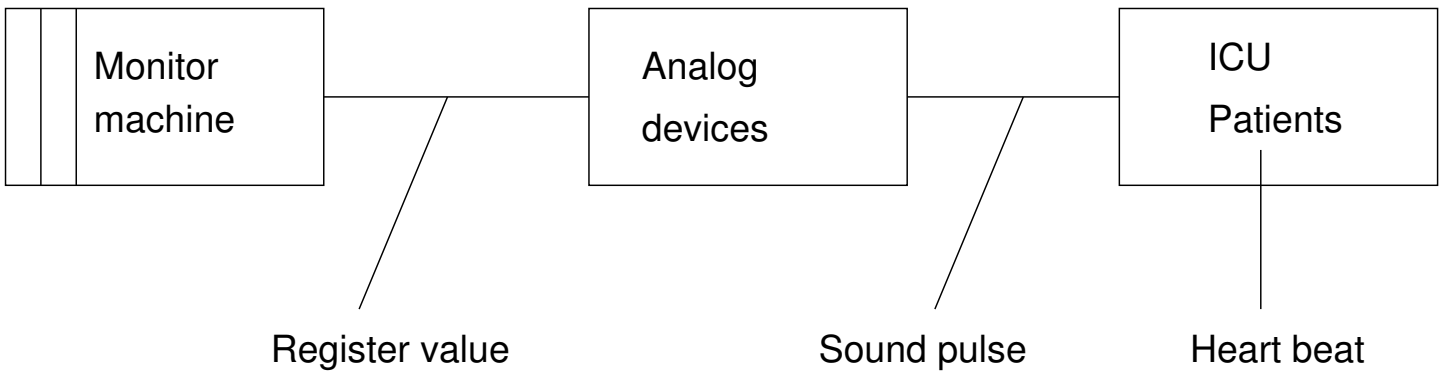
Extended Context Diagrams

Systems with more than one machine are necessary, when the machines are physically distributed (e.g., Client-Server Systems). Therefore, Jackson's context diagrams are extended to allow more than one machine domain.



Context Diagram – Connection Domains I

Patients and the machine are not directly connected, but indirectly through a causality chain:

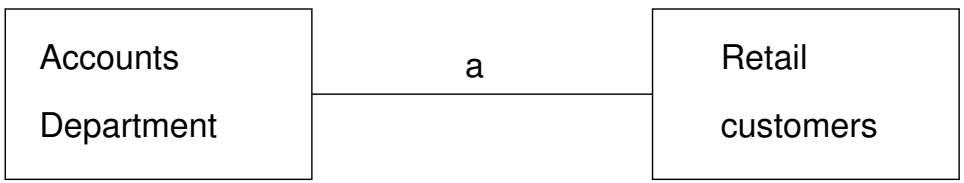


Context Diagram – Connection Domains II

Shared phenomena in context diagrams are abstractions of real phenomena, e.g. by

- omitting properties that are not relevant for the purpose at hand at that moment
  e.g., the event *Notify* surely has arguments
- treat complex episode of interaction as an instantaneous event

Care must be taken in the latter abstraction, as the following example taken from retail shows:



Context Diagram – Connection Domains III

If issuing an invoice and payment are carried out via postal mail, then the above mentioned context diagram would represent a (too great) abstraction of what actually happens :

b: SendBill, ReceivePayment

c: SendPayment, ReceiveBill

The phenomena *Bill* and *Pay* are in fact no shared phenomena of *AccountsDepartment* and *RetailCustomers*, since the mail acts as intermediary, which causes delays and uncertainties.
$\Rightarrow$ We need the *Post Office* as a connection domain.

Context Diagram – Connection Domains IV

Connection domains are necessary, if

- they are explicitly mentioned in the requirements, such as analog devices in the patient monitoring system
- may cause delays, that cannot be ignored, as shown in the retail example
- the transmission via the connection domain is unreliable, e.g., failure of an analog device.

Setting up Context Diagrams

Problem: circularity:

- It can only be decided on the context, if an overview of the problem is available.
- A problem is only properly known, if its embedding in environment is known.

$\Longrightarrow$ Iterative analysis of problem context and requirements

R, D & S – Requirements and specification I

- Known:
  (1) Fixed characteristics of the environment (domain knowledge)
  (2) Desired characteristics of the system (requirements)
- Clear: Machine must close the "gap" between (1) and (2)
- Searched: specification of the machine
  "How should the machine act, so that the system fulfills the requirements?"

Requirements describe the environment, the way it should be, after the machine is integrated.

R, D & S – Functional vs. non-functional requirements

- Functional requirements: state how the system should act
- Non-functional requirements: concern quality characteristics such as efficiency or user-friendliness

$$\text{fulfillment of functional requirements} \mathrel{\widehat{=}} \text{correctness}$$
$$\text{fulfillment of non-functional requirements} \mathrel{\widehat{=}} ???$$

Decisions on fulfillment of non-functional requirements need the definition of separate criteria!
In the following: only functional requirements

R & D – Types of statements

Indicative Statements describe the environment irrespective of how the machine is built.
Other notion: **domain knowledge**.
Example: a door cannot be open and closed at the same time.

Optative statements describe the environment, in the way we would expect it after the machine is integrated.
Example: After the button was pressed, the elevator stops at the corresponding floor.

Note: Statements are characterized by being true or false.

Requirements are thus optative statements.

D – Types of domain knowledge

Facts describe conditions that are always fulfilled.

Example: *When the motor turns right, the elevator moves to a higher floor*

(This fact is needed to transform the requirement "Move to another floor" into a specification with phenomena visible to the machine (turn motor right).)

Assumptions describe conditions that are needed, so that the requirements are satisfiable.

Example: *When the elevator reaches my floor, I enter*

(This assumption is needed for fulfilling the requirement that the elevator carries all waiting persons to their destination.)

D – Types of domain knowledge: Facts

- When it rains, the sensor gets wet.
- A wet sensor has an impedance below 100 Ω, and a dry sensor has an impedance above 200 Ω.
- When a airplane is on the ground and it is not stopped the wheels are turning. (?)
- When a car passes the sensor a pulse is generated.

D – Types of domain knowledge: Assumptions

- In case of fire the user presses the emergency button.
- The cars passing the sensor have a height of more than one meter.
- The button is pressed for more than 0.5 seconds.

R & S – Specifications

- are descriptions that are sufficient for building the machine
- are **implementable** requirements
- correctness condition:

If the machine fulfills the specification, the system fulfills the requirements.

$$S \wedge D \Rightarrow R$$

R & S – Specifications vs. requirements

Requirements are NOT implementable, if they

- constrain actions that are controlled by the environment
  Example: *The elevator is not be overloaded.*
- refer to actions that are not observable by the machine
  Example: *The elevator should go to a floor where people are waiting.*
- express conditions that can only be decided in the future
  Example: *As soon as a user has dialed the last digit, he receives the dial tone, the busy signal, or the announcement "number not assigned".*

Glossary – Designations

- System descriptions require designations as basic vocabulary
- Each designation has
  (1) a name
  (2) a (detailed) explanation
  Example.: A *student* is somebody who is enrolled at a university.
- With designations, we can form statements, e.g.

$$\forall\, s : \mathit{student} \bullet \exists\, l : \mathit{lecture} \bullet \mathit{enrolled}(s, l)$$

(assuming that designations of *lecture* and *enrolled* are available)

Glossary – Definitions

- expand the available vocabulary, but not its expressiveness
- can be absurd or useless, but not false

A defined notion can always be replaced by its definition.

Example:

$$student(s) \mathrel{\widehat{=}} \exists\, l : lecture \bullet enrolled(s, l)$$

Summary of the terminology

You should have learned the following notions:

- machine
- environment
- designation
- definition
- indicative statement
- optative statement
- assumption
- fact
- shared phenomenon (action / event / operation)
- requirement
- specification

Phase 1: Describe system in use

input:	informal description of the task	natural language
output:	context diagram of system in use	Jackson without machine domain
	shortcomings	natural language
	domain knowledge D $(F \wedge A)$	natural language, (HTA, state machines)
	glossary with definitions and designations	natural language
	list of possible development alternatives	natural language
validation:	all domains and phenomena in the context diagram must be described.	
	the context diagram must contain all domains necessary to describe the shortcomings.	
	shortcomings must be stated using elements of the domain knowledge description.	
	the glossary contains the notions used in D.	
	each entry in the list of possible development alternatives must consider at least one of the shortcomings.	

Executing Phase 1 I

All items of the output can be developed in parallel. The following points should be considered after we have an initial context diagram and some initial shortcomings:

- A shortcoming usually refers to some domains that must be introduced.
- Each domain requires a description of its domain knowledge.
- Each domain controls or observes some phenomena that must be decribed in the context diagram and may need a description in the glossary.
- For a shortcoming a new development alternative can be identified.
- A new development alternative may also address other shortcomings.

Executing Phase 1 II

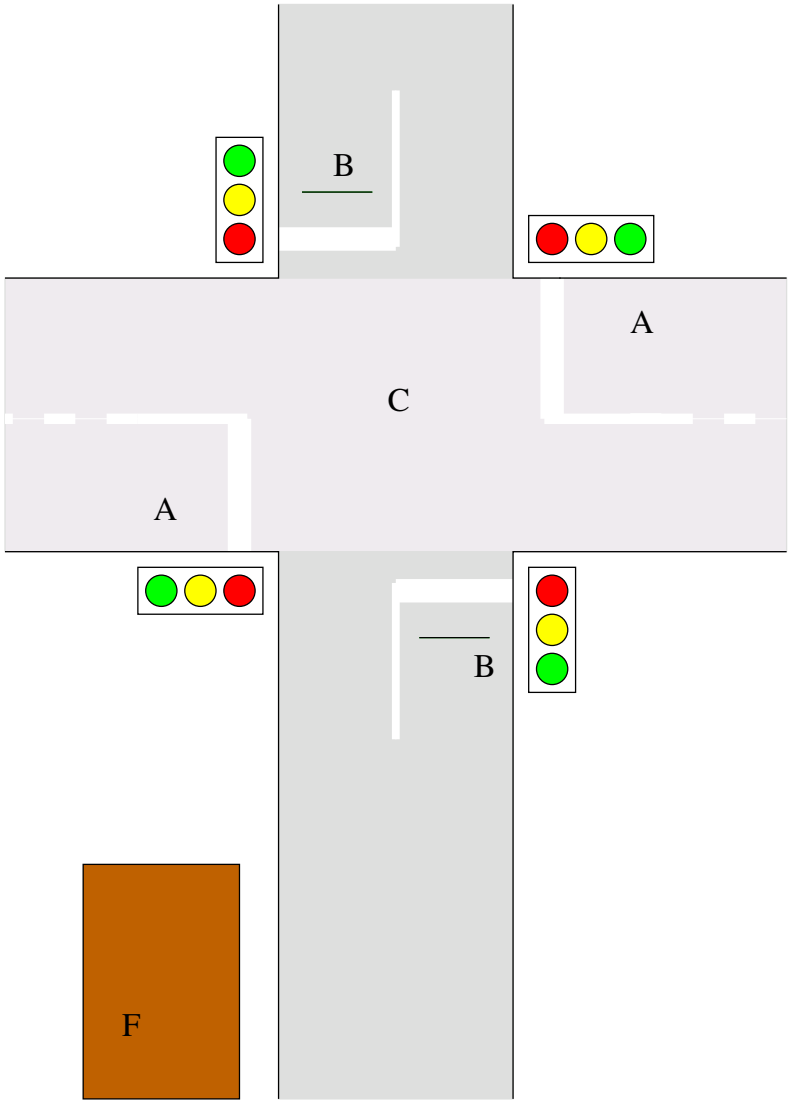
- An assumption may show us that there are additional shortcomings.

Remarks I

- Only the actual state is described.
- Possible solutions are discussed.
- Problem: Which domains should be described.
- Solution: Those domains, that are relevant for describing the shortcomings.
- For users in the environment a Hierarchical Task Analysis (HTA) can be performed, see
  www.hfidtc.com/pdf/reports/HTA%20Literature%20Review.pdf .
- Other systems in the environment can be described by state machines (the notation is introduced in Phase 9).
- Do not forget to include the existing solution in the list of alternatives (even if it addresses no shortcoming).

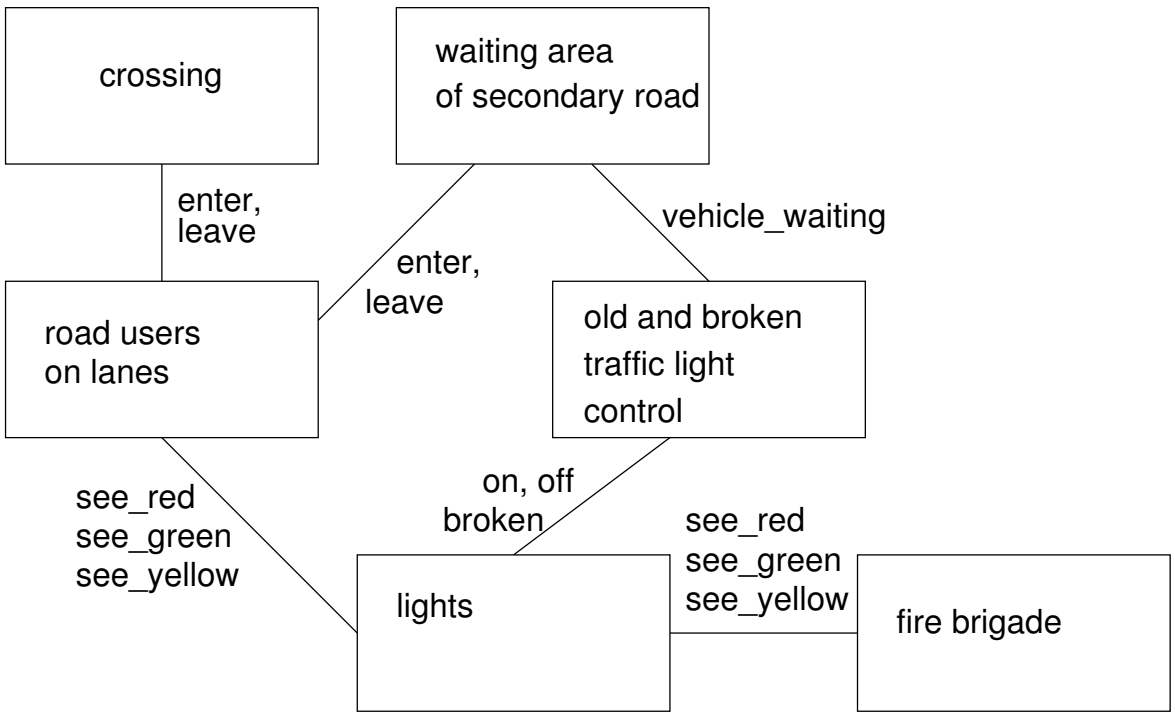
Example 1: traffic light control

Informal description of the task



We should build a system that prevent accidents on the crossing. It should also help the fire brigade (on the left) to pass the crossing and arrange for a fair and adapted flow of traffic between the main road (horizontal) and the secondary road (vertical).

Context diagram of system in use



Shortcomings

SC1: The old traffic light control is broken and cannot be repaired and improved.

SC2: Vehicles of the fire brigade are disturbed by cars on the secondary road when the secondary road is not allowed to pass. This causes delays in case of fire.

SC3: The amount of traffic has increased on both roads.

SC4: Too many accidents happened without working traffic lights.

Domain knowledge: Facts I

F1 : Traffic rule: stop if red *(see_red)*.

F2 : Traffic rule: cross if green. *(see_green)*

F3 : Traffic rule: leave **crossing** as fast as possible.

F4 : Fair means (for this **crossing**) that vehicles on the main road are allowed to pass the crossing for more than twice the time vehicles of the secondary road are allowed.

F5 : Traffic rule: if yellow *(see_yellow)*: stop if possible.

F6 : Vehicles cannot stop immediately.

F7 : A *broken* light bulb can be detected by measurement of the electric current (no current = no light).

F8 : There are tunnels for pedestrians.

F9 : Induction loops can be used for monitoring the **waiting area of secondary road**.

Domain knowledge: Facts II

F10 : The **lights** operate with a power supply of 24 V.

F11: The **fire brigade** is allowed to ignore the red light (but must be careful).

F12 : If the red phase is more than 30 s, some car drivers ignore the red light.

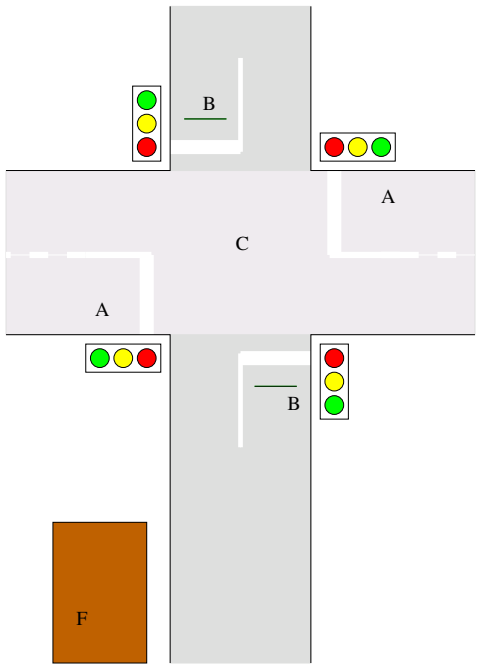
F13 : The **old traffic light control** is broken and cannot be repaired. Before, the vehicles on the main road were allowed to pass until a vehicle on the secondary road was detected. Cars in the secondary road were allowed to pass for 5 seconds.

Domain knowledge: Assumptions

A1 : All vehicles follow the traffic rules.

A2 : Pedestrians use the pedestrian tunnels.

Glossary for crossing



- lane / waiting area of main road: A
- lane / waiting area of secondary road: B
- traffic lights: device containing colored light bulbs to signal “stop” or “go”
- fire brigade: F
- crossing: critical section: C
- vehicle waiting: sensor detecting if a vehicle is in the waiting area of the secondary road
- accident: 2 or more vehicles at the same time at the same place

List of development alternatives

ALT1: Remove old traffic lights and use signs (addresses shortcoming SC1)

ALT2: Replace old traffic lights by a roundabout/rotary (addresses shortcomings SC1, SC3, and SC4)

ALT3: Replace the broken traffic lights control by a new traffic lights control (addresses shortcomings SC1, SC2, SC3 and SC4)

ALT4: Leave everything as it is

Validation I

All domains domains and phenomena in the context diagram are described:

crossing:	F4
road users on lanes:	F1, F2, F3, F5, F6, F12, A1
waiting area of secondary road:	F9
old and broken traffic light control:	F13
lights:	F10, F7
fire brigade:	F11

The pedestrians are not considered in the context diagram, since pedestrian tunnels exists and are used (F8, A2). The

Validation II

context diagram contains all domains necessary to describe the shortcomings and the shortcomings are stated using elements of the domain knowledge description.

SC1: old and broken traffic light control

SC2: fire brigade, waiting area of secondary road, road users on lanes

SC3: old and broken traffic light control

SC4: old and broken traffic light control

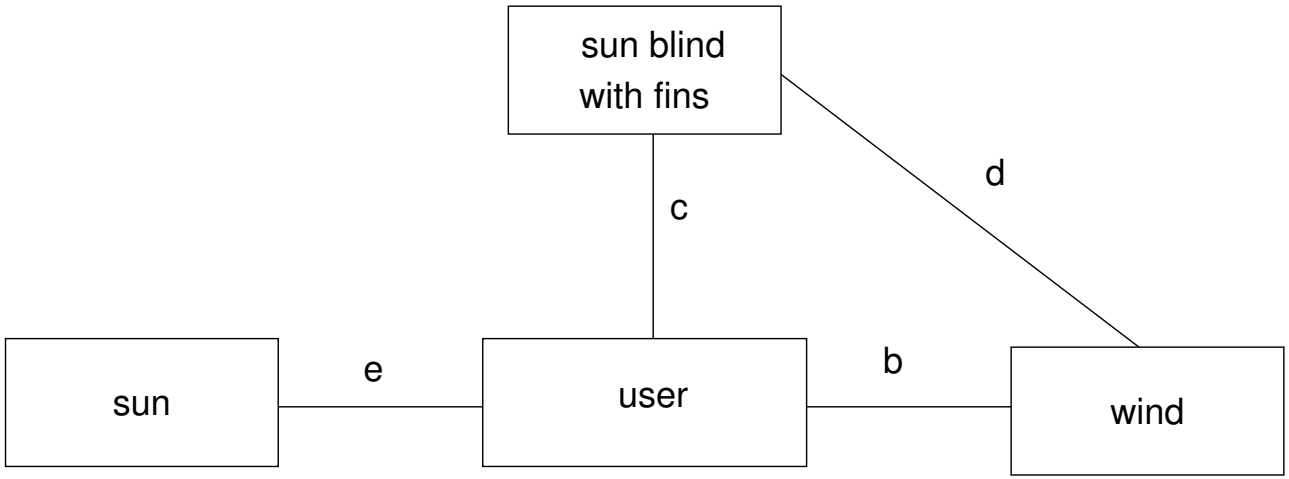
The glossary shoud contain the notions used in D. The given TLC glossary is just an example and not complete. Each entry in the list of possible solutions considers at least one of the shortcomings. This is directly stated on the corresponding slide.

Example 2: sun blind control

Informal description of the task

A building is equipped with sun blinds made up of metallic fins which are attached to the outer side of the window. The sun blinds are controlled manually. Unfortunately, the sunblinds can be destroyed by heavy wind if they are not pulled up. The system should be improved to achieve a comfortable working ambiance by not disturbing the users by sunshine.

Context diagram of system in use



b: heavy wind, no heavy wind

c: lower sun blind, pull up sun blind, stop sun blind, sun blind is lowered, sun blind is pulled up, rotate fins with positive degree, rotate fins with negative degree

d: destroy sun blind

e: sunshine, no sunshine

Shortcomings

SC1: Users forget to pull up the sunblind when there is no sunshine what may have an influence on the well-being and health of users.

SC2: Users forget to pull up the sunblind when there is heavy wind or even do no recognize that there is heavy wind.

SC3: Users have to stop their work to lower the sunblind if there is sunshine with high intensity and the users cannot read their monitor content.

SC4: Sometimes users destroy the sunblind accidentally by pulling heavily at the wires.

Facts

F1: Interface between user and sun blind are the control wires.

F2: The intensity of sunshine on a sunny day ranges from 32 000 lux to 100 000 lux. More than 32 000 lux makes it hard to read content of a standard monitor.

F3: The fins are turnable from 80° to 0° and to -80°. If a user tries to rotate more (with normal power), nothing happens.

F4: Heavy wind has a speed of more than 80 kilometers per hour. No heavy wind has a speed of less than 80 kilometer per hour.

F5: The sun blind is destroyed by heavy wind.

F6: The sun blind is destroyed if the user pulls too heavily at the wires (especially if the sunblind is lowered or pulled up).

F7: The fins have an interface that can be directly connected to a microcontroller.

Assumptions

Assumptions, description of user:

A1: Users usually lower the sunblind when the sun is shining.

A2: Users pull up the sunblind when the sun is not directly shining

A3: Users pull up the sunblind when there is heavy wind.

A4: Users adjust the fins as convenient.

A5: User recognize if the wind is blowing heavily (heavy wind).

A6: Users pull up the sunblind when they want to look out of the window.

A7: Users lower the Sunblind when they do not want to be seen.

Glossary for sun blind (example)

Designations:

metallic fins: metallic plates rotating on a horizontal axis.

outer side: not inside the building.

windows: part of a building made of glass.

sun blind: is made up of *metallic fins* which are attached to the *outer side* of the *window*.

sinshine: ...

wind: ...

wire: ...

microcontroller: ...

monitor: ...

List of development alternatives

ALT1: Hire an employee who pulls up all sunblinds in case of heavy wind. (addresses shortcoming SC1)

ALT2: Replace sunblinds by drapes or other types of sunblinds. (addresses shortcomings SC2 and SC4)

ALT3: Replace the display units by monitors with higher intensity and let the sunblind pulled up. (addresses shortcomings SC1, SC2, SC3, and SC4)

ALT4: Automatic sunblind control (addresses shortcomings SC1, SC2, SC3, and SC4)

ALT5: Leave everything as it is

Validation I

All domains domains and phenomena in the context diagram are described:

User:	A1 - A7, F1
Wind:	F4, F5
Sunblind with fins:	F1, F3, F5, F6, F7
Sun:	F2

Validation II

The context diagram contains all domains necessary to describe the shortcomings and the shortcomings are stated using elements of the domain knowledge description.

SC1: User, Sun, and Sunblind

SC2: User, Wind, and Sunblind

SC3: User, Sunblind (monitor intentionally left out, seen as part of user domain)

SC4: User, Sunblind (wires intentionally left out, seen as part of Sunblind domain)

The glossary contains exactly the notions used in D, but is not complete. Each entry in the list of possible solutions considers at least one of the shortcomings. This is directly stated on the corresponding slide.

Phase 2: Describe system to be built

1. Describe system in use
2. Describe system to be built
3. Decompose problem
4. Derive machine behavior specification for each subproblem
5. Design global system architecture
6. Derive specifications for all components of the global system architecture
7. Design a software architecture for all components of the global system architecture that should be implemented in software
8. Specify the behavior of all components of all software architectures, using sequence diagrams

...

Phase 2: Describe system to be built

input:	all results of Phase 1	Jackson/ natural language
output:	system mission statement	natural language
	selected development alternatives	natural language
	context diagram of system to be built	ext. Jackson
	changed domain knowledge D $(F \wedge A)$	natural language, (HTA, state machines)
	initial set of requirements R_{init}	natural language
	requirements R to be implemented	natural language
validation:	only the limited set of operators is applied on the context diagram of system in use to derive the context diagram of system to be built	
	system mission statement must address the shortcomings or refer to domain knowledge of the system in use	
	domains and phenomena in the context diagram and in R and D must be consistent	
	R must be a subset of R_{init}	
	changes in the domain knowledge must be justified by the requirements	
	$D \wedge R$ are non-contradictory	

Executing Phase 2 I

- The system mission statements have to be defined. The system mission statements can be derived from good properties and from the shortcomings of the system in use.
- The development alternatives can be compared according to their estimated costs and addressed shortcomings.
- The context diagram of the system to be built is created by applying the following rules:
  - Introduce domain
  - Split domain
  - Remove domain with connected interfaces
  - Replace/change domain
  - Add or remove interfaces and phenomena for introduced, changed or replaced domains and between split domains
  - Connect interfaces to other domains

Executing Phase 2 II

- State the requirements in terms of domains and phenomena of the context diagram.
- Give the condition for each requirement explicitly. Requirements consist of a precondtion and a postcondition. (when / if ... happens, do ...)
- State the domain knowledge for all introduced and replaced domains.
- Verify and update existing domain knowledge.

For each system mission statement:

- Determine all requirements that are necessary to achieve it.
- Determine if the necessary requirements (together with the domain knowledge) are also sufficient to achieve the mission statement. If not, more requirements are needed.

Executing Phase 2 III

The requirements obtained in this way are the "need-to-have" (mission-critical) requirements.
All other requirements are "nice-to-have" requirements or the system mission must be adjusted. The "nice-to-have" requirements should be prioritized, and a return-on-investment analysis should be performed.
Result: Consolidated set of requirements to be achieved (all of the mission-critical plus selection of the "nice-to-have" requirements).

Remarks I

- Only the desired state is described.
- One of the possible solutions is chosen.

Example 1: traffic light control

System mission

SM1 : The system should prevent accidents on the crossing.

SM2 : The system should help the fire brigade to pass the crossing with priority.

SM3 : The system should arrange for a fair and adapted flow of traffic between the main and the secondary road (and maximize the flow of traffic).

Select development alternative

ALT1: Using signs will not improve the fair flow of traffic.

ALT2: A parcel of land must be bought to build a roundabout/rotary big enough for the vehicles of the fire brigade. The owner does not want to sell his parcel of land for an acceptable price. Estimated costs:

- Parcel: EUR 5.000
- Build the roundabout: EUR 20.000

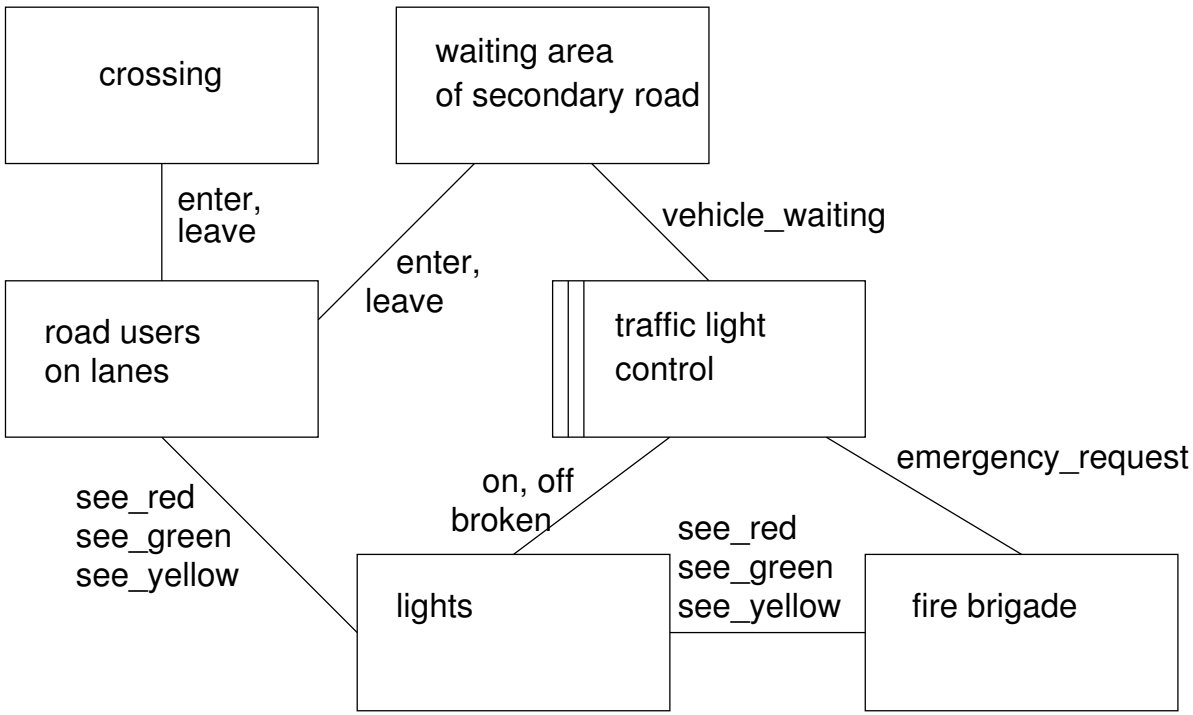
ALT3: For a new traffic lights control the old lights and induction loops can be reused. Estimated costs:

- New control: EUR 5.000
- Maintanance: EUR 500 / year

ALT4: Costs of accidents are much higher

The roundabout charges off after 40 years
$\Rightarrow$ new traffic lights control

New context diagram for traffic lights



- the *old and broken traffic light control* is replaced (replace domain)
- the *fire bridage* can send an *emergency request* (add interfaces with phenomena, change domain)

Changed/added/removed domain knowledge

- A1 and A2 are still true
- F1 – F12 are still applicable.
- F13 is removed (because domain is removed)

The following assumption is added:

A3 : In case of emergency the emergency switch is activated and deactivated *emergency_request* after crossing.

Initial requirements for traffic lights I

R1 : When there is a car waiting *(vehicle_waiting)* on the **secondary road**, the traffic **lights** should stop *(see_red)* the flow of traffic on the main road for a period of time and allow *(see_green)* the traffic flow on the secondary road.

R2 : When the emergency switch is activated *(emergency_request)*, the flow of traffic on the main road should be stopped *(see_red)* after a reasonable time and the flow of traffic on the secondary road should be allowed *(see_green)* after a reasonable time.

R3 : Vehicles on the main road should be allowed to pass the **crossing** for a longer period of time than from the secondary road (if not emergency-case[1]).

Initial requirements for traffic lights II

R4 : While vehicles on one road are allowed to pass *(see_green)*, the others should be stopped *(see_red)*.

R5 : The **lights** should switch in the following order: red - red+yellow - green - yellow - red. Other combinations (except "all off", yellow blinking, and green - yellow - green in emergency case[2]) are not allowed *(see_red, see_yellow, see_green)*.

R6 : In case of a *broken* light bulb the traffic **lights** should blink in yellow *(see_yellow)* for the secondary road, after all red lights have been switched on for a period of time *(see_red)*.

R7 : After switching to red *(see_red)*, the traffic flow of both roads should be stopped *(see_red)* for a period of time *[3].

[1] Added later to eliminate contradictions
[2] Added later to eliminate contradictions
[3] A star (*) denotes: added later

Consolidate traffic light requirements I

$$R \wedge F \wedge A \Longrightarrow SM$$

- SM1: avoid accidents
  Accidents will not occur if at most one road gets the "go" signal **and** cars have time to leave the crossing when the signal is changed to "stop", provided drivers obey to the rules.
  - necessary: R4 (at most one road may pass), R5 (yellow before red, red/yellow before green), R7 (both roads get "stop" signal for some time)

Consolidate traffic light requirements II

- sufficient:

  $$(R4 \wedge F1 \wedge A1) \wedge (F6 \wedge R5 \wedge R7 \wedge F3 \wedge F5 \wedge A1) \Longrightarrow SM1$$

  (only one road may pass, stop if red, vehicles follow rules) (vehicles cannot stop immediately, correct order of signalling, period with red for all, leave critical section as fast as possible, stop on yellow if possible, vehicles follow rules)

- SM2: priority for fire brigade
  This mission statement is achieved by the emergency button.
  - necessary: R2 (emergency button yields “go” signal for secondary road)

Consolidate traffic light requirements III

- sufficient: $R2 \wedge F1 \wedge F2 \wedge A3 \wedge A1 \Longrightarrow SM2$
  (emergency button stops main road, stop if red, drive if green, button is pressed on emergency, vehicles follow rules)

- SM3: fair traffic flow
  Requests from the secondary road must be taken into account, but main road should be allowed to pass twice as long as secondary road.
  - necessary: R1, R3
  - sufficient: $R1 \wedge R3 \wedge F4 \wedge A3 \Longrightarrow SM3$
    (secondary road request leads to "go", longer period for main road, definition of fairness, emergency button is released)

Consolidate traffic light requirements IV

Summary:
$R' = \{R1, R2, R3, R4, R5, R7\}$ (mission critical requirements)
$R_{init} \setminus R' = \{R6\}$
but $R6$ required for safety
$\Longrightarrow$ All requirements will be implemented, $R = R_{init}$.

Validation I

- The applied operators for the context diagram are given directly below the diagram.
- The system mission statement addresses the shortcomings or refer to domain knowledge of the system in use:
  - SM1 (prevent accidents) addresses shortcomings SC4
  - SM2 (help the fire brigade) addresses shortcoming SC2
  - SM3 (fair and adapted flow of traffic) addresses shortcomings SC3
- The phenomena and the domains of the context diagram are printed emphasized in the requirements and in the domain knowledge.

Validation II

- All given and designed domains are referenced in the requirements and the domain knowledge:
  - The *crossing* is referenced in $F4$ and $F6$.
  - The *waiting area of main road* is referenced in $R3$.
  - The *waiting area of secondary road* is referenced in $R1$ and $F9$.
  - The *road users on lanes* is referenced in $R1, R2, R3, R4, R7, F1, F2, F3, F5, F6, A1$.
  - The *fire brigade* with its emergency button is referenced in $R2, R3, A3$.
  - The *lights* are referenced in $R5, R6, F7$.

Validation III

- Usually, in all requirements and domain knowledge only elements of the context diagram are referenced. Pedestrians are referenced in $F8$ and $A2$. A domain *pedestrian* is not included in the context diagram since there are no shared phenomena with the machine or other domains being relevant for the problem.

- In $D \wedge R$ no contradictions were found.

Example 2: sun blind control

System mission

SM1: Achieve a comfortable working ambiance by not beeing disturbed by sunshine.

SM2: Achieve a comfortable working ambiance according to user wishes.

Select development alternative

For the SunBlind problem, we only consider the development alternatives addressing all shortcomings.

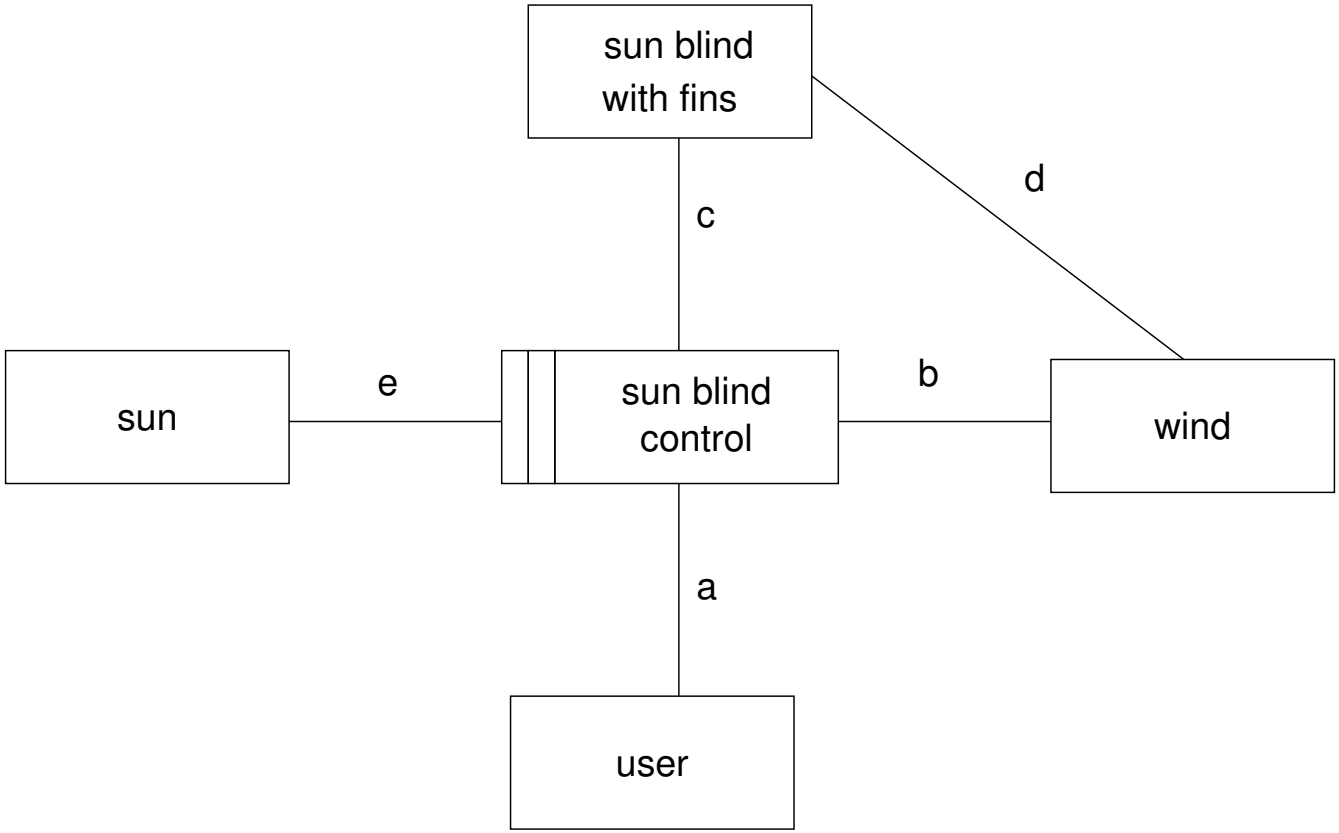
ALT3: Replace the display units by monitors with higher intensity and let the sunblind pulled up. (estimated costs for each PC 4000 EUR)

ALT4: Automatic sunblind control. (estimated costs for each window 1000 EUR)

Ratio PCs and Windows: 1 PC : 2 Windows

$\Rightarrow$ Automatic sunblind control

New context diagram for sun blind I



New context diagram for sun blind II

a: manually adjust fins with negative degree, manually adjust fins with positive degree, manually open sun blind, manually close sun blind, stop closing sun blind, stop opening sun blind (introduced with machine domain)

b: heavy wind, no heavy wind now connected to machine, not to user

c: lower sun blind, pull up sun blind, stop sun blind, sun blind is lowered, sun blind is pulled up, rotate fins with positive degree, rotate fins with negative degree now connected to machine, not to user

d: destroy sun blind

e: sunshine, no sunshine now connected to machine, not to user

Initial requirements for sun blind I

R1 If there is *sunshine* for more than one minute but *no heavy wind*, the **sun blind** will be lowered (*lower sun blind*). *(Negation of precondition of R3 is included to prevent contradictions. R8 has priority!)*

R2 If there is *no sun shine* for more than 5 minutes, the **sun blind** will be pulled up (*pull up sun blind*). *(R8 has priority!)*

R3 The **sun blind** should not be destroyed: If there is *heavy wind*, the **sun blind** will be pulled up (*pull up sun blind*).

Initial requirements for sun blind II

R4 If there is *no heavy wind* and the **user** *manually adjusts the fins with positive degree* the **fins** are rotated with positive degree (*rotate fins with positive degree*).
If there is *no heavy wind* and the **user** *manually adjusts the fins with negative degree* the **fins** are rotated with negative degree (*rotate fins with negative degree*).

R5 If the **user** *manually opens the sun blind*, the **sun blind** will be pulled up (*pull up sun blind*).

R6 If the **user** *manually closes the sun blind* and there is *no heavy wind*, the **sun blind** will be lowered (*lower sun blind*). *(Negation of precondition of R3 is included to prevent contradictions.)*

Initial requirements for sun blind III

R7 The **sun blind** should not be destroyed: When the **sun blind** is in its lowest position (*sun blind is lowered*) or in it highest position (*sun blind is pulled up*) the **sun blind** should stop (*stop sun blind*).

R8 If the **user** interacts with the **sun blind** (*manually opens the sun blind*, *manually closes the sun blind*, *manually rotate fins with positive degree* or *manually rotate fins with negative degree*), then *sun shine* and *no sun shine* are ignored for the next 4 hours.

Revised domain knowledge I

- A1, A2, A3, A5 are removed
- A4, A6, A7 are still true
- F2, F3, F4, F5, and F7 are still applicable.
- F1 is changed to: The control wires are connected to the motor of the machine to be built. When the motor turns right, the **sun blind** is lowered *(lower sun blind)*. When the motor turns left, the **sun blind** is pulled up *(pull up sun blind)*. When the motor stops, the **sun blind** is stopped *(stop sun blind)*.
- F6 is changed to: The wires are protected from the **user**. The **sun blind** is destroyed if the **Sun Blind Control** does not stop pulling or releasing the wires (within 0.2 s) when the *sunblind is pulled up* or *lowered*.

Consolidate sun blind control requirements

The system mission can be split into the following parts:

SM1: react to sun shine (R1, R2, F1, F2, F3, F4)

SM2: react to user needs (R4, R5, R6, R8, F3, F4, F5, A1, A2, A3)

All contradictions are eliminated. (R3)
Requirements R1, R2, R4, R5, R6, R8 are "need to have".
R3 and R7 (together with F4, F6, F7, A4) prevent sun blind from taking damage, thus contributing to the system mission SM1 and SM2.
$\Longrightarrow R_{init} = R$

Validation I

- The applied operators for the context diagram are given with the diagram.
- The system mission statement addresses the shortcomings or refer to domain knowledge of the system in use:
  - SM1 (react to sunshine to achieve a comfortable working ambiance) address shortcomings SC1 and SC3.
  - SM2 (react to user needs) is included to do not make things worse, and addresses the assumption A3, A6, A7 but also the shortcoming SC4 since there is no direct user interaction.
- The phenomena and the domains of the context diagram are printed emphasized in the requirements and in the domain knowledge.
- All given and designed domains are referenced in the requirements and the domain knowledge.

Phase 3: Decompose problem

1. Describe problem
2. Consolidate requirements
3. Decompose problem
4. Derive machine behavior specification for each subproblem
5. Design global system architecture
6. Derive specifications for all components of the global system architecture
7. Design a software architecture for all components of the global system architecture that should be implemented in software
8. Specify the behavior of all components of all software architectures, using sequence diagrams

...

Phase 3: Decompose problem

input:	requirements R to be implemented of Phase 2	natural language
	domain knowledge D of Phase 2	natural language
	context diagram of Phase 2	ext. Jackson
output:	set of problem diagrams with associated set of requirements	Jackson with dot-notation
	expression of the subproblem relationships	grammar
validation:	consistent with context diagram of Phase 2	
	requirements R of Phase 2 must be treated in at least one subproblem	

Notations and concepts

- Terminology
- Problem diagrams, simple problems
- Problem decomposition
- Problem frames
- Decomposition operations
- Subproblem relationships

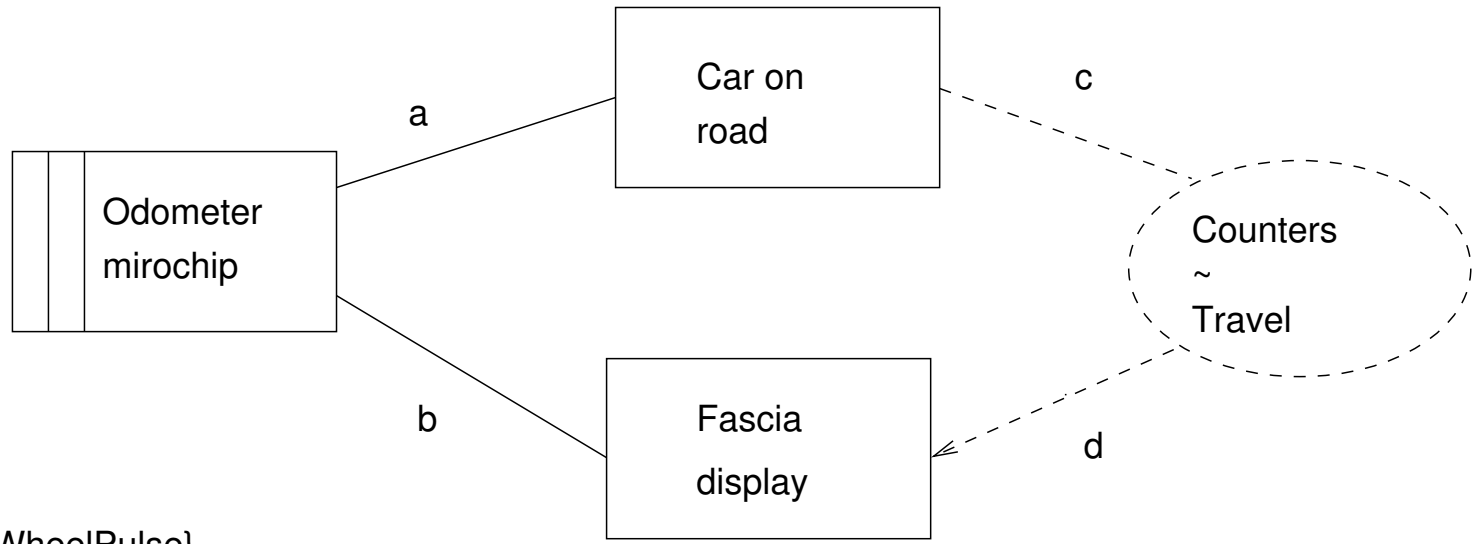
Terminology: Context Diagrams vs. Problem Diagrams vs. Problem Frames

- Context Diagram: "Where is the problem located?"
- Problem Diagram: "What is the problem?"
  - contains requirements (that refer to / constrain the problem domains)
  - contains information on who controls shared phenomena
- Problem Frame: Pattern for a Problem Diagram
  - describes classes of *simple* problems.

Simple Problem Example: odometer display problem

A microchip computer is required to control a digital electronic speedometer and odometer in a car. One of the car's rear wheels generates pulses as it rotates. The computer can detect these pulses and must use them to set the current speed and total number of miles traveled in the two visible counters on the car fascia. The underlying registers of the counters are shared by the computer and the visible display.

Corresponding problem diagram



a: CR!{WheelPulse}

b: OM!{IncSpeed, DecSpeed, IncDist}

c: CR!{Speed, CumDist}

d: FD!{SpeedCount, DistCount}

Simple problems are characterized by

- Simple requirements, i.e., the purpose of the machine is intuitively comprehensible.
- Simple interfaces, i.e., it is easy to characterize the way the parts interact at each interface.
- Simple roles of the domains, i.e., it is clearly defined, which domain should be constrained and which should only be observed, etc.

Decomposition of realistic problems into simpler subproblems

- Necessary for solving the problems.
- Also necessary for capturing, describing and understanding realistic problems.
- Questions:
  - How can a good decomposition be made?
  - How do you know, that solving subproblems is easier than solving the initial problem?
- Desirable: subproblems that belong to known classes (*Problem frames* as "problem patterns")

Approaches to decomposition I

- Top-Down-Decomposition ("oldest and worst approach")
  - Arrange functions in a hierarchy of several levels
  - At each level, decompose each function into a number of functions at the next level.
  - Stop when a level is reached where all functions are regarded as elementary.
  - Disadvantages
    - Approach takes no explicit account of the problem to be decomposed.
    - Unlikely to achieve a good decomposition if not already familiar with the problem.
    - Example: Reduce enumerating all prime numbers up to a certain limit to determining the next prime number greater than a given number. Subproblem is not simpler than original problem.

Approaches to decomposition II

- Use-Case-Decomposition
  - Known through object-oriented analysis.
  - Works well when it makes sense to think of the machine as a facility offering discrete services that are used in clearly defined episodes.
  - Not suitable for continuing interaction between machine and problem domains, as often needed for embedded systems, e.g. in patient monitoring problem.
- "Knowledge-based" decomposition through projection.
  - Decomposition into "parallel" (not hierarchical) subproblems.
  - Knowledge of problem classes and their solutions are used for decomposition.

Characteristics of subproblems

- Subproblems treat sets of related requirements.
- Subproblems are complete, independent problems with their own problem diagrams.
- When analyzing a subproblem, the other subproblems are considered as solved (*separation of concerns*).
- Subproblems are related to each other.
- Subproblems are *projections* of the overall problem, i.e., some aspects are ignored.

Problem frames

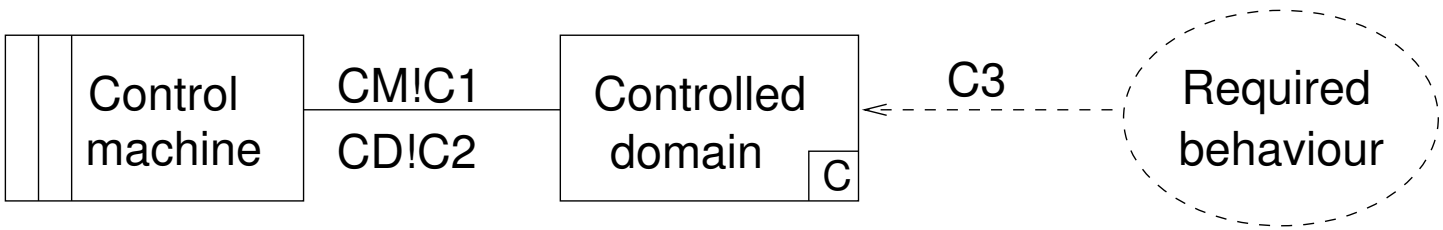
- are patterns that characterize frequently occurring problems
- simple subproblems resulting from problem decomposition should fit to a problem frame
- are represented with frame diagrams
- concrete problems are "fitted to problem frames"
- six fundamental problem frames and variants exist
- Literature: Jackson, Chapter 4

Problem frame: Required Behaviour

Informal Description

There is some part of the physical world whose behaviour is to be controlled so that it satisfies certain conditions. The problem is to build a machine that will impose that control.

Frame Diagram



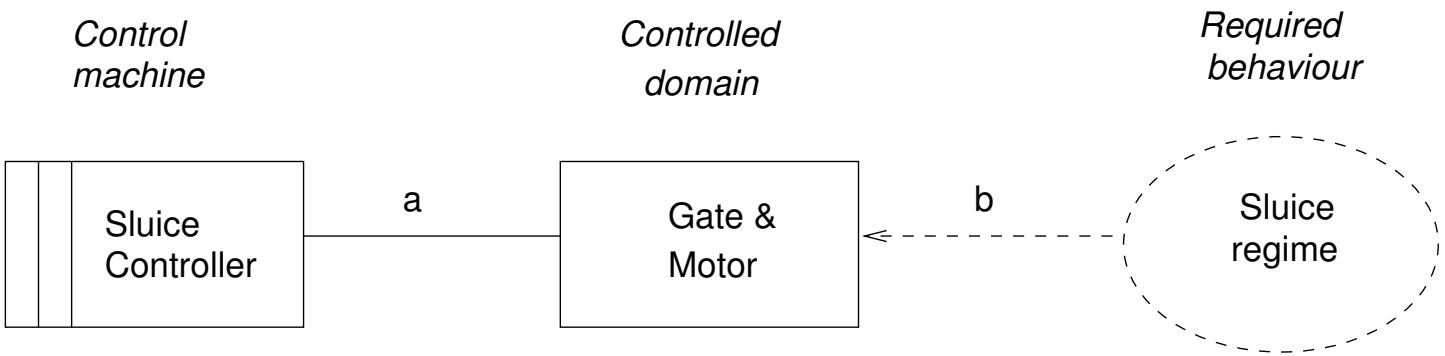
C: Causal domain, reacts to events in a predictable way
C1–C3: indicate *causal* relations, C2: *Feedback*

Example: Sluice Gate Control I

A small sluice is used in a simple irrigation system. The sluice gate can rise and fall. A computer system should control the sluice gate. The requirement is that the gate should be held in the fully open position for ten minutes in every three hours and otherwise kept in the fully closed position.
The gate is opened and closed by rotating vertical screws. The screws are driven by a small motor, which can be controlled by clockwise, anticlockwise, on and off pulses. There are sensors at the top and bottom of the gate travel. At the top it's fully open, at the bottom it's fully shut. The connection to the computer consists of four pulse lines for motor control and two status lines for the gate sensor. Sluice Gate Control fitted to

Example: Sluice Gate Control II

Required Behaviour frame



a: SC!{Clockw, Anti, On, Off} [C1]
GM! {Top, Bottom} [C2]
b: {Open, Shut} [C3]

Fitting problems to problem frames is achieved by instantiation of the variables contained in the frame diagram.

Problem frame: Commanded Behaviour I

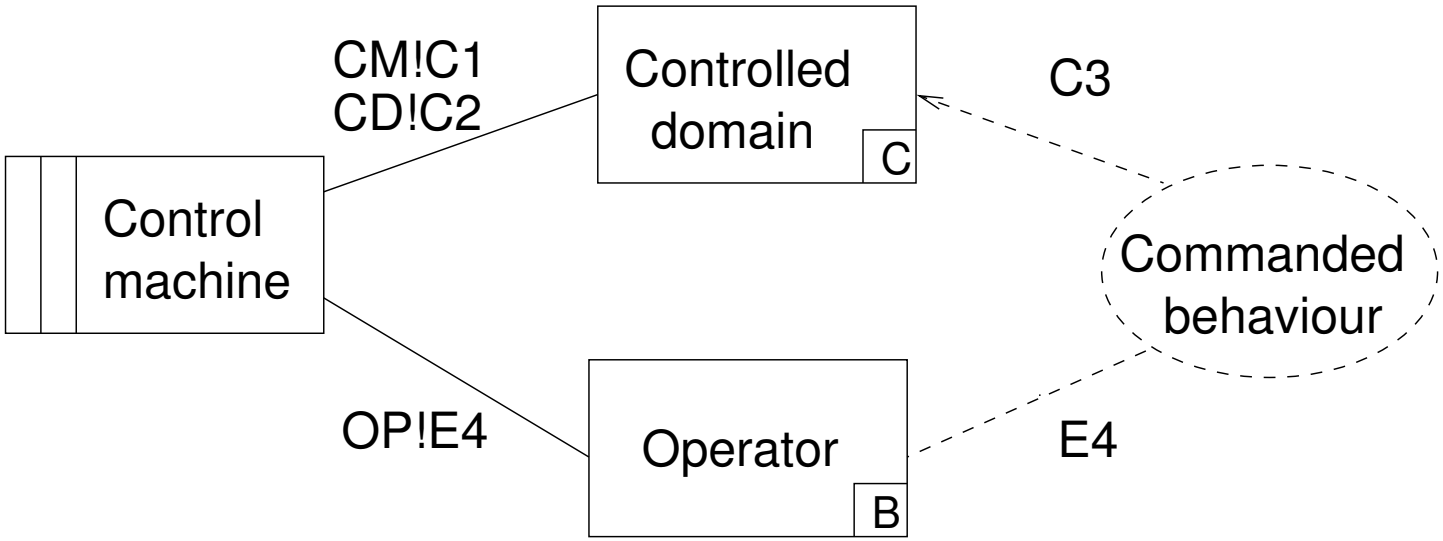
Informal Description

There is some part of the physical world whose behaviour is to be controlled with commands issued by an operator. The problem is to build a machine that will accept the operator's commands and impose the control accordingly.

Difference to "required behaviour": An *operator* can issue commands that influence the behaviour of the controlled domain. Usually operator knows that he/she is an operator

Problem frame: Commanded Behaviour II

Frame Diagram



B: A biddable domain cannot be influenced through a machine
E4: operator commands
Determine how and when the machine should – or should not – cause C1-phenomena in response to E4 commands.

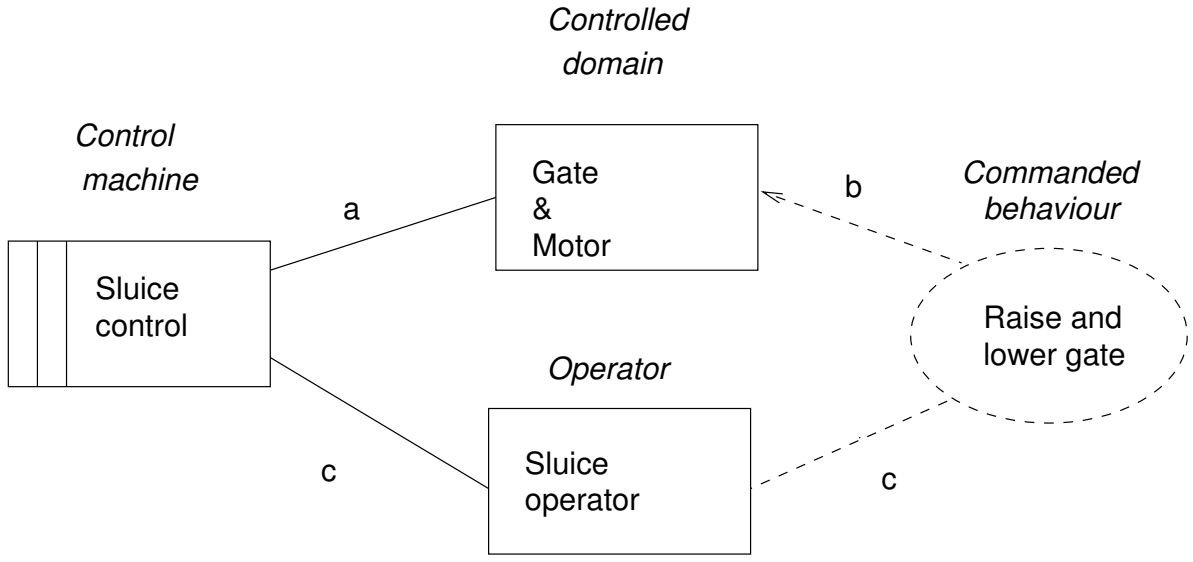
Example: Occasional Sluice Gate Control I

A small sluice is used in a simple irrigation system. The sluice gate can rise and fall. A computer system is needed to raise and lower the sluice gate in response to the commands of the operator.

The gate is open and closed by rotating vertical screws. The screws are driven by a small motor, which can be controlled by clockwise, anticlockwise, on and off pulses. There are sensors at the top and bottom of the gate travel; at the top it's fully open, at the bottom it's fully shut. The connection to the computer consists of four pulse lines for the gate sensor, a status line for each class of operator command.

Example: Occasional Sluice Gate Control II

Fitting the Occasional Sluice Gate Control to the *Commanded Behaviour* frame



a: SC!{Clockw, Anti, On, Off} [C1]
GM!{Top, Bottom} [C2]

b: {Open, Shut, Rising, Falling} [C3]

c: SO!{Raise, Lower, Stop} [E4]

Problems in introducing an operator

- possible that commands cannot be obeyed
  Example: two stop-commands in succession
- possible that commands should not be obeyed
  Example: raise-command when the gate is already at the top of its travel.

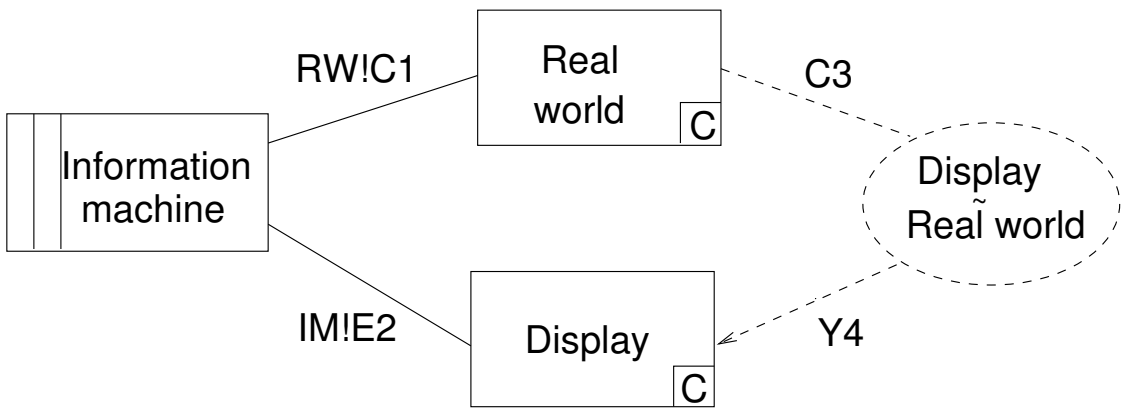
These situations must be taken into account when elicitating the requirements!

Problem frame: Information Display I

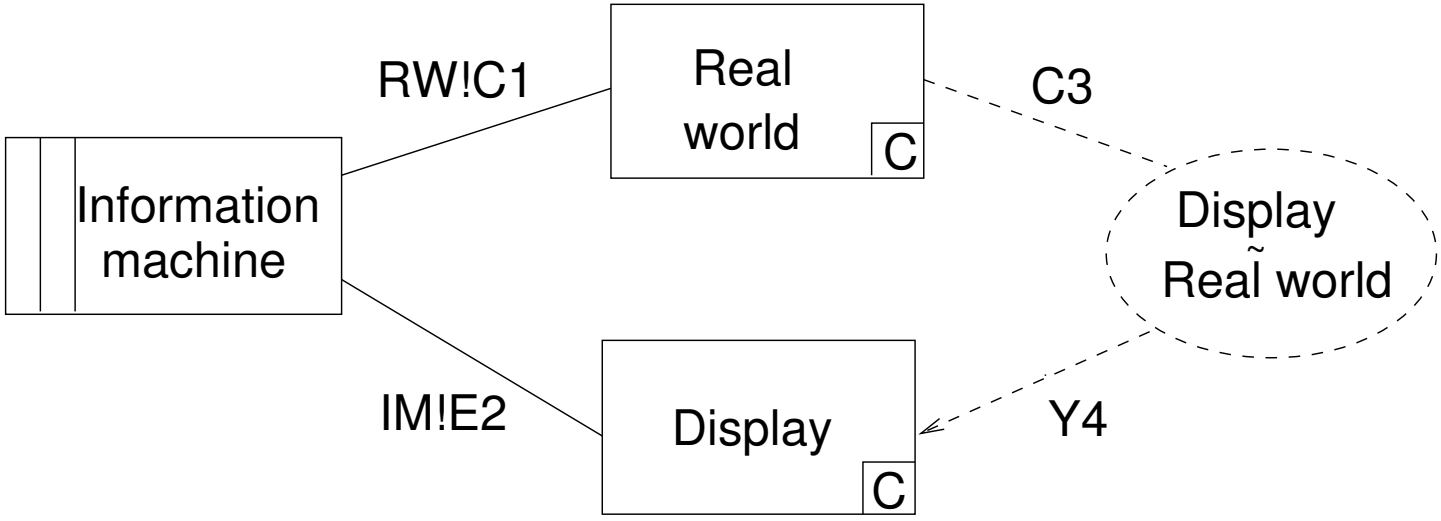
Informal Description

There is some part of the physical world about whose states and behaviour certain information is continually needed. The problem is to build a machine that will obtain this information from the world and present it at the required place in the required form.

Frame diagram



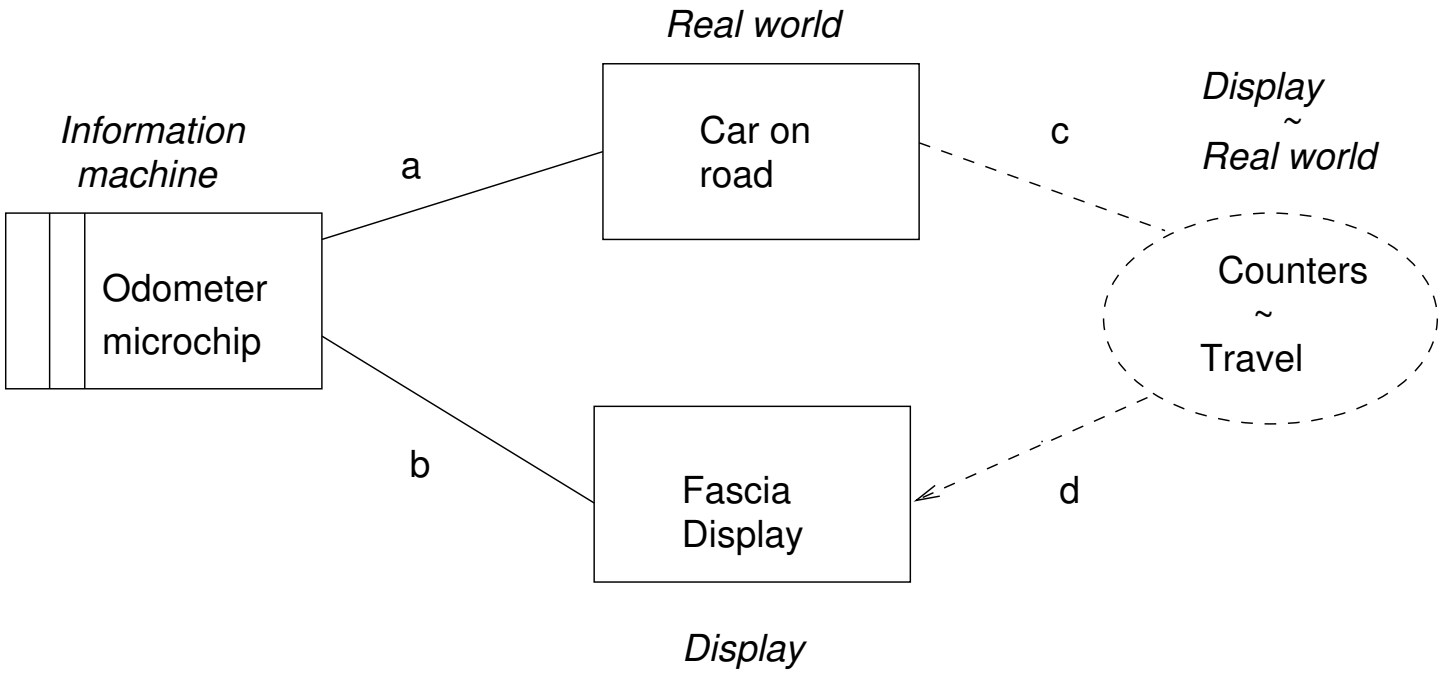
Problem frame: Information Display II



- The real world is entirely autonomous, nothing in the problem context can affect its behaviour.
- The machine must satisfy the requirements by diagnosing phenomena C3 from the phenomena C1.
- It must cause events E2, in order to generate phenomena Y4.

Example: Odometer display

Odometer display fitted to information display frame



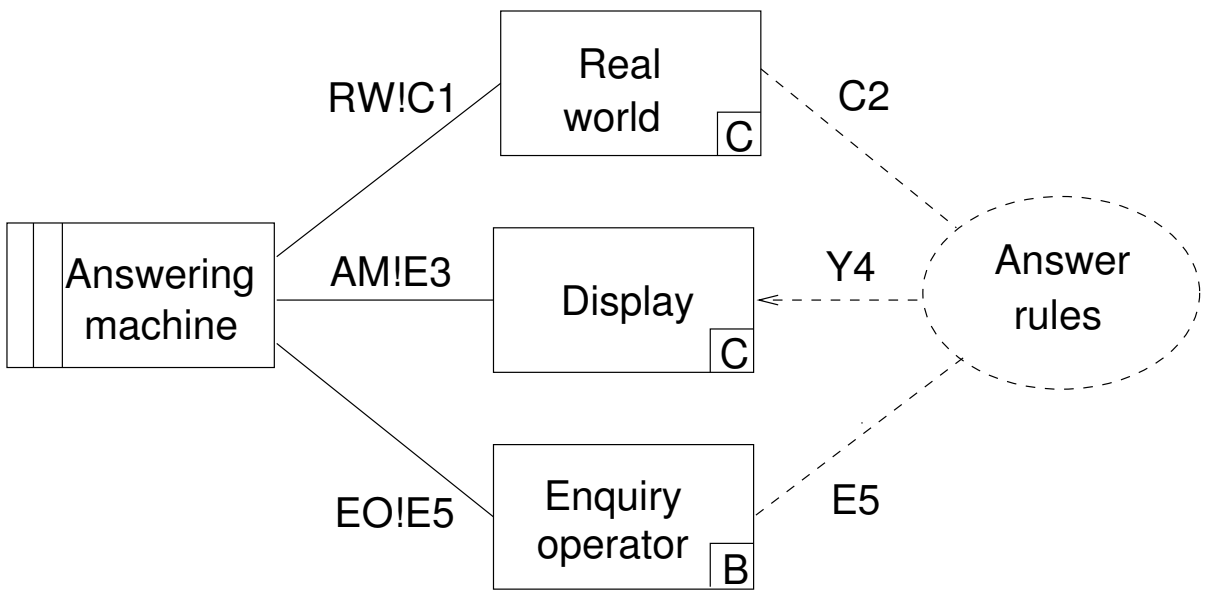
a: CR! {WheelPulse}	[C1]
b: OM! {IncSpeed, IncDist, DecSpeed}	[E2]
c: {Speed, CumDist}	[C3]
d: {SpeedCount, DistCount}	[Y4]

Problem frame: Commanded Information (Simple Information Systems) I

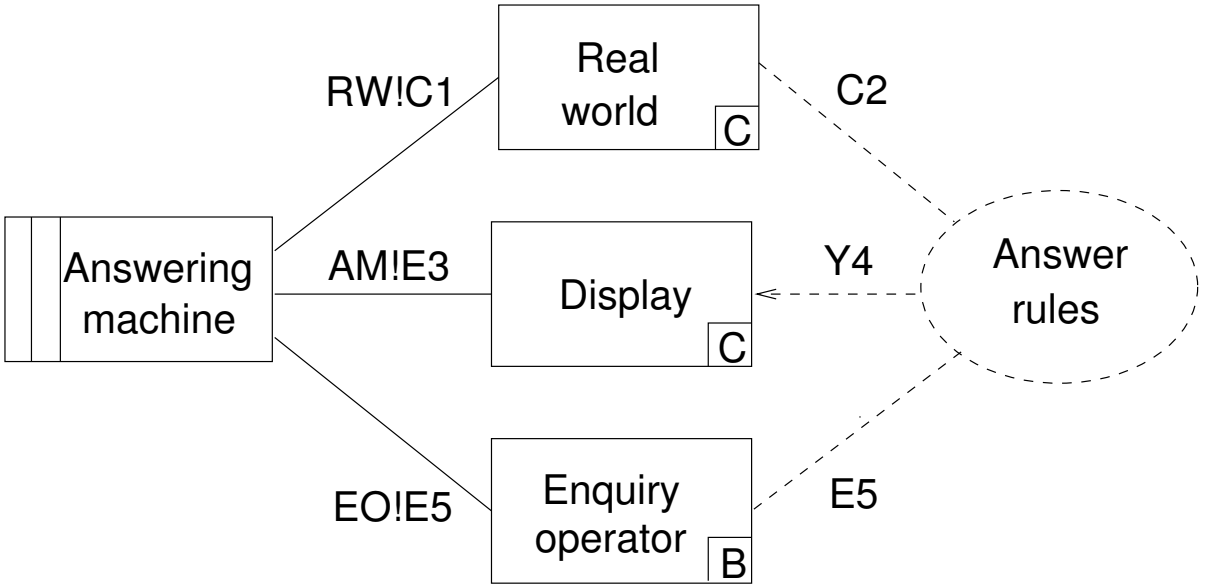
Informal description

A machine is to be built, which answers questions about a domain of the real world.

Frame diagram



Problem frame: Commanded Information (Simple Information Systems) II



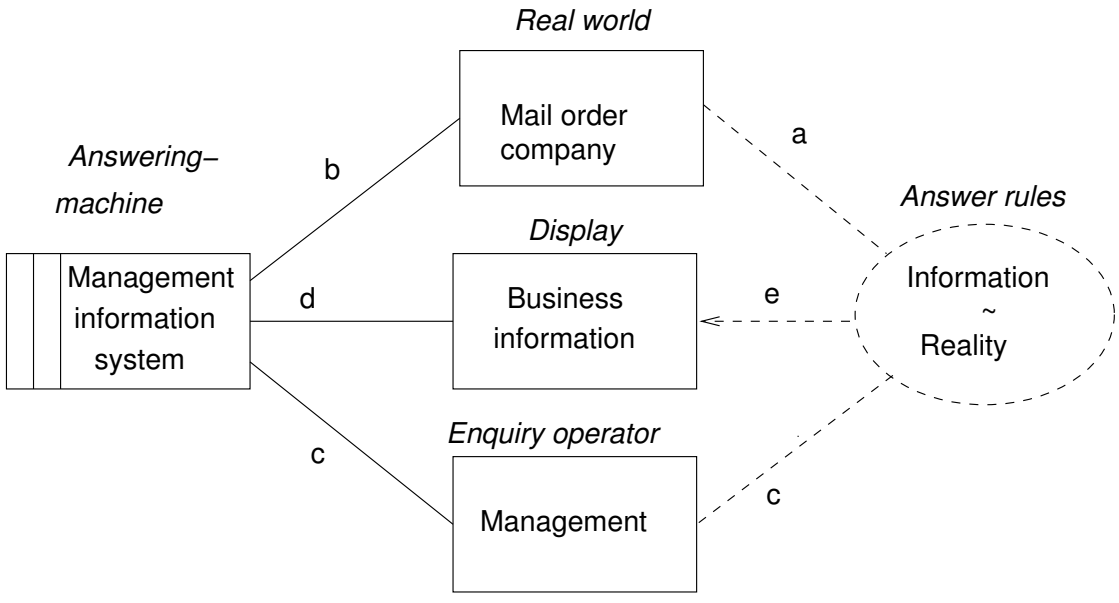
- Enquiries E5 are regarded as commands, shared with the machine
- The requirements stipulate the answers to be produced for each combination of a real world state C2 with an E5 enquiry
- Answers are symbolic phenomena Y4.

Example: Mail Order Company I

Customers can order products from a mail order company. The mail order company can introduce new products and discontinue products that do not sell well. A machine is to be built that provides information to the management on the sales that have taken place. It should be possible to e.g. retrieve information about the goods in stock, how much of a product has been sold in a certain time period, as well as information about the last order placed by a customer.

Example: Mail Order Company II

Mail order company fitted to commanded information frame.



a: {Order, Stock, Customers} [C2]

b: MOC!{StoredOrders, StoredStock, StoredCustomers} [C1]

c: M!{AvailableAmount, SoldAmount, LastOrder} [E5]

d: MIS!{OrderDisplComm, StockDisplComm, CustomerDisplComm} [E3]

e: {OrderDispl, StockDispl, CustomerDispl} [Y4]

Problem frame: Simple Workpieces I

Informal Description

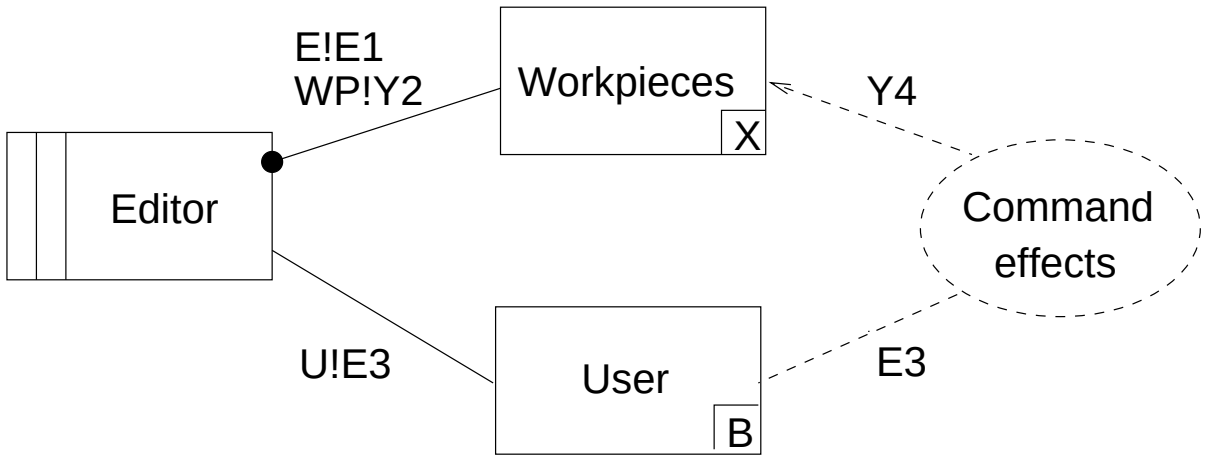
A tool is needed to allow a user to create and edit a certain class of computer-processable text or graphic objects, or similar structures, so that they can be subsequently copied, printed, analysed or used in other ways. The problem is to build a machine that can act as this tool.

Workpieces:
Normally a piece of material worked on by a machine, e.g. wood blocks, metal castings, etc.

Here: things that are worked on by a computer.

Problem frame: Simple Workpieces II

Frame Diagram



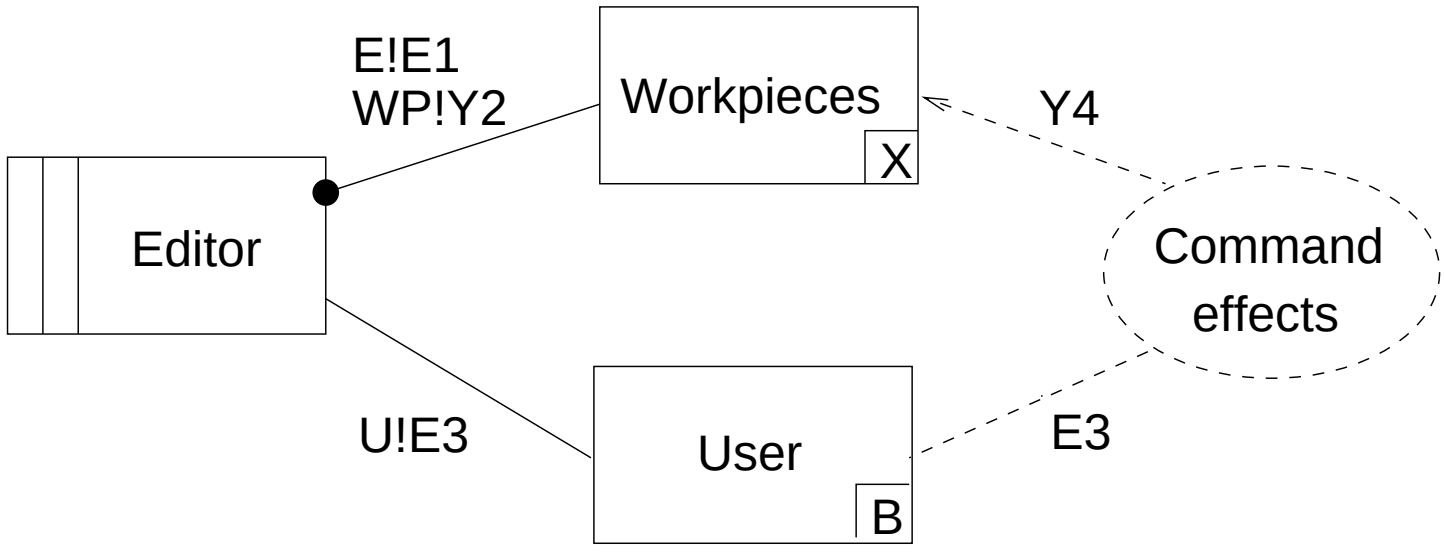
X: Lexical Domain, i.e. data type

Phenomena E1: *Operations* on workpieces

Phenomena Y2: allow the machine to examine the current state and values of the workpiece.

Workpieces domain is contained in the machine, indicated by "•".

Problem frame: Simple Workpieces III

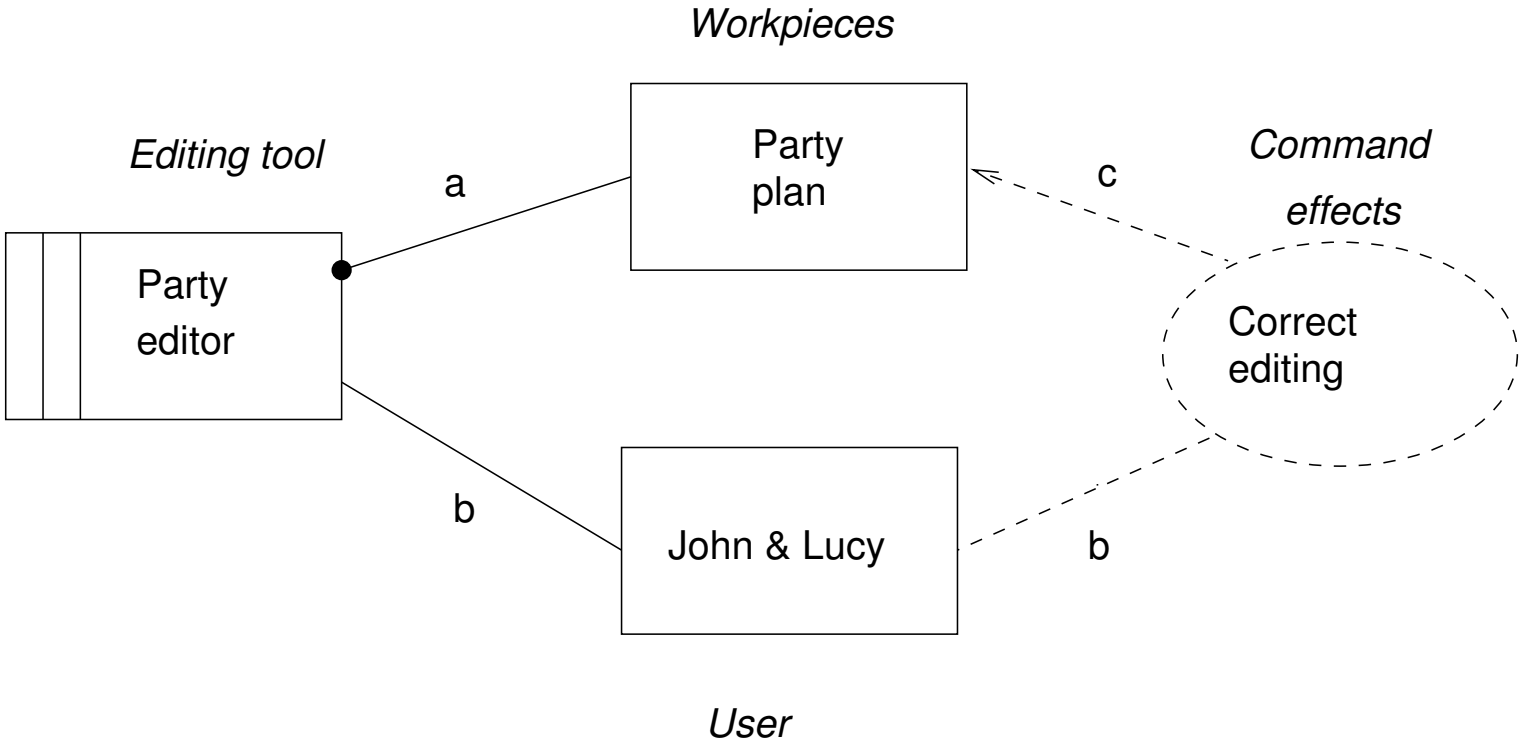


Phenomena E3: user commands
Requirements: describe effects, commands E3 should have on phenomena Y4.
Y4 often differs from Y2: Phenomena Y4 can have some meaning to the user that is insignificant for editing.
Simple workpieces does not deal with printing, representing or further processing the workpieces.

Example: Party plan I

Lucy and John need a system to keep track of the many parties they give and the many guests they invite. They want a simple editor to maintain the information, which they call their *party plan*. Essentially the party plan is just a list of parties, a list of guests, and a note of who is invited to each party. The editor will accept command-line text input.

Example: Party plan II



a: PE!{PlanOperations} [E1]
PP!{PlanStates} [Y2]
b: JL!{Commands} [E3]
c:{PlanEffects} [Y4]

Remark

The structure of the problem frames simple workpieces and commanded behaviour are very similar!
Differences:

- The Workpieces domain is lexical and formal, but the Controlled domain is causal and informal
- In the Controlled domain, approximations must be taken into account, which is unnecessary for the Workpieces domain
- The requirements in the simple workpieces frame relate only to the user's commands, while in the commanded behaviour frame other requirements can occur, e.g. that the sluice gate is not driven beyond the limits of its vertical travel.

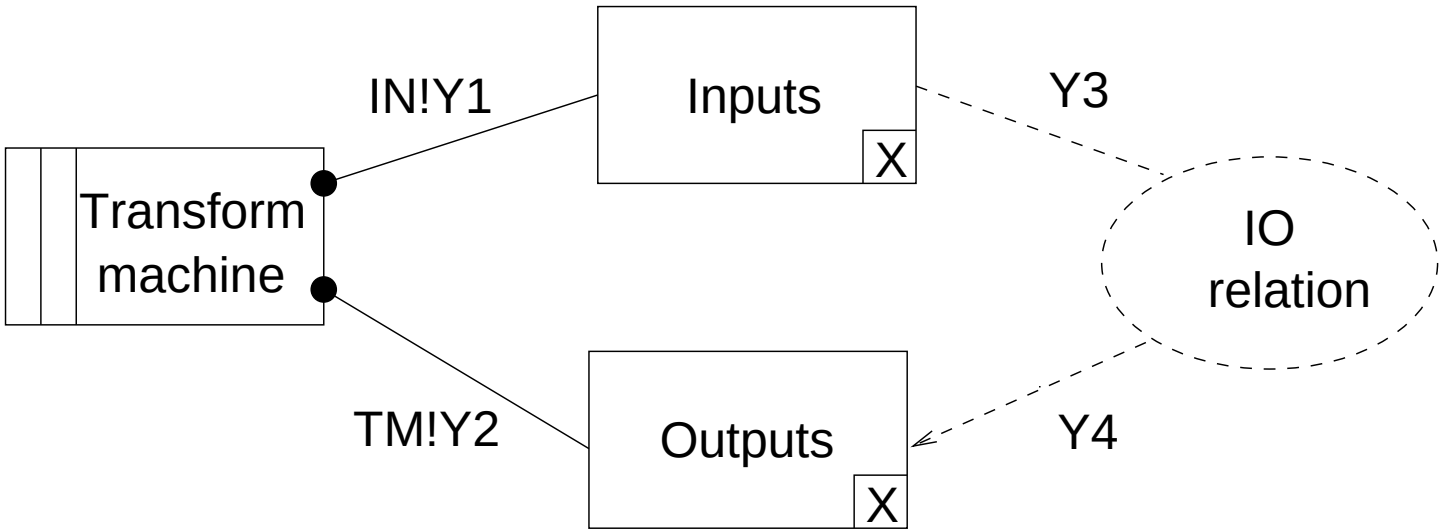
Problem frame: Transformation I

Informal Description

There are some given computer-readable input files whose data must be transformed to give certain required output files. The output data must be in a particular format, and it must be derived from the input data according to certain rules. The problem is to build a machine that will produce the required outputs from the inputs.

Problem frame: Transformation II

Frame diagram



The input cannot be changed, and is not restricted through requirements.
Y1 and Y3, as well as Y2 and Y4 may differ, but do not have to.

Example: Mailfiles analysis I

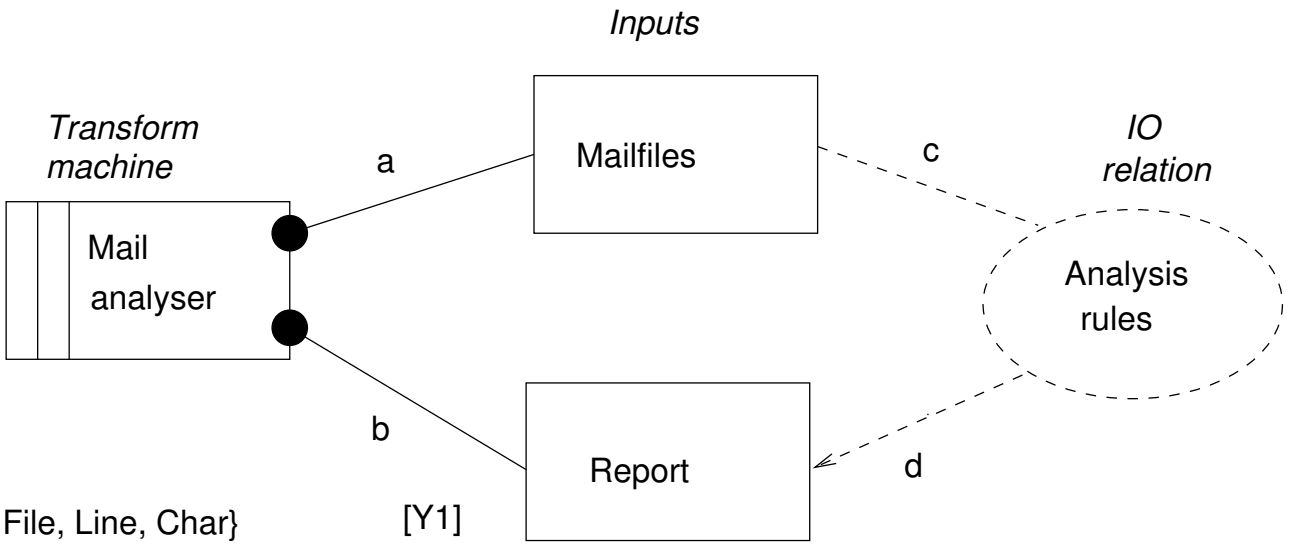
John wants to analyse his email-correspondence. He is interested in the average number of messages he receives and sends in a week, the average and maximum message length, and similar things. After some thought he has worked out that he wants a report that looks like this:

Name	# In	Max.Length	∅ Length	# Out	Max.Length	∅ Length
Albert	19	52136	6027	17	21941	2123
Anna	31	13248	1736	37	34763	2918
...	...	...	...	...	...	...

The report contains a line for each of his correspondents.

Example: Mailfiles analysis II

Mailfiles analysis fitted into transformation frame diagram



a:MF! {MsgDir, File, Line, Char} [Y1]

b: MA! {ReportLine, Char} [Y2]

c: {Msg, From, To, Date, Lenght} [Y3]

d: {LineData} [Y4]

Summary

- Simple problems that occur during problem decomposition should belong to known problem classes, which have common solution methods.
- These kinds of problem classes are characterized through *problem frames*. Problem frames contain patterns for the domains involved and their shared phenomena. They also define which domains the requirements refer to and which domains they restrict.
- In order to solve a subproblem with a method that is associated with a problem frame, the subproblem needs to be "fitted" to the frame by instantiating the frame diagram.

Decomposition

- Realistic problems must be divided into (simple) subproblems.
  A useful way of decomposition is *projection*, i.e., ignoring certain aspects.
  This kind of decomposition adds to other types of decomposition, such as top-down or Use Cases.
- The problem frames help to find suitable subproblems.
- If a problem does not fit into one of the problem frames, a new problem diagram can be developed.

Decomposition operations

To fit a subproblem into a problem frame, the following operations can be applied on the context diagram:

- Leave out domains (with corresponding interfaces)
- Combine several domains into one domain
- Divide a domain
- Introduce a connection domain
- Reduce interface between domains
- Refine phenomena
- Combine (i.e., abstract) phenomena

Subproblem relationships

The following relationships between subproblems can be identified and help to compose the solution:

- *Parallel* subproblems are largely independent of each other, and the global machine will have to treat the problems in parallel. If the same domain is contrained in parallel subproblems, *priorities* should be assigned. These priorities must be consistant with the priorities in the requirements. They must be updated if necessary.
- *Sequential* subproblems have to be treated one after another.
- *Alternative* problems are exclusive. Only one of them will have to be treated at a given time.

Subproblem relationships can be expressed using a context-free grammar in in so-called Backus-Naur Form (BNF). The BNF can be extended with a symbol for parallel subproblems.

Context-free grammars, Backus-Naur-Form

- A context-free defines a language, i.e., a set sequences of (terminal) symbols.
- A BNF specification is a set of production rules, written as $< NTS >::=<$ *expression with symbols* $>$.
- $< NTS >$ is a nonterminal symbol.
- The expression consists of sequences of symbols, where "|" indicates alternatives.
- The expression describes possible substitution for the symbol on the left.
- The language consists of all sequences of terminal symbols that can be derived using the production rules, starting from a specified start symbol.

Subproblem relationship notation example

The dependencies between the subproblems can be summarized using a context-free grammar describing the possible sequences. In the following grammar, "||" denotes parallel problems and "|" denotes an alternative.

$$
\begin{array}{lcl}
< \textit{start} > & ::= & < \textit{main\_passing} > || < \textit{fire} > || < \textit{broken\_light} > \\
< \textit{main\_passing} > & ::= & \textit{MainRoadPassing} < \textit{sec\_passing} > \\
< \textit{sec\_passing} > & ::= & \textit{SecondaryRoadPassing} < \textit{main\_passing} > \\
< \textit{fire} > & ::= & \textit{EmergencyRequest} < \textit{main\_passing} > \\
< \textit{broken\_light} > & ::= & \textit{BrokenLightSafeState}
\end{array}
$$

In this grammar, the terminal symbols *MainRoadPassing*, *SecondaryRoadPassing*, *EmergencyRequestSecondaryRoadPassing* and *BrokenLightSafeState* are all the subproblems of the traffic light control problem.

Subproblem relationship notation remarks

- We always start with the nonterminal $< start >$.
- The subproblems correspond to terminal symbols of the grammar.
- The nonterminals ($< main\_passing >$, $< sec\_passing >$, $< fire >$, and $< broken\_light >$) may correspond to states. These states form the conditions the the previous subproblem establishes and the succeeding subproblem requires.

Phase 3: Decompose problem

input:	requirements R to be implemented of Phase 2	natural language
	domain knowledge D of Phase 2	natural language
	context diagram of Phase 2	ext. Jackson
output:	set of problem diagrams with associated set of requirements	Jackson with dot-notation
	expression of the subproblem relationships	grammar
validation:	consistent with context diagram of Phase 2	
	requirements R of Phase 2 must be treated in at least one subproblem	

Executing Phase 3

To decompose the problem the following steps have to be performed:

- Identify subproblems that should fit to problem frames.
- Set up the correspondig problem diagrams. Try to assign each requirement to exactly one subproblem.
- Add connection domains (possibly part of the machine) according to facts and assumptions.
- Introduce domain knowledge for connection domains.
- Describe the relationships between the subproblems.

Example 1: traffic light control

Requirements for traffic lights I

R1 : When there is a car waiting on the secondary road, the traffic lights should stop the flow of traffic on the main road for a period of time and allow the traffic flow on the secondary road.

R2 : As long as the emergency button is activated, the flow of traffic on the main road should be stopped and the flow of traffic on the secondary road should be allowed.

R3 : Vehicles on the main road should be allowed to pass the crossing for a longer period of time than from the secondary road (if not emergency-case[4]).

R4 : While vehicles on one road are allowed to pass, the others should be stopped.

Requirements for traffic lights II

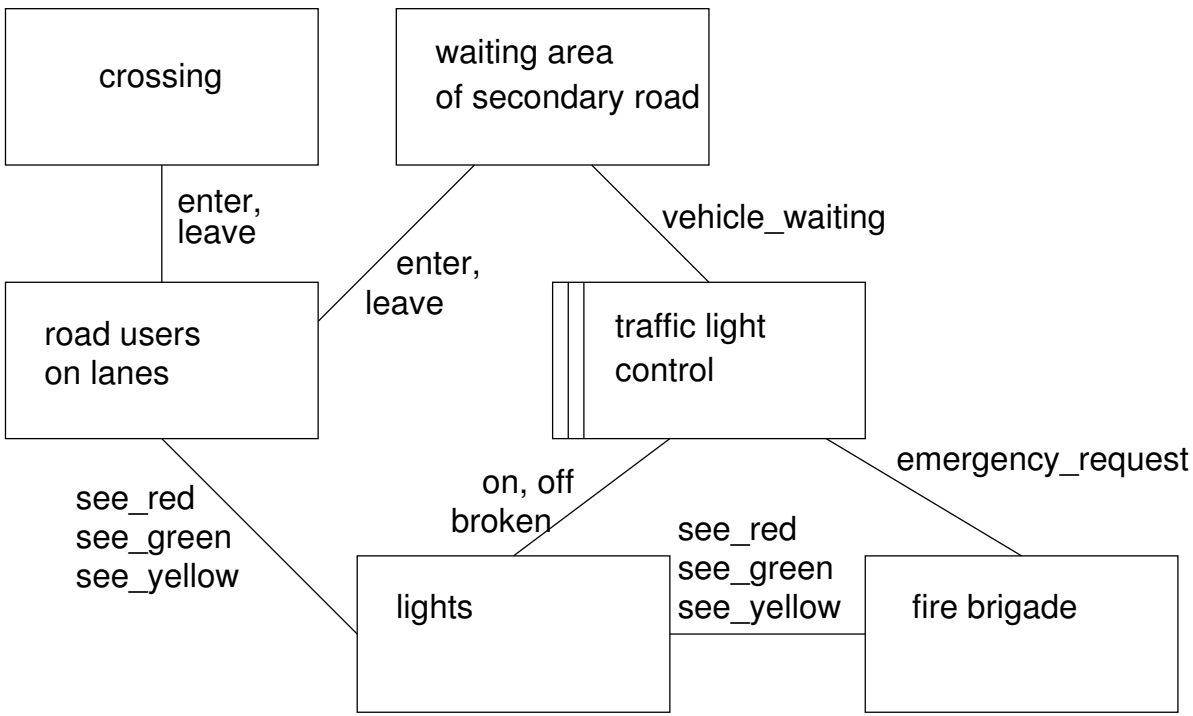
R5 : The lights should switch in the following order: red - red+yellow - green - yellow - red. Other combinations (except "all off", yellow blinking, and green - yellow - green in emergency case[5]) are not allowed.

R6 : In case of a broken light bulb the traffic lights should blink in yellow for the secondary road, after all red lights have been switched on for a period of time.

R7 : After switching to red, the traffic flow of both roads should be stopped for a period of time *[6].

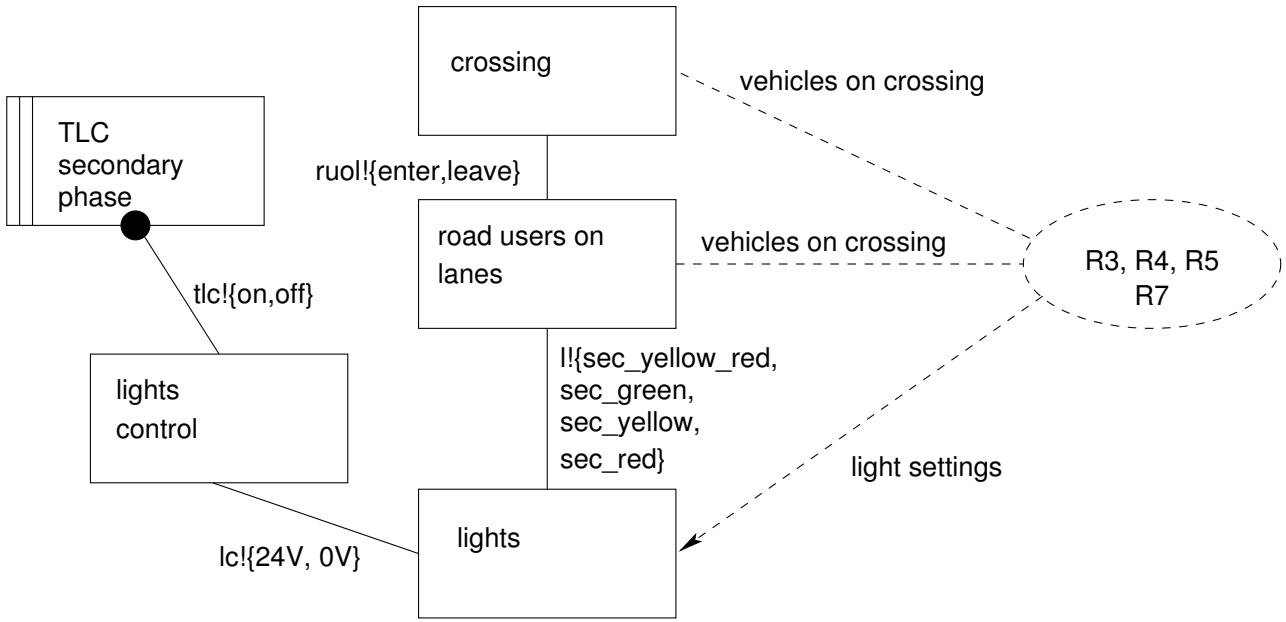
[4] Added later to eliminate contradictions
[5] Added later to eliminate contradictions
[6] A star (*) denotes: added later

Context diagram for traffic lights



SecondaryRoadPassing Problem Diagram I

- Variant of the *required behavior* problem frame. (no operator, lights domain is controlled, additional domains without a direct connection to the machine exist)
- Because of F10 it is not possible to connect lights and the machine directly (different voltage). Therefore, the connection domain *lights control* (being part of the machine) must be introduced.



SecondaryRoadPassing Problem Diagram II

In this diagram only the parts of the requirements concerning the secondary phase are considered.
The following projection operators have been applied:

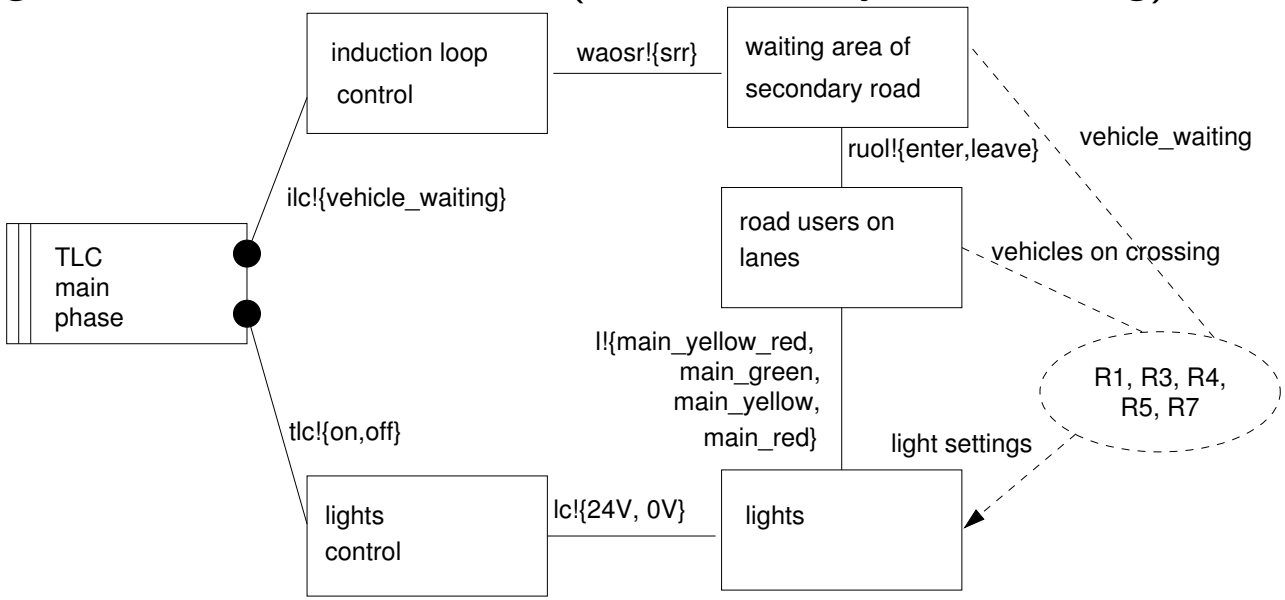
- The domains *waiting area of secondary road*, *fire brigade*, and the corresponding interfaces are left out.
- The connection domain *light control* is introduced. The phenomena of the context diagram (*on*, *off*) are used between the machine *TLC secondary phase* and the connection domain.
- The interface between machine and *lights* domain is reduced (dropping the phenomenon *broken*).

SecondaryRoadPassing Problem Diagram III

- The interface between *road users on lanes* and *lights* domain is refined (e.g., see_green $\Rightarrow$ *sec_green* or *main_green*) and reduced (e.g., dropping *main_green*; it only contains the phenomena relevant for the secondary road passing phase).

MainRoadPassing Problem Diagram I

- Variant of the *required behavior* problem frame.
- To detect if a road user is on the *waiting area* an additional connection domain is necessary (F9). The connection domain *induction loop control* is introduced.
- *Lights control* is introduced (see SecondaryRoadPassing).



In this diagram only the parts of the requirements for the secondary phase are considered.

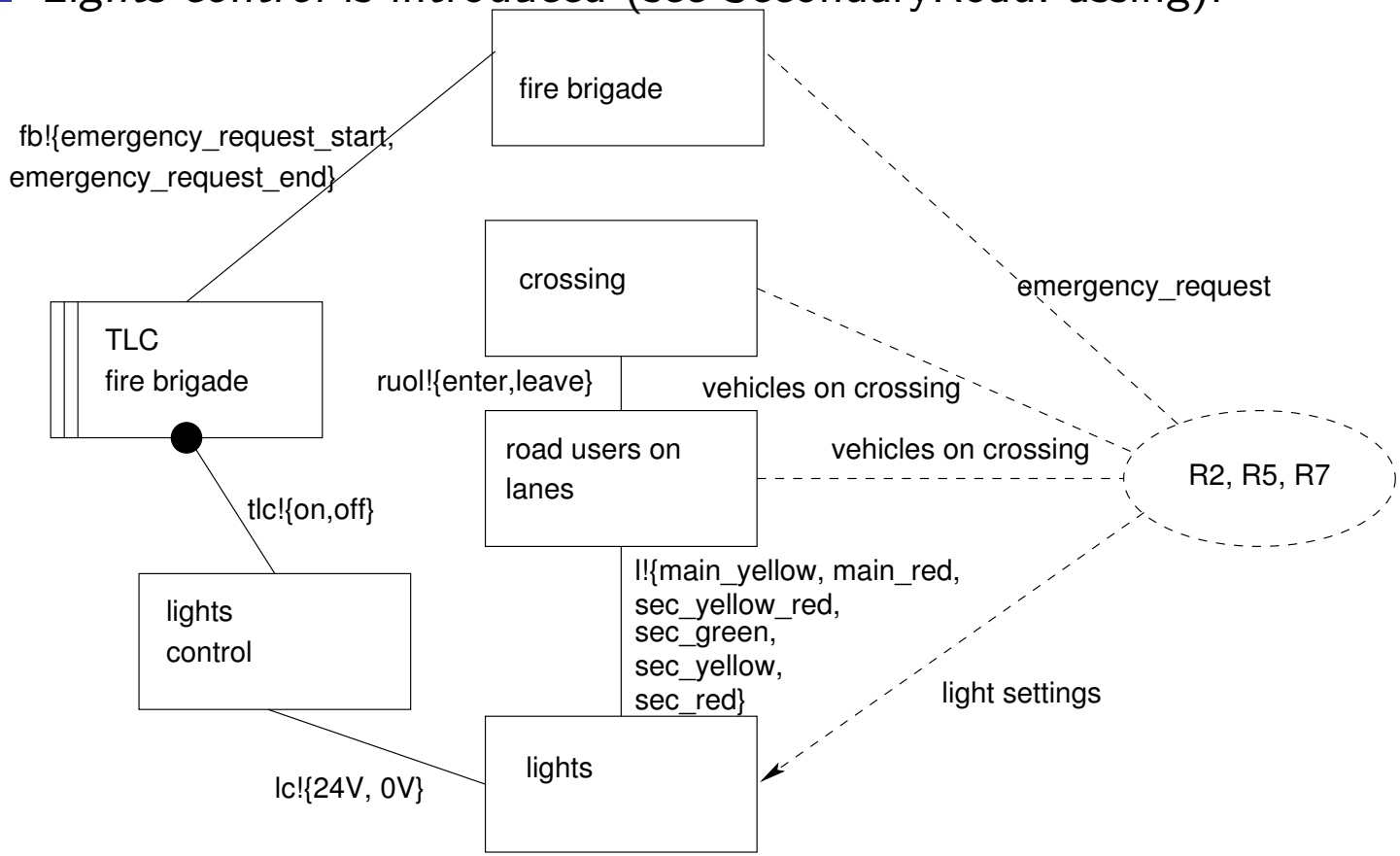
MainRoadPassing Problem Diagram II

The following projection operators have been applied:

- The domains *crossing*, and *fire brigade*, and the corresponding interfaces are left out.
- The interface between machine and *lights* domain is reduced (dropping the phenomenon *broken*).
- The interface between *road users on lanes* and *lights* domain is refined (e.g., see_red $\Rightarrow$ *sec_red* or *main_red*) and reduced (e.g., dropping *sec_green*; it only contains the phenomena relevant for the main road passing phase).

EmergencyRequest Problem Diagram I

- Variant of the *commanded behavior* problem frame. The domain *fire brigade* is the operator (biddable domain) in the problem frame.
- *Lights control* is introduced (see SecondaryRoadPassing).



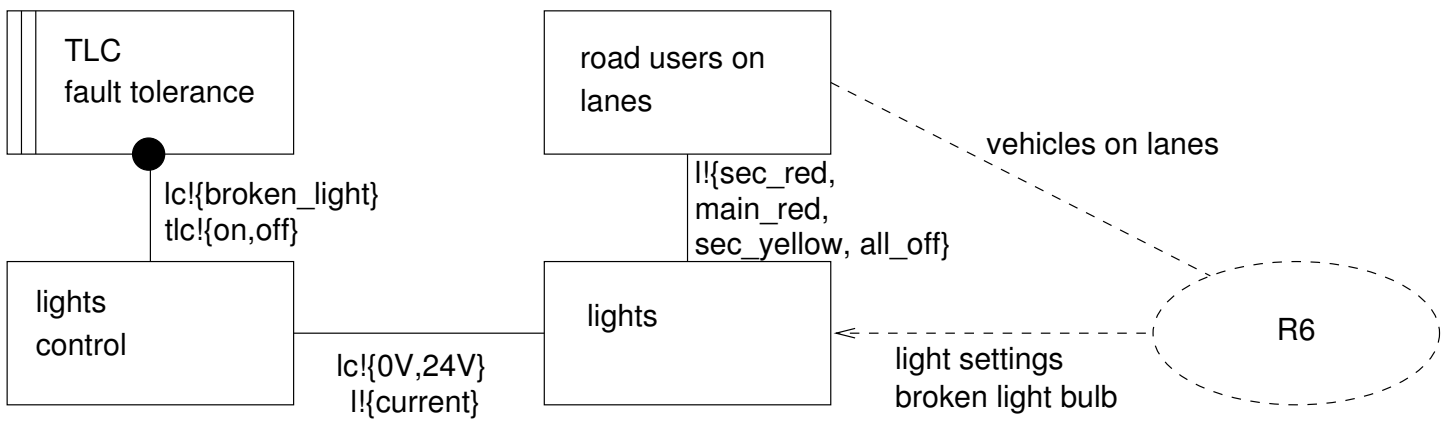
EmergencyRequest Problem Diagram II

The following projection operators have been applied:

- *Lights control* is introduced (see SecondaryRoadPassing).
- The interface between machine and *lights* domain is reduced (dropping the phenomenon *broken*).
- The interface between *road users on lanes* and *lights* domain is refined (e.g., see_green $\Rightarrow$ *sec_green* or *main_green*) and reduced (e.g., dropping *main_green*; it only contains the phenomena relevant for the emergency phase).
- The phenomenon *emergency_request* is refined into *emergency_request_start* and *emergency_request_stop*.

BrokenLightSafeState Problem Diagram I

- Variant of the required behavior problem frame.
- The connection domain *lights control* is also necessary to detect broken lights by measurement of the power consumption ($F7$).



BrokenLightSafeState Problem Diagram II

The following projection operators have been applied:

- The domains *crossing*, *waiting area of secondary road*, *fire brigade*, and the corresponding interfaces are left out.
- The interface between *road users on lanes* and *lights* domain is refined (e.g., see_yellow $\Rightarrow$ *sec_yellow* or *main_yellow*) and reduced (e.g., dropping *main_green*; it only contains the phenomena relevant for the broken phase).
- The domain *lights control* with its phenomena (*0V*, *24V*, *current*) is introduced. The electric current is used to detect a broken light bulb ($F7$).

Problem Diagram relationships

$$
\begin{array}{lcl}
< \textit{start} > & ::= & < \textit{main\_passing} > || < \textit{fire} > || < \textit{broken\_light} > \\
< \textit{main\_passing} > & ::= & \textit{MainRoadPassing} < \textit{sec\_passing} > \\
< \textit{sec\_passing} > & ::= & \textit{SecondaryRoadPassing} < \textit{main\_passing} > \\
< \textit{fire} > & ::= & \textit{EmergencyRequest} < \textit{main\_passing} > \\
< \textit{broken\_light} > & ::= & \textit{BrokenLightSafeState}
\end{array}
$$

- Once activated, the subproblem *EmergencyRequest* has priority. That implies, only the machine for *EmergencyRequest* is allowed to control the lights until it gives the control to the machine for *MainRoadPassing*.
- The subproblem *BrokenLightSafeState* acts in parallel to the subproblems *EmergencyRequest*, *MainRoadPassing*, and *SecondaryRoadPassing*. It has a higher priority than all the other subproblems: Once activated, only the machine for *BrokenLightSafeState* is allowed to control the lights.

Validation

The problem diagrams are consistent with the context diagram:

- The problem diagrams were derived from the context diagram by applying the introduced operators.
- All phenomena and all domains of the context diagram are covered.

Example 2: sun blind control

Requirements for sun blind I

R1 If there is *sunshine* for more than one minute but *no heavy wind*, the sun blind will be lowered (*lower sun blind*). *(Parts of R3 included to prevent contradictions)*

R2 If there is *no sun shine* for more than 5 minutes, the sun blind will be pulled up (*pull up sun blind*).

R3 The sun blind should not be destroyed: If there is *heavy wind*, the sun blind will be pulled up (*pull up sun blind*).

Requirements for sun blind II

R4 If there is *no heavy wind* and the user *manually adjusts the fins with positive degree* the fins are rotated with positive degree (*rotate fins with positive degree*).
If there is *no heavy wind* and the user *manually adjusts the fins with negative degree* the fins are rotated with negative degree (*rotate fins with negative degree*).

R5 If the user *manually opens the sun blind*, the sun blind will be pulled up (*pull up sun blind*).

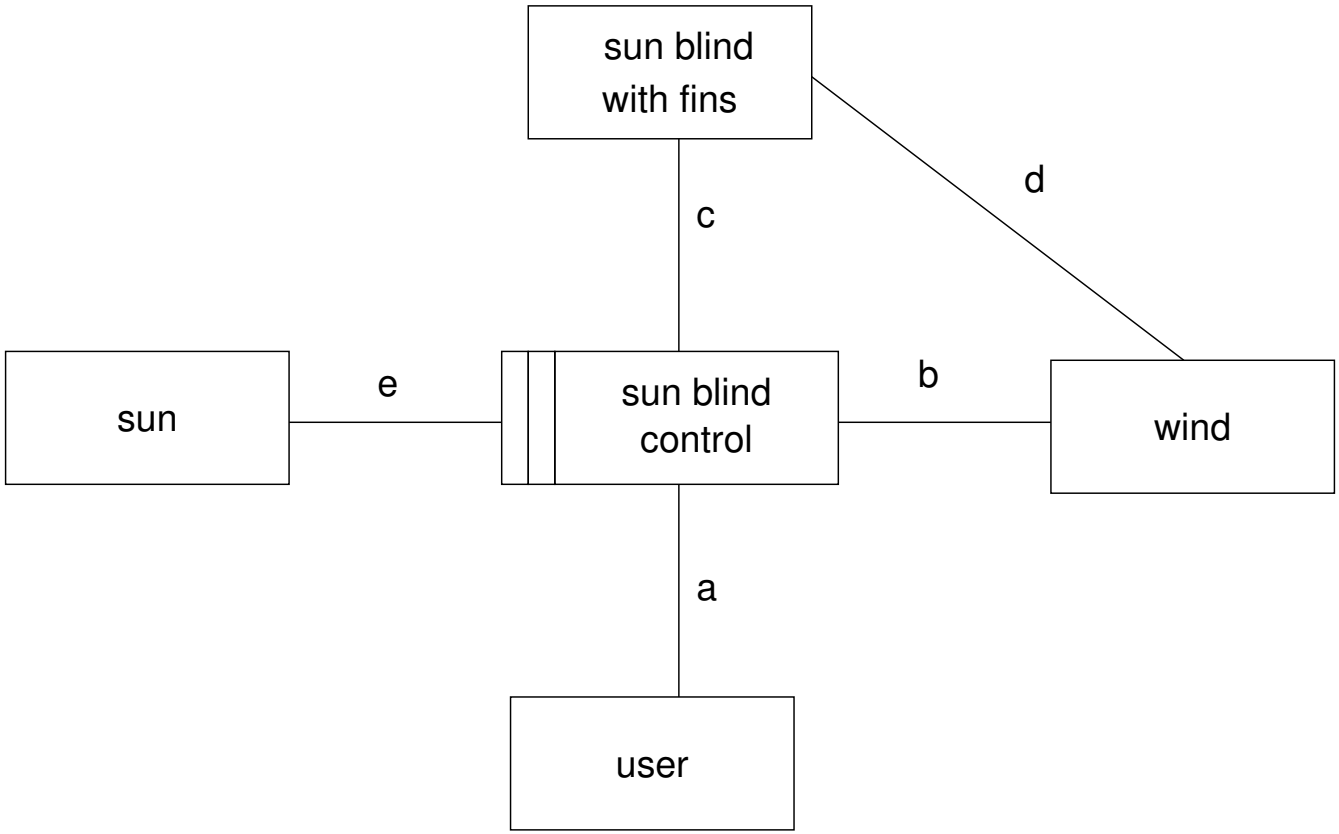
R6 If the user *manually closes the sun blind* and there is *no heavy wind*, the sun blind will be lowered (*lower sun blind*).
(Parts of R3 included to prevent contradictions.)

Requirements for sun blind III

R7 The sun blind should not be destroyed: When the sun blind is in its lowest position (*sun blind is lowered*) or in it highest position (*sun blind is pulled up*) the sun blind should stop (*stop sun blind*).

R8 If the user interacts with the sun blind (*manually opens the sun blind*, *manually closes the sun blind*, *rotate fins with positive degree* or *rotate fins with negative degree*), *sun shine* and *no sun shine* are ignored within the next 4 hours.

Context Diagram for sun blind I



a: manually adjust fins with negative degree, manually adjust fins with positive degree, manually open sun blind, manually close sun blind, stop closing sun blind, stop opening sun blind

Context Diagram for sun blind II

b: heavy wind, no heavy wind

c: lower sun blind, pull up sun blind, stop sun blind, sun blind is lowered, sun blind is pulled up, rotate fins with positive degree, rotate fins with negative degree

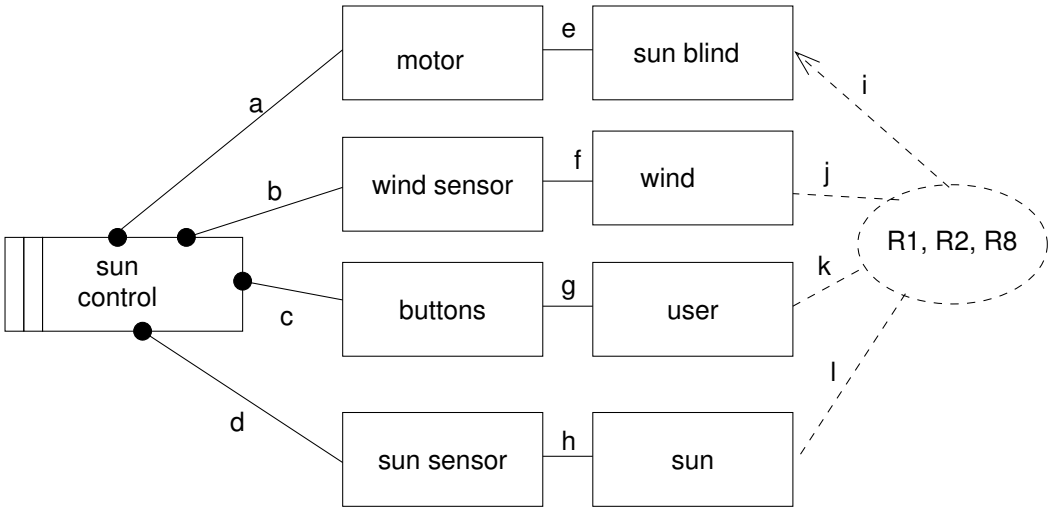
d: destroy sun blind

e: sunshine, no sunshine

SunControl Problem Diagram I

- Variant of the *commanded behavior* problem frame. The domain *user* is the operator (biddable domain). The domains *wind*, *sun*, and *sun blind* are the the *controlled domains*.
- Because of $A1$, $A2$, and $A3$ (description of the user interactions) the connection domain *button* (being part of the machine) must be introduced.
- $F1$ and $F2$ show that a *sun sensor* is necessary to measure the intensity.

SunControl Problem Diagram II



a: SC!{turn motor right, turn motor left}
b: WS!{intensity of wind}
c: B!{up-button pushed, down-button pushed}
d: SS!{intensity of sun shine}
e: M!{lower sun blind, pull up sun blind}
f: W!{heavy wind, no heavy wind}
g: U!{manually open sun blind, stop closing sun blind, manually close sun blind, stop opening sun blind, manually adjust fins with negative degree, manually adjust fins with positive degree}
h: S!{sunshine, no sunshine}
i: "heavy wind"
j: "user input"
k: "weather conditions"
l: "position of sun blind"

SunControl Problem Diagram III

The following projection operators have been applied:

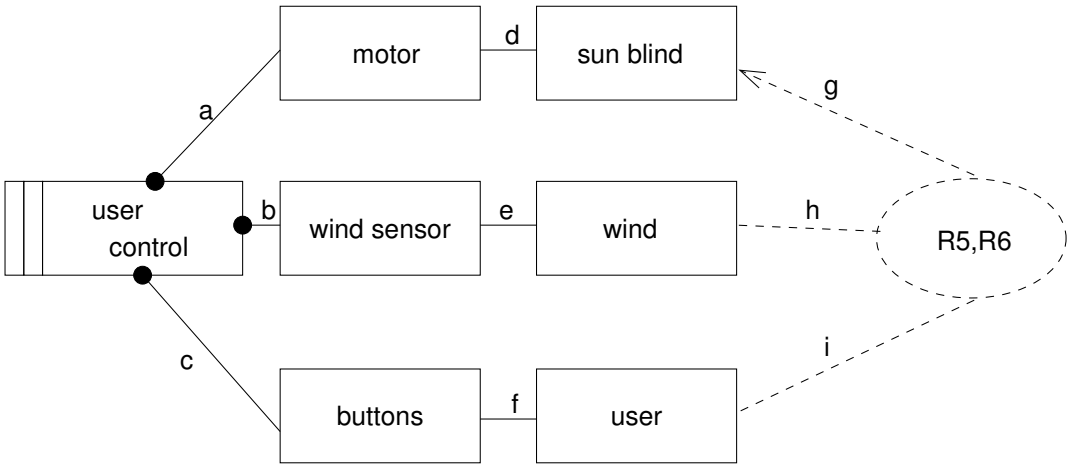
- The domain *sun blind with fins* is split and the domain *fins* is left out with the corresponding phenomena.
- The connection domains *motor*, *wind sensor*, *buttons* and *sun sensor* are introduced with additional phenomena.
- The phenomena *rotate fins with positive degree*, *rotate fins with negative degree* are dropped.
- The phenomena dealing with pulled up or lowered sunblind and blocked motor are dropped.

All phenomena of the user are included since *sun_shine* and *no_sunshine* are ignored for 4 hours in if one of the user phenomena occurs.

UserControl Problem Diagram I

- Variant of the *commanded behavior* problem frame. The domain *user* is the biddable domain. The domains *wind* and *sun blind* are *controlled domains*.
- Because of $A1$, $A2$, and $A3$ (description of the user interactions) the connection domain *button* (being part of the machine) must be introduced.
- $F3$, $F4$, $F11$, and $F13$ show that a *motor* is necessary to control the sun blind (conection domain). (domin knowledge added in Phase 1)

UserControl Problem Diagram II



a: UC!{turn motor right, turn motor left}

b: WS!{intensity of wind}

c: B!{up-button pushed, up-button released, down-button pushed, down-button released}

d: M!{lower sun blind, pull up sun blind}

e: W!{heavy wind, no heavy wind}

f: U!{manually open sun blind, stop closing sun blind, manually close sun blind, stop opening sun blind}

g: "position of sun blind"

h: "heavy wind"

i: "user input"

UserControl Problem Diagram III

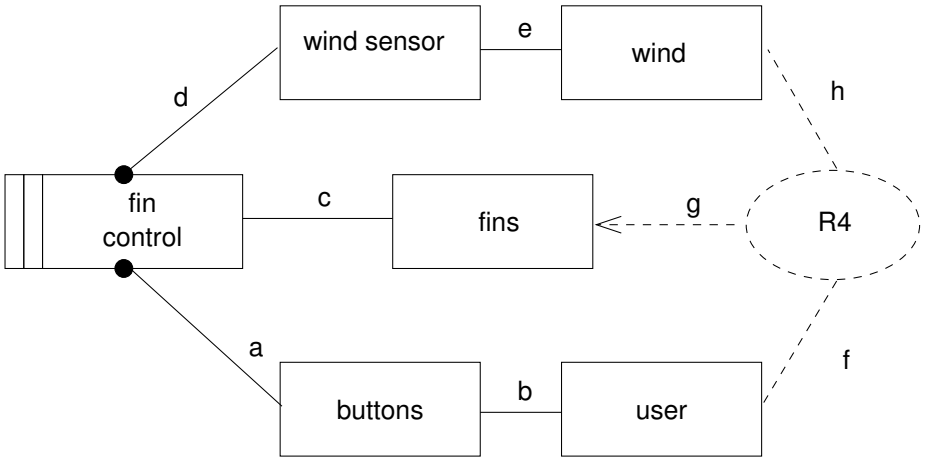
The following projection operators have been applied:

- The domain *sun blind with fins* is split and the domain *fins* is left out with the corresponding phenomena.
- The domain *sun* is left out with the corresponding phenomena.
- The connection domains *buttons* and *motor* and necessary phenomena are introduced.
- The phenomena concerning the fins are dropped in the user interface.
- The phenomena dealing with pulled up or lowered sunblind and blocked motor are dropped.

FinsControl Problem Diagram I

- Variant of the *commanded behavior* problem frame. The domain *user* is the biddable domain. The domains *wind* and *fins* are *controlled domains*.
- Because of $A1$, $A2$, and $A3$ (description of the user interactions) the connection domain *button* (being part of the machine) must be introduced.
- $F6$ shows that a *wind sensor* is necessary to measure the speed of the wind.
- The fins have an interface that can be directly connected to a microcontroller ($F12$ added).

FinsControl Problem Diagram II



a: B!{up-button pushed, up-button released, down-button pushed, down-button released}

b: U!{manually adjust fins with negative degree, manually adjust fins with positive degree}

c: FC!{rotate fins with positive degree, rotate fins with negative degree}

d: WS!{intensity of wind}

e: W!{heavy wind, no heavy wind}

f: "user input"

g: "fins position"

h: "wind strength"

The following projection operators have been applied:

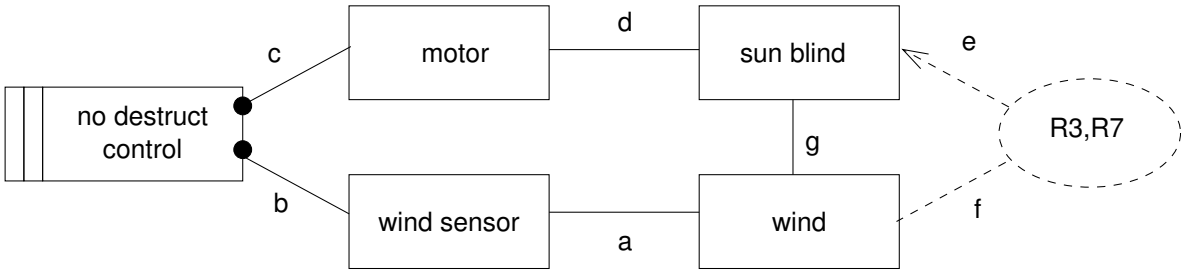
FinsControl Problem Diagram III

- The domain *sun blind with fins* is split and the domain *sun blind* is left out with the corresponding phenomena.
- The domain *sun* is left out with the corresponding phenomena.
- The connection domains *buttons* and *wind sensor* and necessary phenomena are introduced.
- The phenomena *manually open sun blind*, *stop closing sun blind*, *manually close sun blind*, *stop opening sun blind*, *manually adjust fins with negative degree*, *manually adjust fins with positive degree* are dropped.

NoDestruct Problem Diagram I

- Variant of the *required behavior* problem frame. The domains *wind* and *sun blind* are *controlled domain*.
- $F6$ shows, that a *wind sensor* is necessary to measure the speed of the wind.
- $F3$, $F4$ and $F11$ show, that a *motor* is necessary to control the sun blind. Additionally, the *motor* must be able to inform the machine that the sun blind is blocked ($F10$). When the *sun blind is pulled up* or the *sun blind is lowered*, the motor is blocked.

NoDestruct Problem Diagram II



a: W!{heavy wind, no heavy wind}

b: WS!{intensity of wind}

c: NDC!{turn motor right, turn motor left, stop motor}, M!{turn right is blocked, turn left is blocked}

d: M!{lower sun blind, pull up sun blind, stop sun blind}, SB!{sun blind is lowered, sun blind is pulled up}

e: "position of sun blind"

f: "heavy wind"

g: W!{detroy sunblind}

NoDestruct Problem Diagram III

The following projection operators have been applied:

- The domain *sun blind with fins* is split and the domain *fins* is left out with the corresponding phenomena.
- The domains *sun* and *user* are left outwith the corresponding phenomena.
- The connection domains *motor* and *wind sensor* and necessary phenomena are introduced.

Problem Diagram relationships

All supproblems are parallel.

¡start¿ ::= SunControl || NoDestructControl || FinsControl || UserControl

- *NoDestructControl* should have the highest priority.
- *UserControl* should have a higher priority than *SunControl*.
- No priority must be assigned to *FinsControl* since a different domain (*fins*, not *sun blind*) is contrained.

Validation

- Usually, the phenomena in the problem diagrams the same as in the context diagram. Only when connection domains are introduced, new phenomena have been introduced.
- Usually, the domains in the problem diagrams the same as in the context diagram. Only connection domains are introduced.
- All requirements of Phase 2 are captured.

Phase 4: Derive machine behavior specification for each subproblem P_i

...

3. Decompose problem
4. Derive machine behavior specification for each subproblem
5. Design global system architecture
6. Derive specifications for all components of the global system architecture
7. Design a software architecture for all components of the global system architecture that should be implemented in software

...

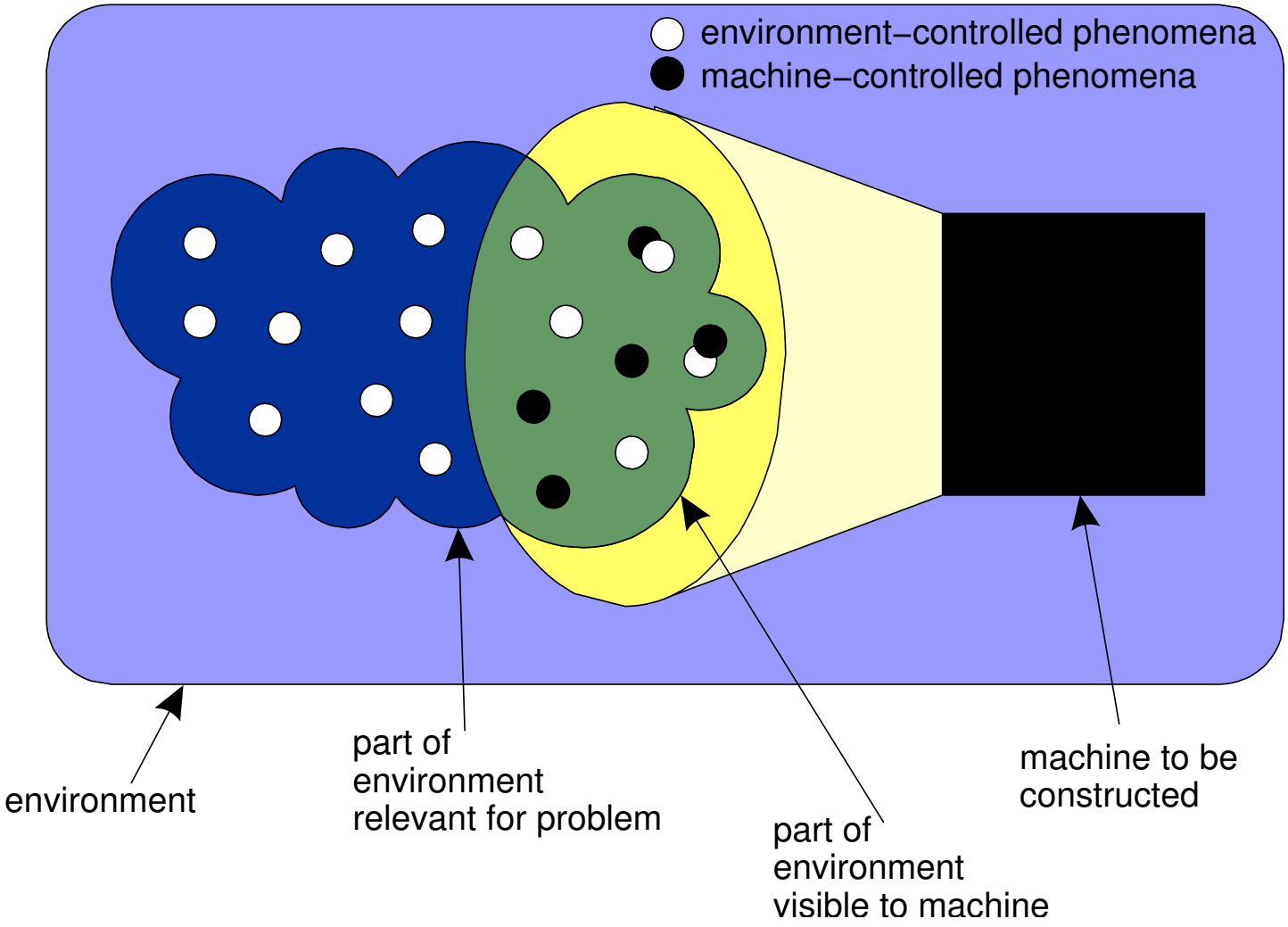
Phase 4: Derive machine behavior specification for each subproblem P_i

input:	requirements R from Phase 2	natural language
	domain knowledge D from Phase 2	natural language
	problem diagram for P_i from Phase 3	Jackson with dot-notation
output:	specification S_{P_i} of machine to construct	natural language
	sequences of interactions with annotated states for the domains in the environment, expressing R_{P_i} and D_{P_i}	sequence diagrams with annotated states
	sequences of interactions on initialization	sequence diagram with annotated states
validation:	$D \wedge S_{P_i}$ are non-contradictory	
	$D \wedge S_{P_i} \Longrightarrow R_{P_i}$	
	all requirements must be captured	
	in the sequence diagrams refined phenomena of the problem diagrams are used as signals	
	direction of signals must be consistent with control of shared phenomena	
	signals must connect domains as connected in problem diagram	
	the relationships of Phase 3 must be consistent with the states	

Notations and concepts

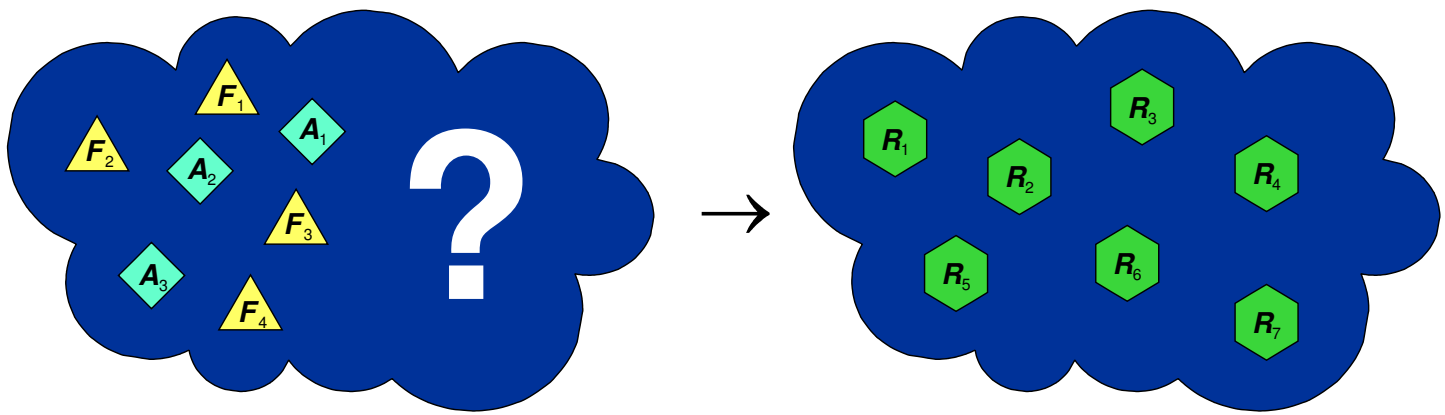
- Terminology
- Specifications
- UML sequence diagrams

Terminology



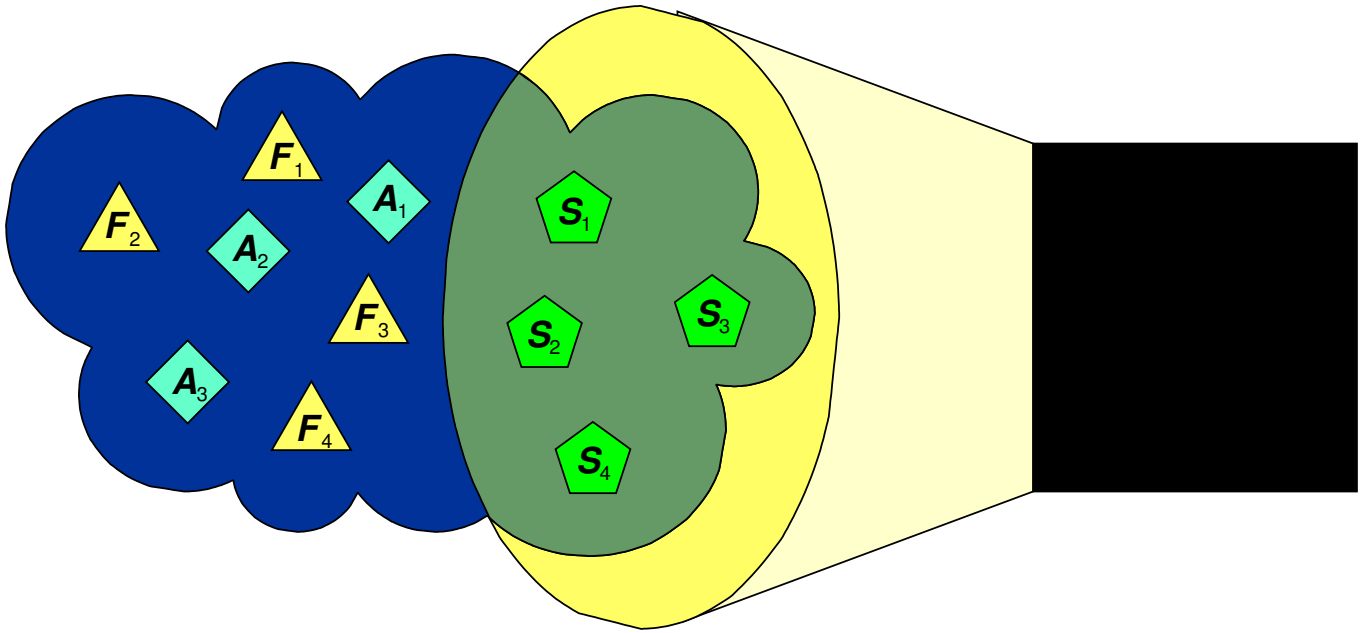
Deriving specifications from requirements I

- Searched: specification of the machine
- Known: facts F, assumptions A and requirements R



- Question: How do you get the specification S, such that $F \wedge A \wedge S \Rightarrow R$ (correctness of the specification)

Deriving specifications from requirements II



Specifications vs. requirements

In contrast to the requirements, the *specification* of the machine gives an answer to the question: "How should the machine act, so that the system fulfills the requirements?" Specifications are descriptions that are sufficient for building the machine. Specifications are implementable requirements. To derive the specification:

- Replace phenomena not observable / controlled by the machine by observable / machine controlled phenomena that are *related* to the requirements phenomena.

A negative Example: Airbus Accident in Warsaw

Requirement: $reverse\_thrust\_allowed \Leftrightarrow plane\_landed$
Reasoning:

$$plane\_landed \Leftrightarrow wheels\_turn\_fast \quad \in F$$

$$wheels\_turn\_fast \Leftrightarrow pulses \quad \in F$$

Conclusion:

$$reverse\_thrust\_allowed \Leftrightarrow pulses \quad \in S$$

Problem: aquaplaning!
Result: plane refused reverse thrust during landing when raining

Correctness criteria

Consider requirements R, facts F, assumptions A, specifications S.

1. Each element of R is considered acceptable by the client, and R contains all customer preferences concerning the software development project.
2. Each element of F was checked for correctness.
3. Each element of A was checked for plausibility.
4. All elements of S are implementable.
5. $F \wedge A \wedge S \Rightarrow R$ is demonstrated.
6. It is proved that F, A and S are consistent.

When these criteria are fulfilled, then the construction of a machine that fulfills S can be started.

Sequence diagrams

UML Superstructures Specification:

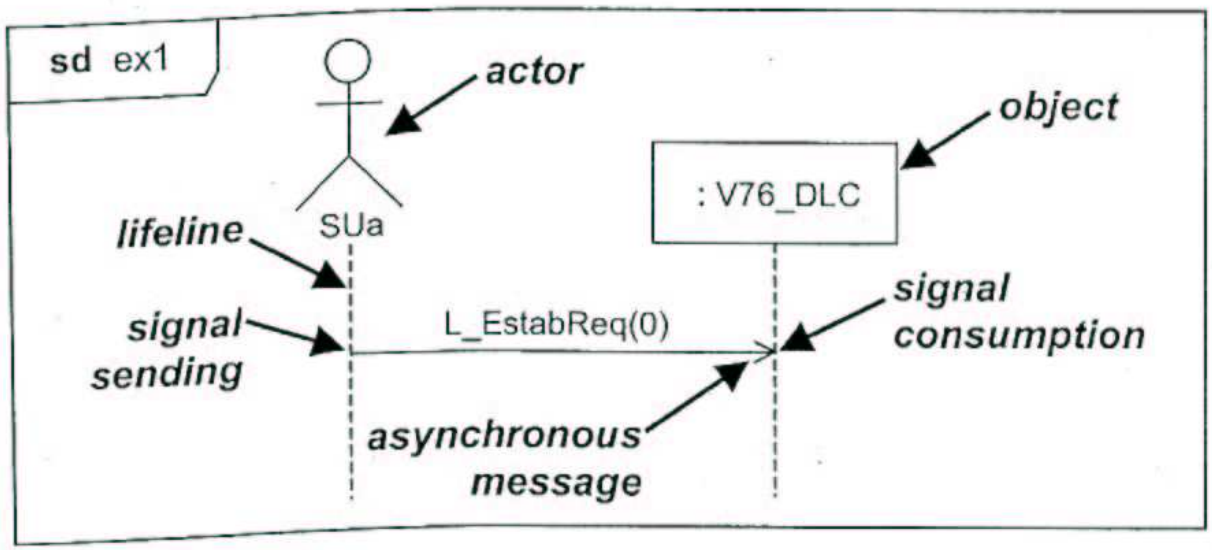
> *A sequence diagram describes an Interaction by focusing on the sequence of Messages that are exchanged, along with their corresponding OccurrenceSpecifications on the Lifelines. An OccurrenceSpecification is the basic semantic unit of Interactions. The sequences of occurrences specified by them are the meanings of Interactions. OccurrenceSpecifications are ordered along a Lifeline.*

Literature:

- Laurent Doldi: *UML 2.0 Illustrated.* TMSO, 2003. http://www.tmso-systems.com
- M. Jeckle, C. Rupp, J. Hahn, B. Zengler, S. Queins: *UML 2 glasklar.* Hanser, 2004.

Basic Elements

Each sequence diagram has a name and a bounding box.



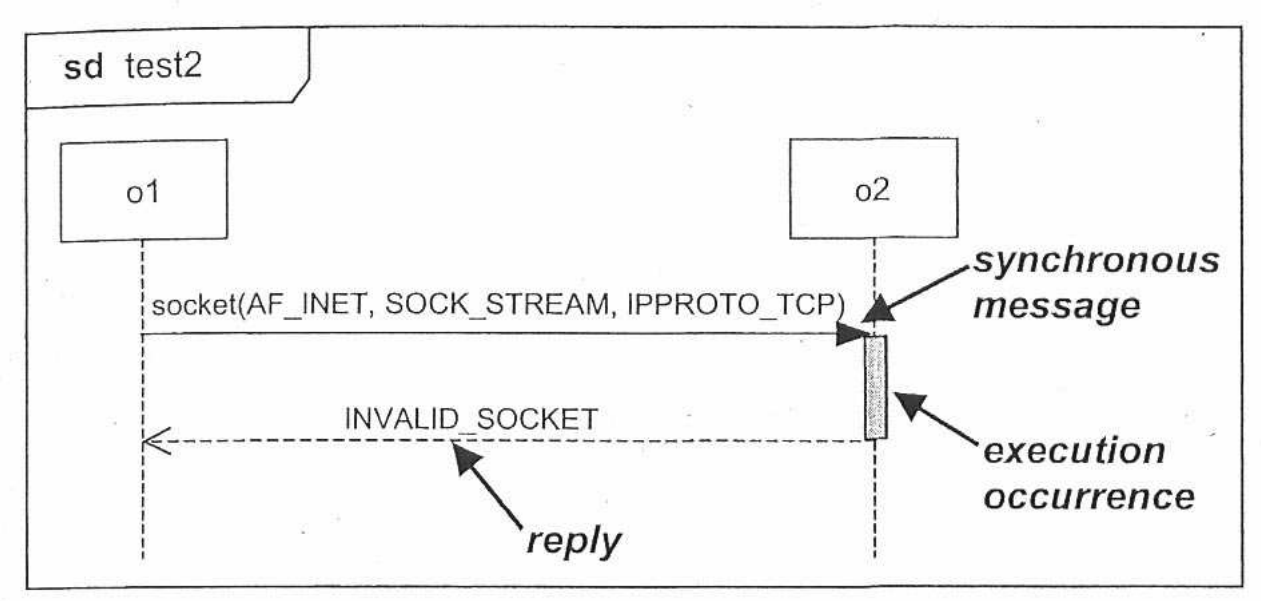
Objects are not underlined.
(This and the following pictures are taken from Doldi, 2003.)

Asynchronous vs. synchronous messages

Asynchronous messages First, the sending of the message occurs, and after a certain (variable) amount of time, the message is consumed by the receiver. The sender does not wait for the receiver's reply but continues its process. Example: mailbox.

Synchronous messages Sending and reception of the message take place at (almost) the same time. The sender waits for reply message before resuming its process. Example: function call.

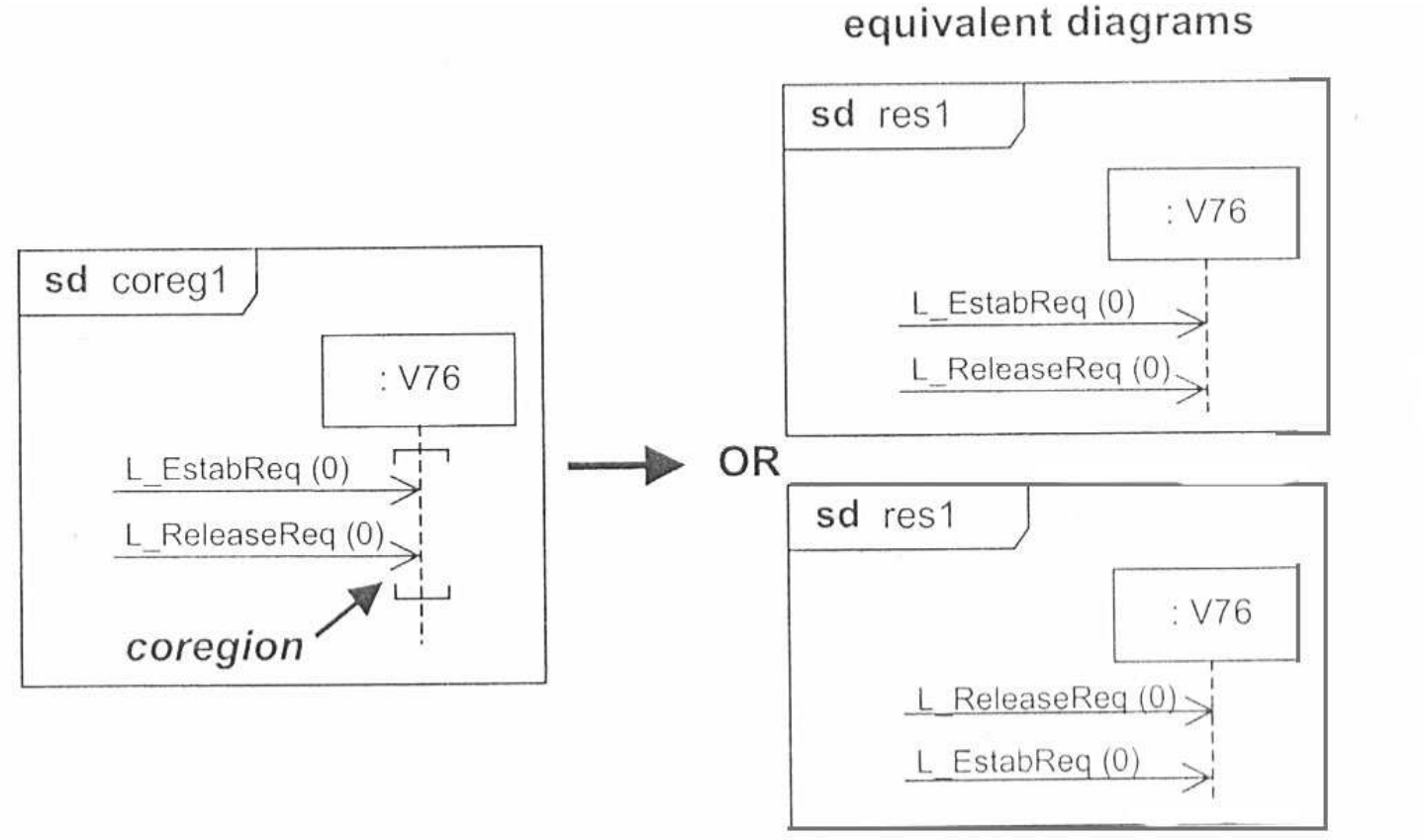
Synchronous messages



Gray box represents execution of actions.
The different kinds of messages are indicated by different arrowheads.

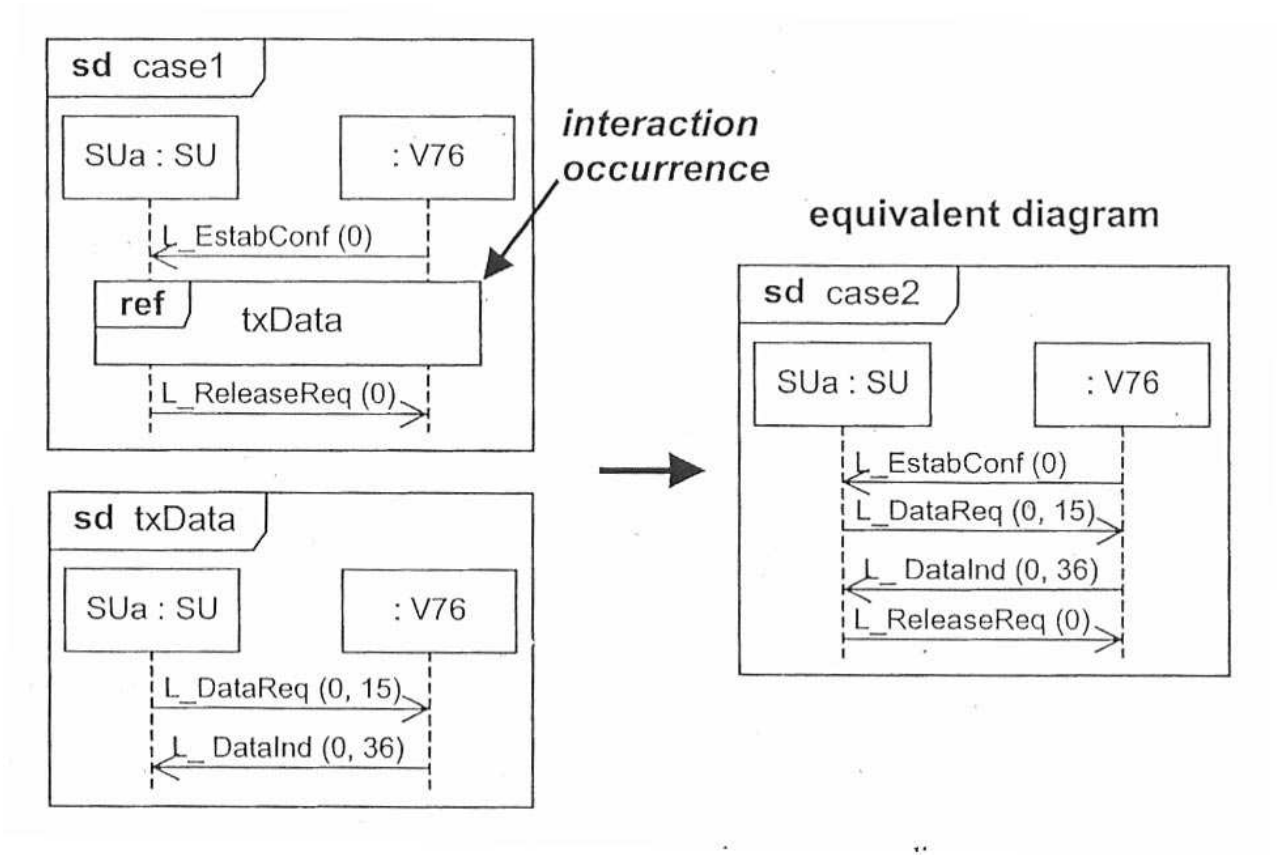
Co-region

Messages can be received in any order.



Reference

Enables reuse of diagrams. Semantics: replace reference by its contents.

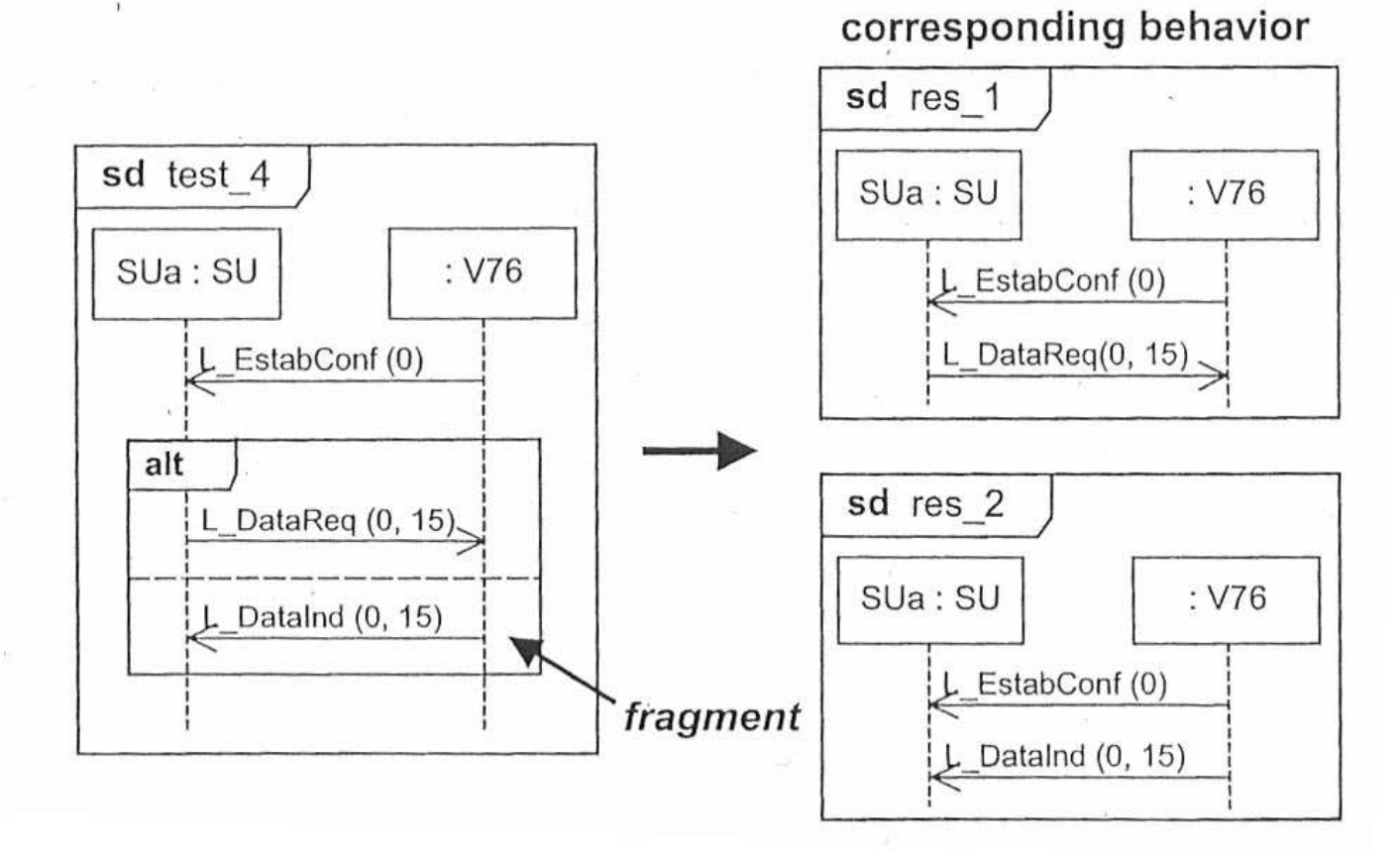


Combined Fragments

Operators for combining different sequence diagrams:

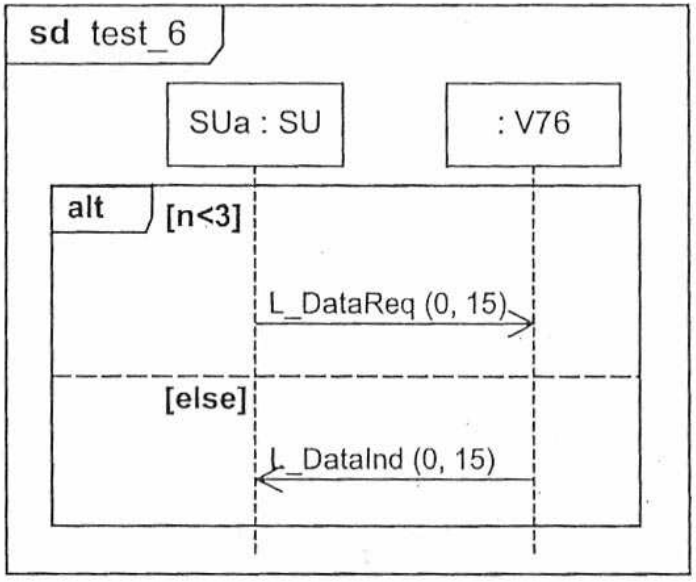
alt	alternatives; more than two alternatives are possible.
opt	option
loop	repetition
break	description of behavior expected after a break
par	parallel independent execution of several operands
ignore	to define messages to be ignored in the execution
consider	to define messages to be considered in the execution
seq	weak sequencing (default)
strict	strict sequencing
neg	to define forbidden behavior
critical	critical region, non-interruptible behavior
assert	assertion, to define a message sequence that must occur

Alt operator, non-deterministic



We will use non-determinism, although not allowed in standard.

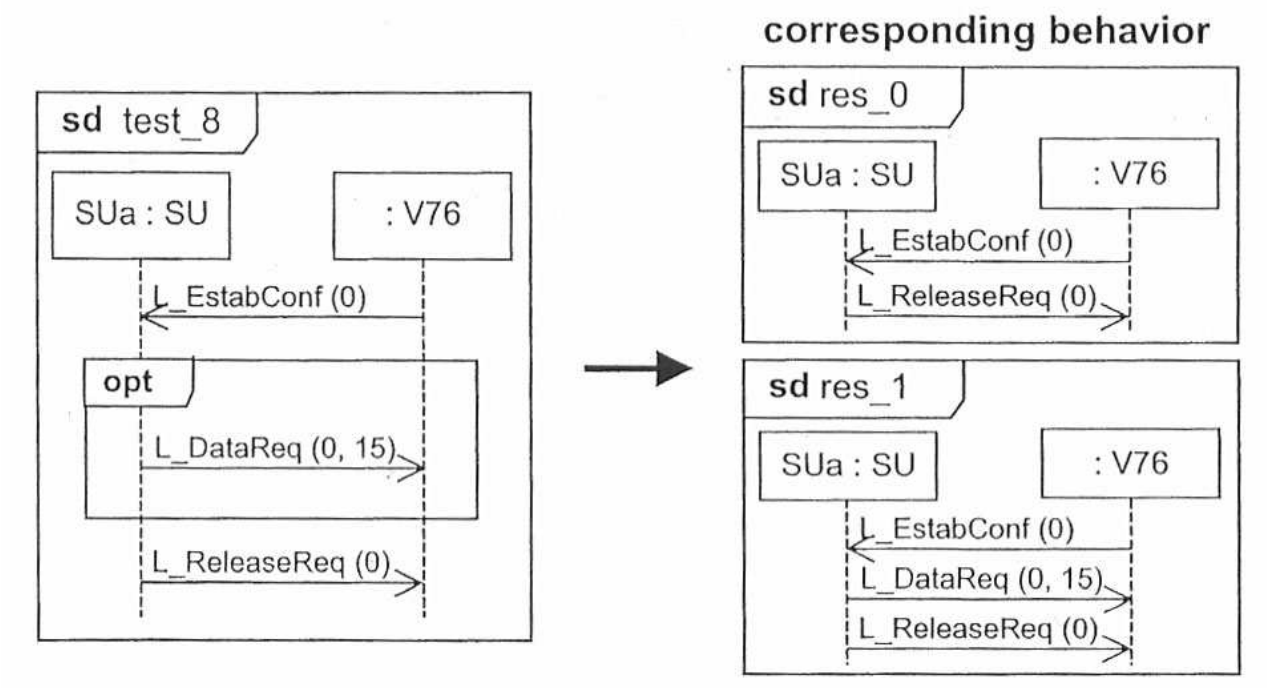
Alt operator with guards



- Guards specify what alternative will be taken.
- More than two possibilities may be specified (**case** construct).
- The guards must be exclusive, and their disjunction must be *true*.

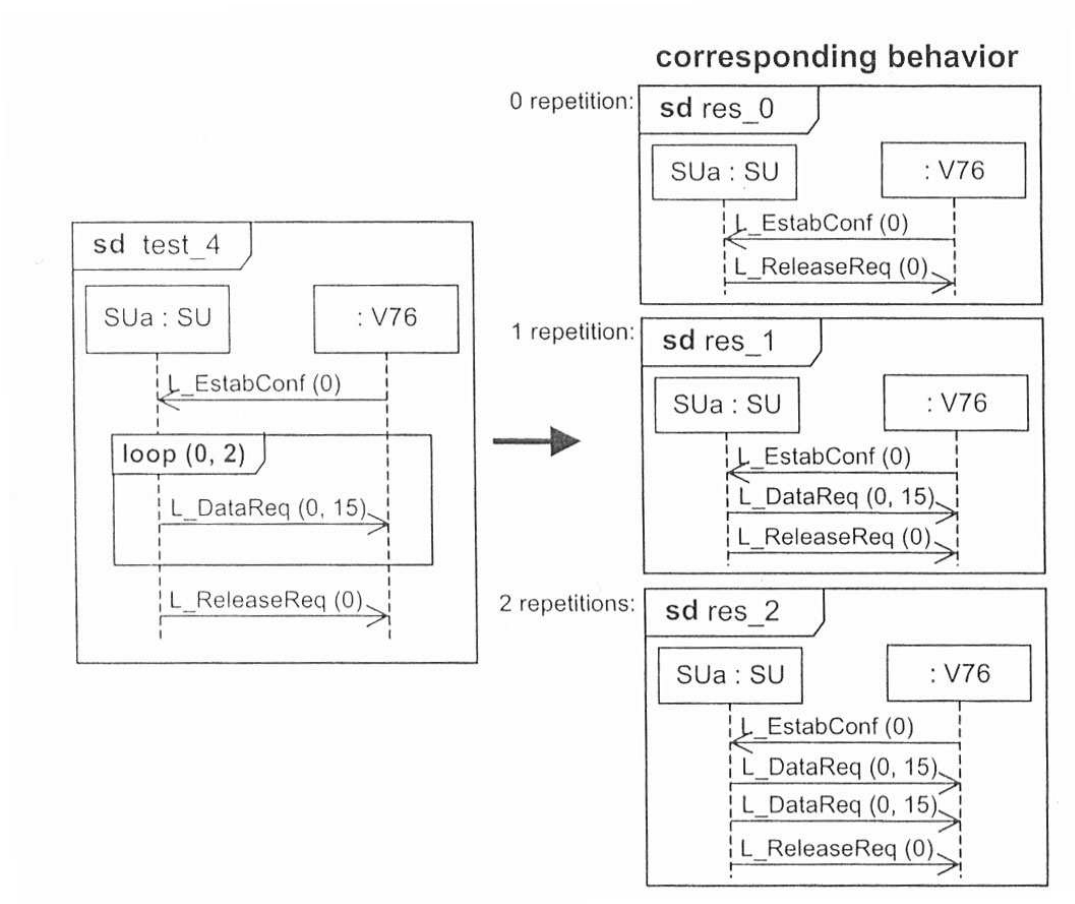
Opt operator

Corresponds to "Alt" with only one alternative. Standard requires interaction condition (conditions not given are considered to be *true*).



Loop operator

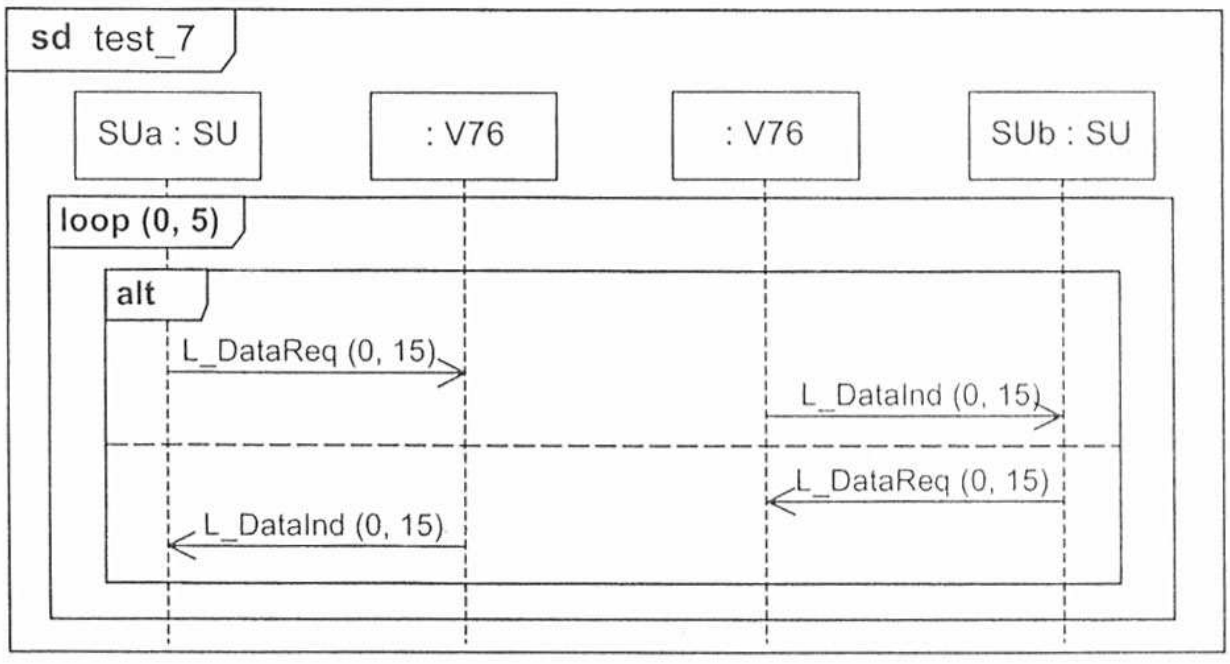
Fragment can be repeated a number of times.



Loop operator variants

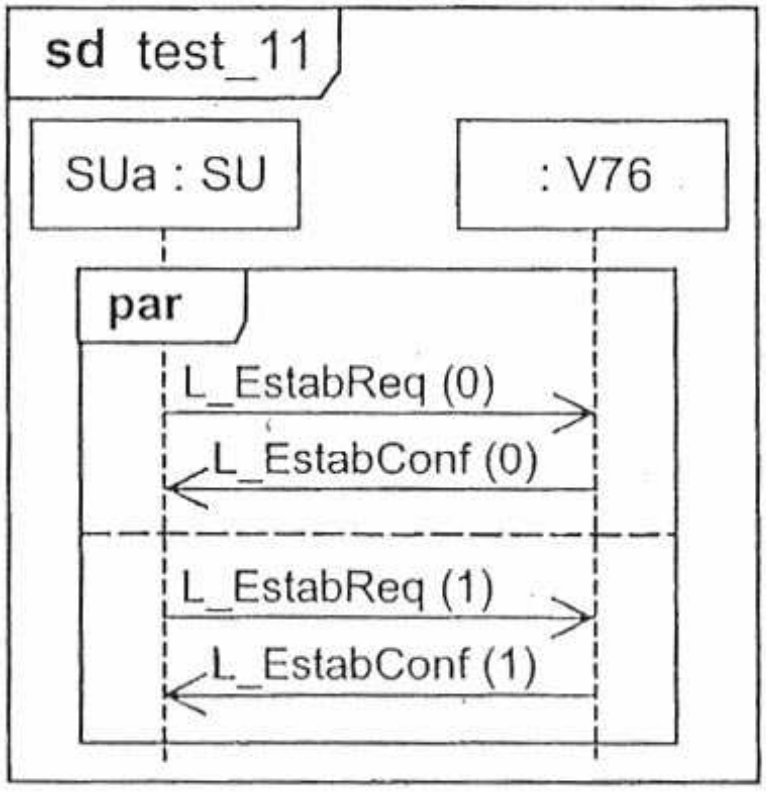
	repetitions	
	minimum	maximum
loop (0, 2)	0	2
loop (3)	3	3
loop (0, *)	0	not limited
loop	0	not limited

Nested operators



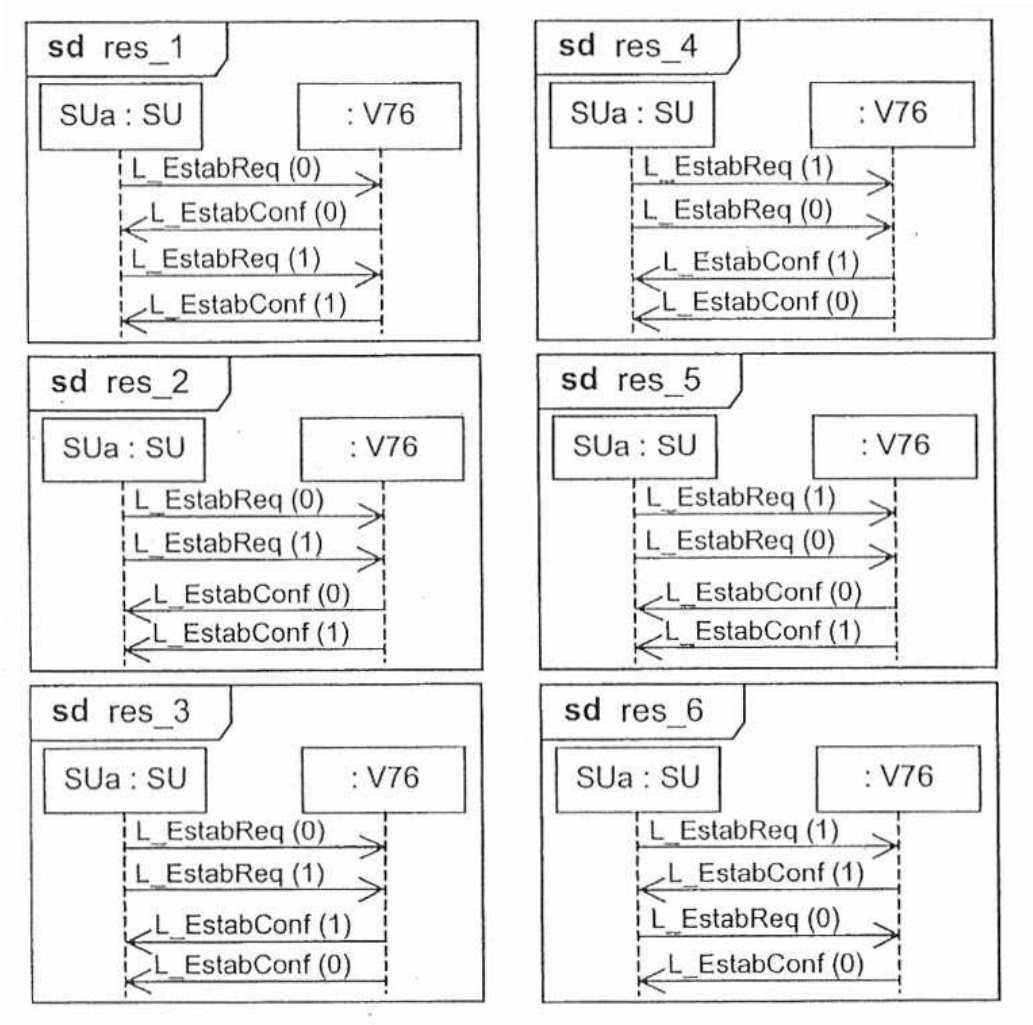
Combined fragments can be nested (e.g., *alt* fragment inside a *loop* fragment).

Par operator



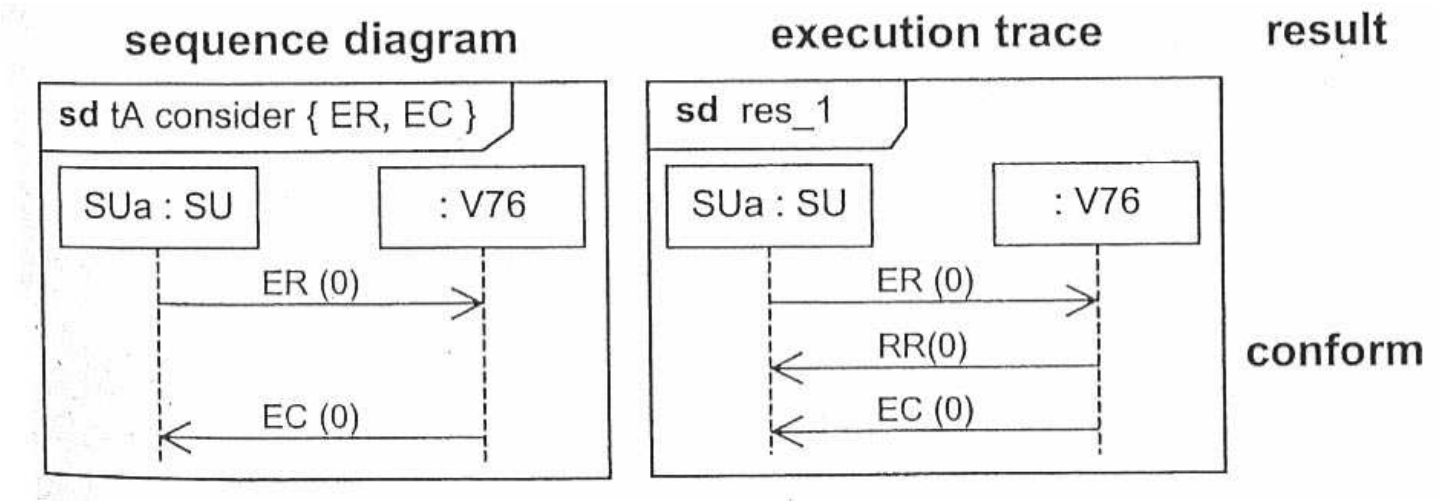
Describes the parallel merge between the behaviors of the operands (interleaving).

Expanded behavior of par operator



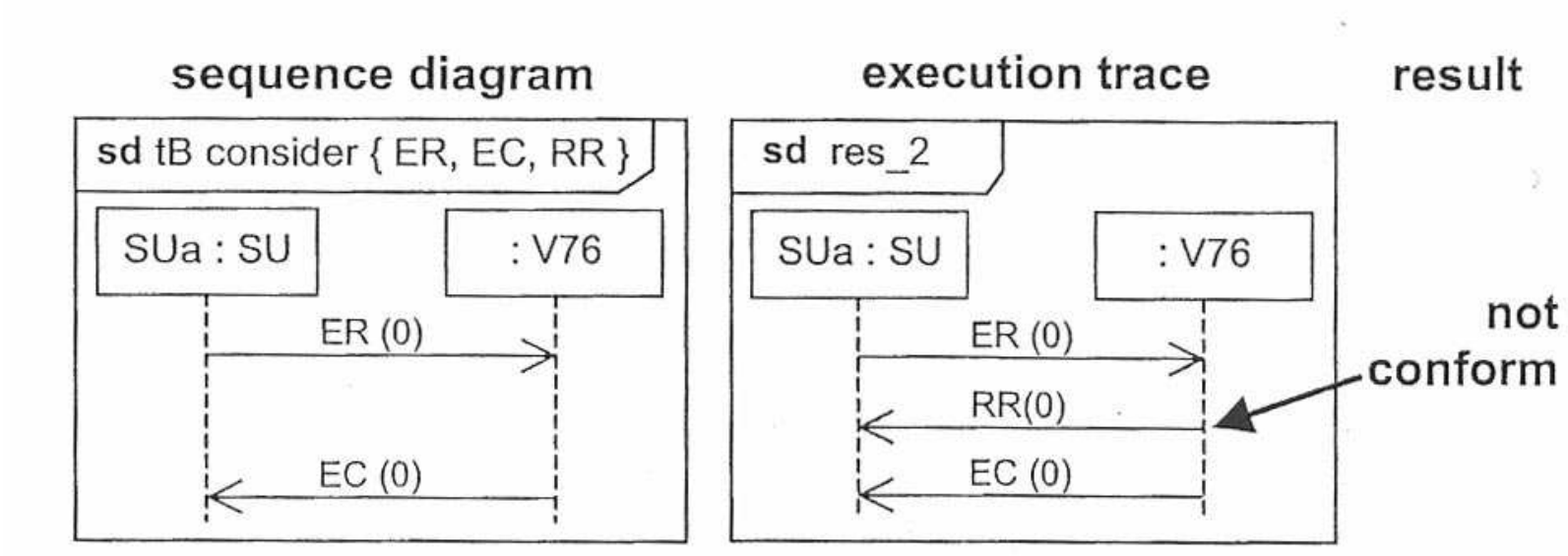
Consider operator (1)

Filter for relevant messages, noted in "headline" after the name of the interaction.



Further ("irrelevant") messages, here *RR*, may occur.

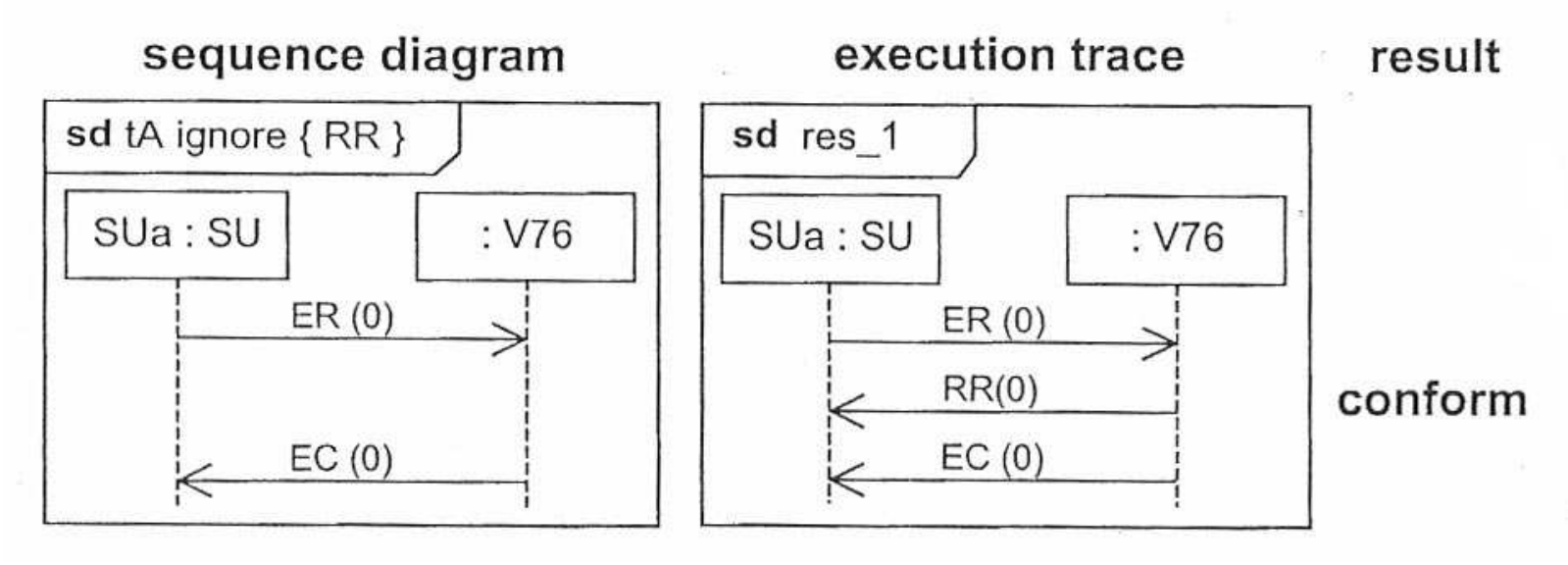
Consider operator (2)



If *RR* is considered, too, then the execution trace does not conform to the sequence diagram *tB*.

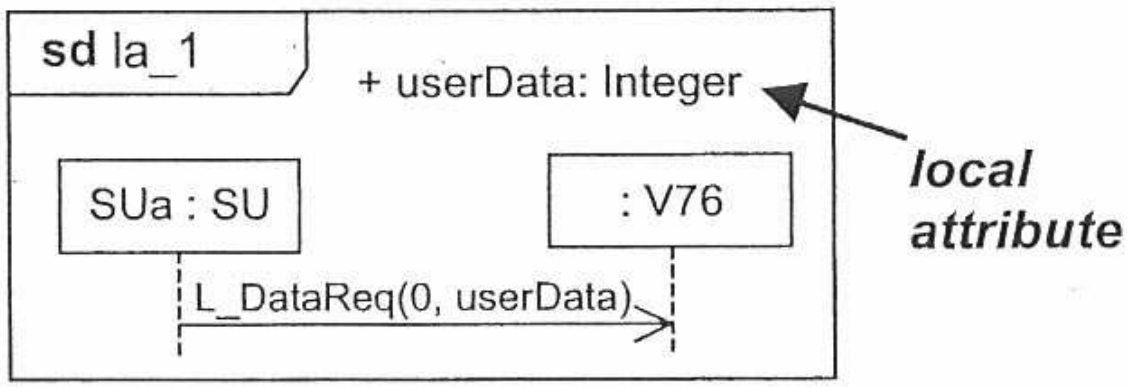
Ignore operator

Dual to consider operator. All messages that are not ignored are considered.



Local attributes

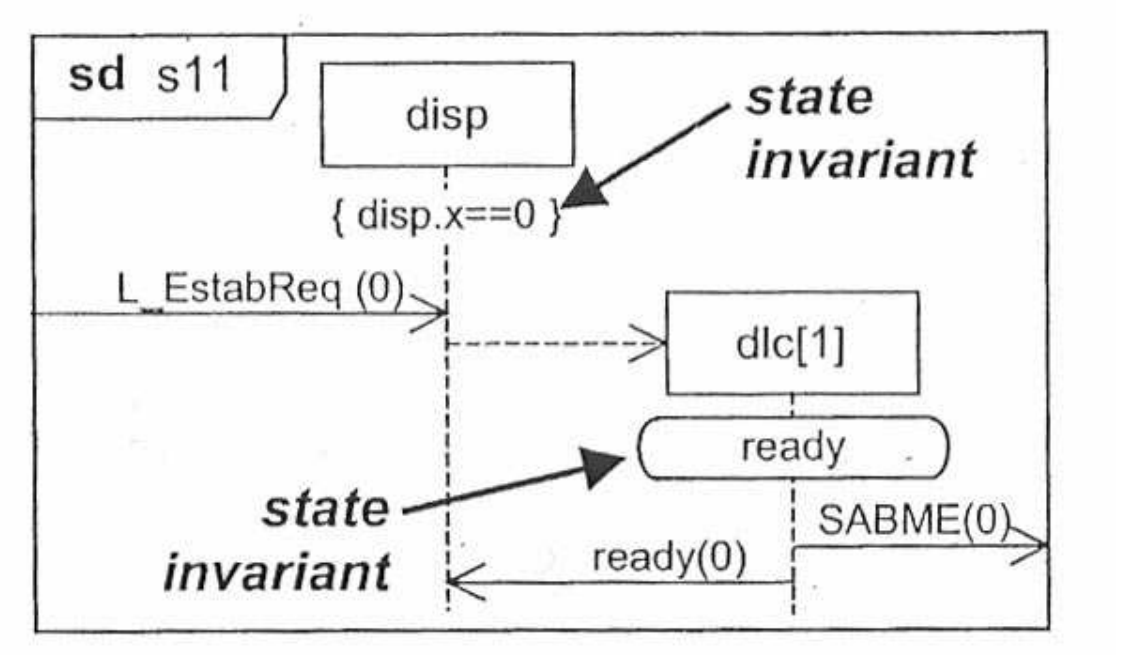
Like classes, sequence diagrams may have local attributes that may be public or private.



Local attributes can be used as parameters of messages.

State invariants

Express conditions for interactions. Interactions where the condition does not hold do not conform to the given diagram.

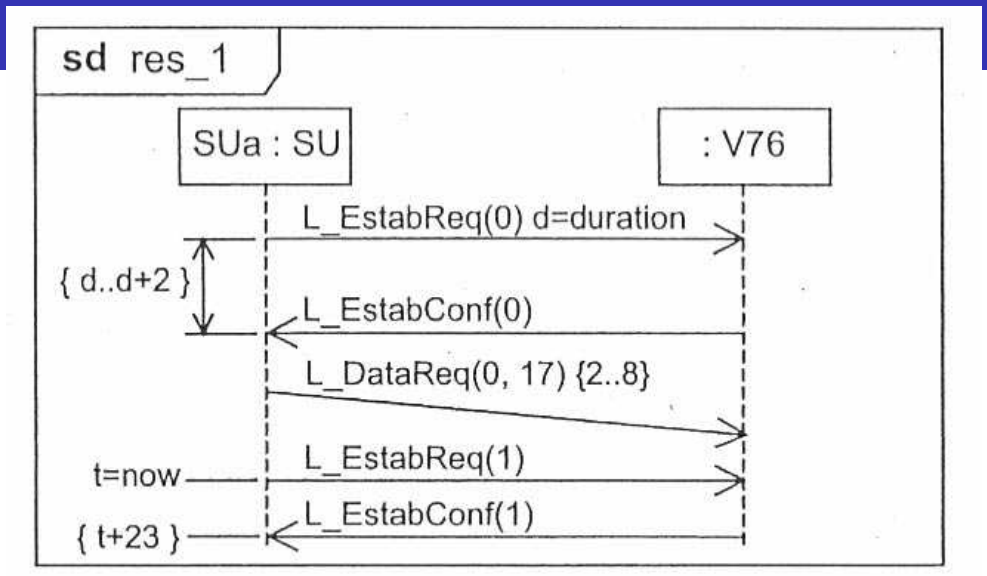


Invariants can be expressed as constraints, state symbols or notes.

Time constraints

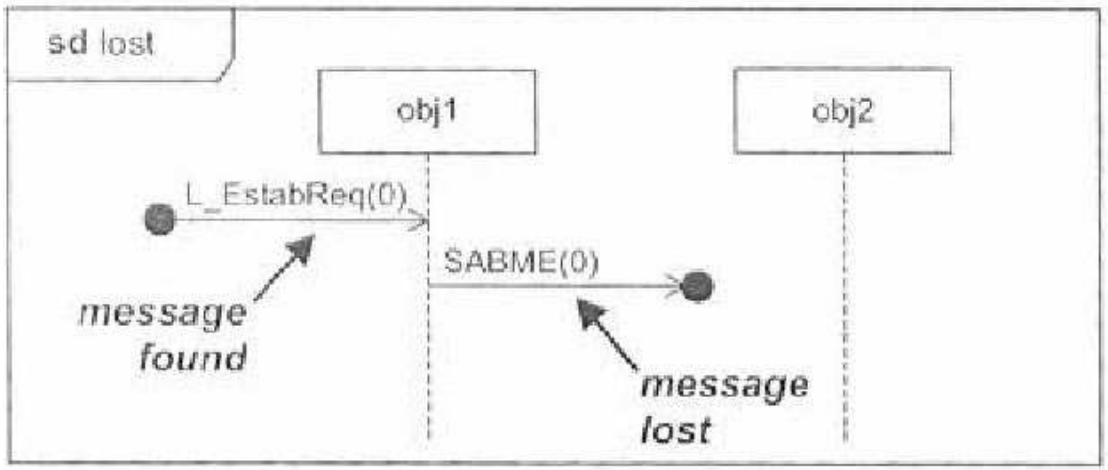
- The used time unit must be specified, e.g., using a note.
- Points in time and durations may be specified.
- Durations may be specified as time intervals, e.g., $\{t..t+3\}$
- Pre-defined: **now** for actual time, **duration** for duration between sending and reception of a message.

Example



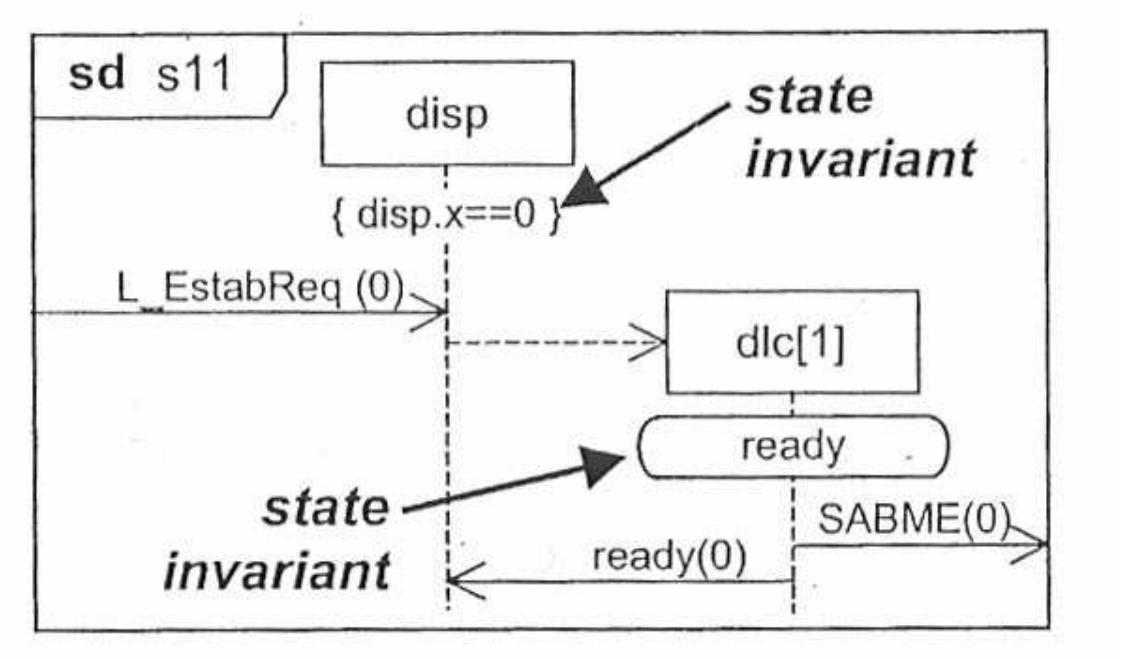
- The time between transmission and reception of signal *L_EstabReq*(0) is measured and stored in d.
- Signal *L_EstabConf*(0) must occur between d and $d+2$ time units after *L_EstabReq*(0).
- The duration of signal *L_DataReq* must be between 2 and 8 time units.
- The date of transmission of *L_EstabReq*(1) is stored into t, and the reception of *L_EstabConf*(1) must occur at $t+23$.

Lost and found messages



- Notation: large dot.
- Lost message: reception event not modeled.
- Found message: sender not known.
  Will be used for initialization of machine.

Messages from and to gates



Used for messages with unknown source or destination. *L_EstabReq(0)* is a message from a gate and *SABME(0)* is a message to a gate.

Phase 4: Derive machine behavior specification for each subproblem P_i

input:	requirements R from Phase 2	natural language
	domain knowledge D from Phase 2	natural language
	problem diagram for P_i from Phase 3	Jackson with dot-notation
output:	specification S_{P_i} of machine to construct	natural language
	sequences of interactions with annotated states for the domains in the environment, expressing R_{P_i} and D_{P_i}	sequence diagrams with annotated states
	sequences of interactions on initialization	sequence diagram with annotated states
validation:	$D \wedge S_{P_i}$ are non-contradictory	
	$D \wedge S_{P_i} \Longrightarrow R_{P_i}$	
	all requirements must be captured	
	in the sequence diagrams refined phenomena of the problem diagrams are used as signals	
	direction of signals must be consistent with control of shared phenomena	
	signals must connect domains as connected in problem diagram	
	the relationships of Phase 3 must be consistent with the states	

Executing Phase 4

1. Derive a specification of the machine.
2. Express requirements and domain knowledge as sequence diagrams.
3. Check the correctness of the developed specification.

Executing Phase 4 – Sequence diagrams I

To create the sequence diagrams the following steps have to be performed:

- For each domain which is directly connected to the machine in a problem diagram, a lifeline is drawn.
- Domains can be merged in the sequence diagram to simplify the description.
- The machine to be built (together with all domains that belong to the machine) can be represented by one lifeline in the sequence diagrams.
- The phenomena are represented by asynchronous signals between lifelines.
- It should be assumed that an asynchronous signal occurs when the state in the environment changes.

Executing Phase 4 – Sequence diagrams II

- To express the coherence between the sequences *states* for the domains in the environment should be included.
- Appropriate case distinctions according to these states should be introduced.
- For the case distinctions new diagrams should be created instead of using the *alt* operator.
- The sequence diagrams can be split at appropriate states, if necessary.
- Specify the initialization of the machine. A *found signal* or a signal from a gate can be used to specify a *power on* signal.
- Refine events by adding parameters to phenomena or define rules for the refinement.
- Add timing constraints.

Remarks I

- The relationships of Phase 3 should be consistent with the states, e.g., if for two sequential sequence diagrams the last state of the first diagram is the same as the first state of the second diagram.
- Sequence diagrams can be used to discuss important aspects with the customers, and they are the outline for the test in Phase 12.
- Each diagram represents one concrete interaction sequence. Do not try to make your diagrams too general. It is better to draw further diagrams. The sequence diagrams should express typical cases with example values. Loops, states, references, and co-regions do not cause any problems, while e.g., parallelism and considered signals should be used with care.

Remarks II

- To reuse these diagrams later, the requirements and the domain knowledge should be expressed as separate sequence diagrams instead of expressing the specification directly.
- For all connection domains (being not part of the machine) the domain knowledge should be described by separate sequence diagrams.
- To express the requirements, the separately described domains should be merged with the machine.

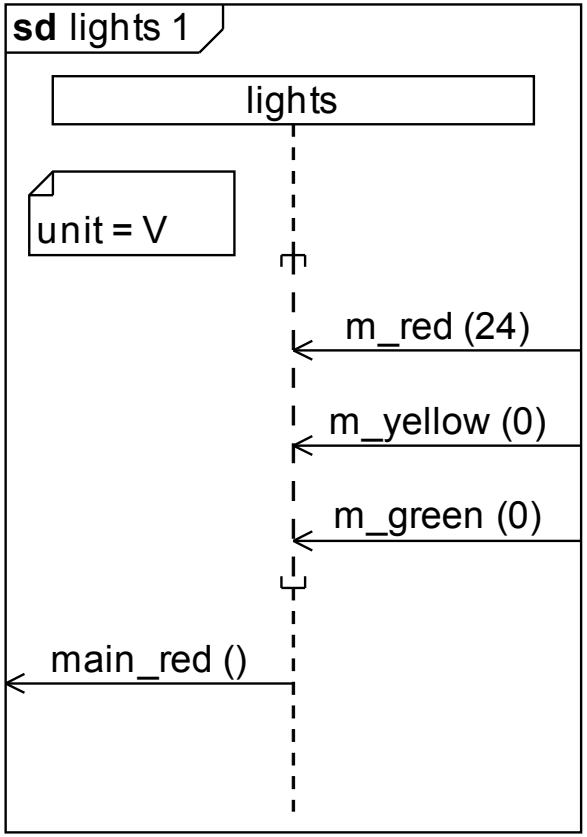
Remarks III

- The relationships between subproblems must be considered for validation:
  - If two subproblems are sequential the sequence diagrams of the first subproblem end with the same states as the second subproblem sequence diagrams start.
  - All sequence diagrams of one subproblem end with the initial states of succeeding diagrams (all alternatives can be considered).
  - Sequences for parallel subproblems must start with states that can be reached in some parallel subproblem.

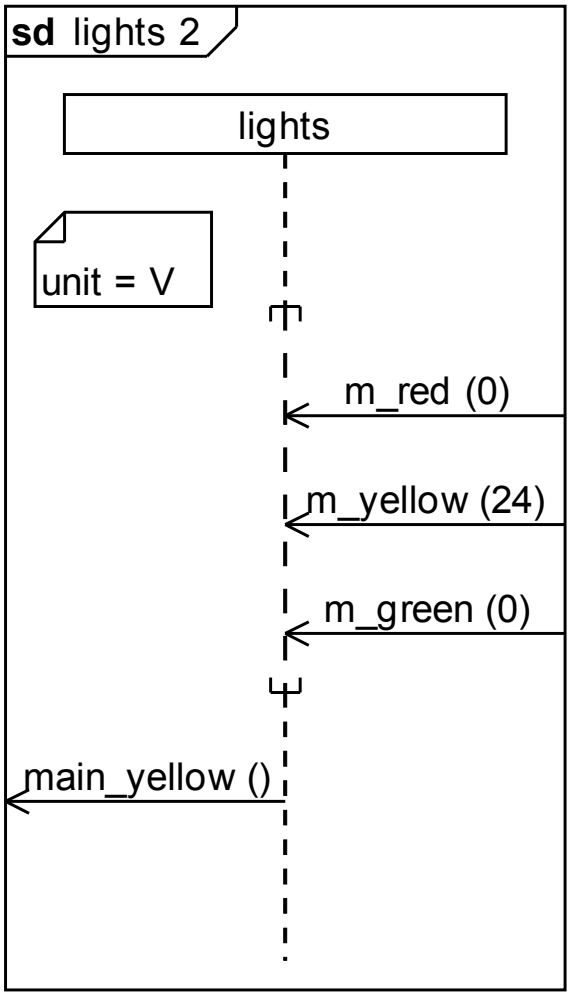
Example 1: traffic light control

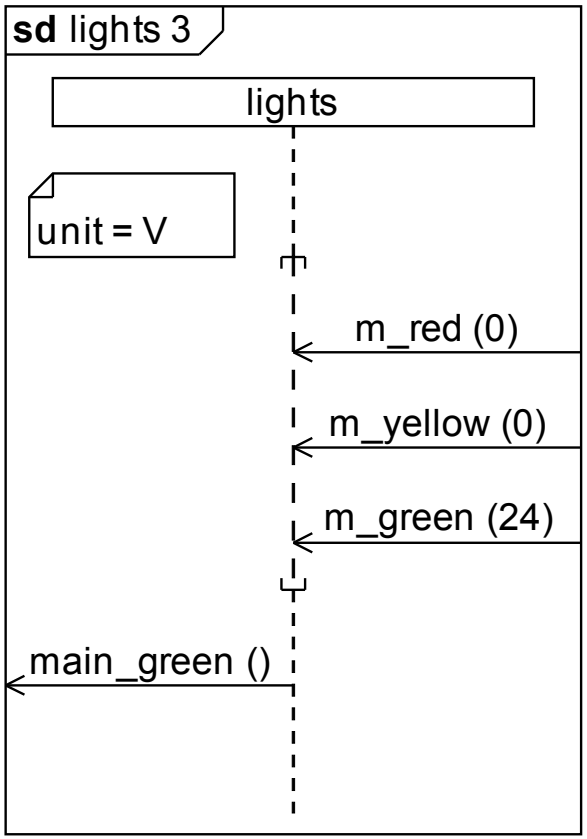
Domain Knowledge of lights domain (Sequence diagram) I

It is difficult (many signals) to express the specification directly, therefore D and R are expressed as separate sequence diagrams.

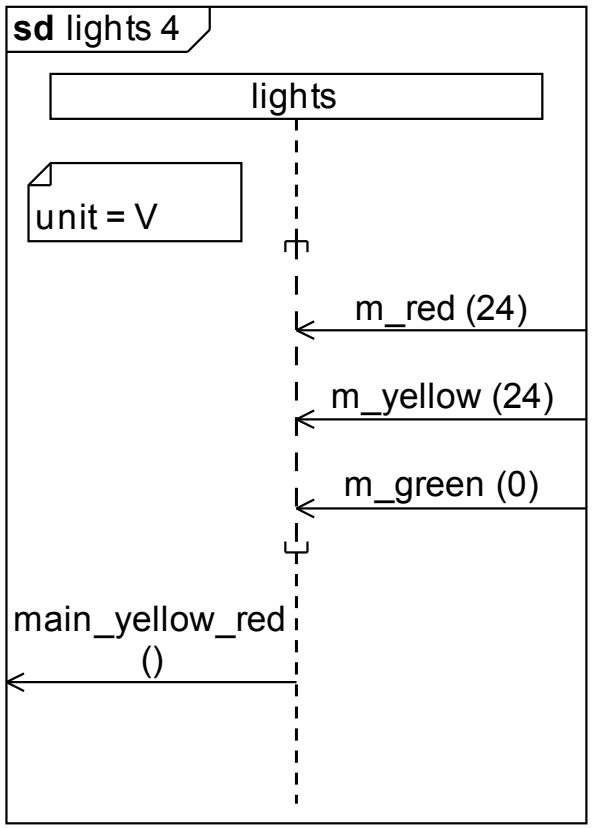


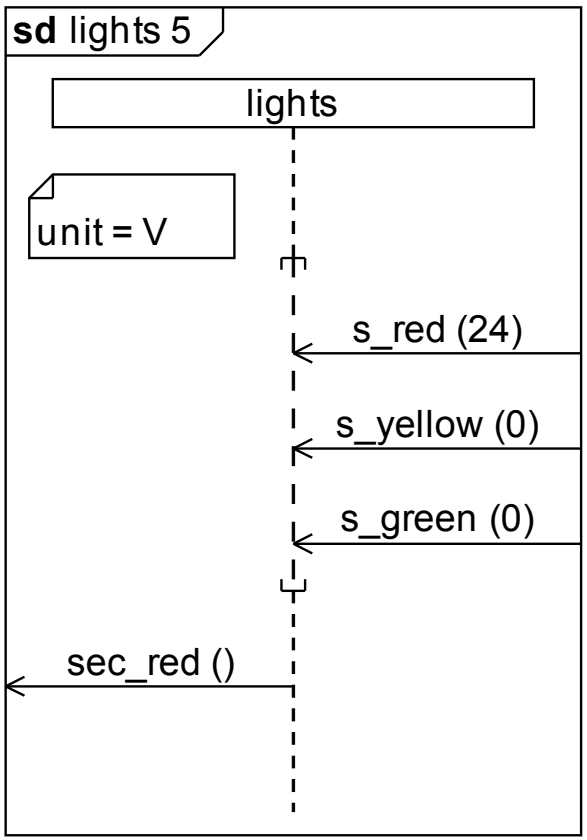
Domain Knowledge of lights domain (Sequence diagram) II



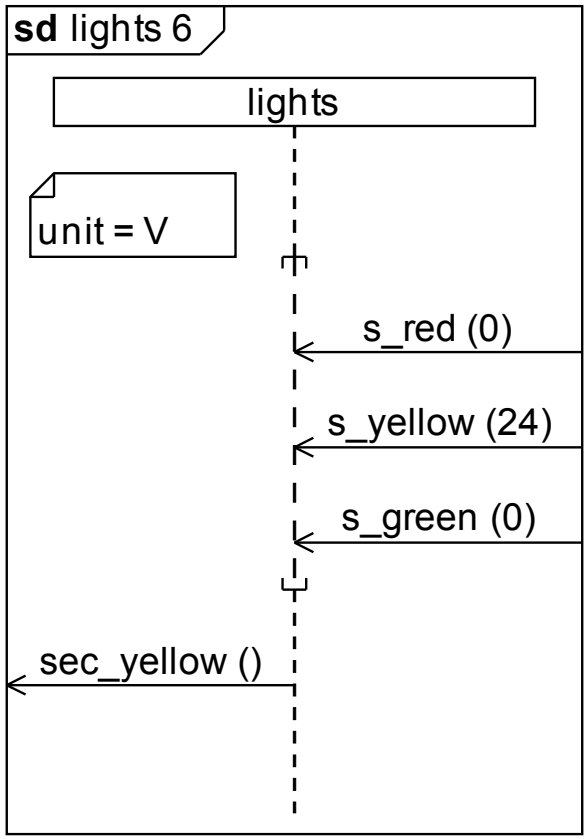


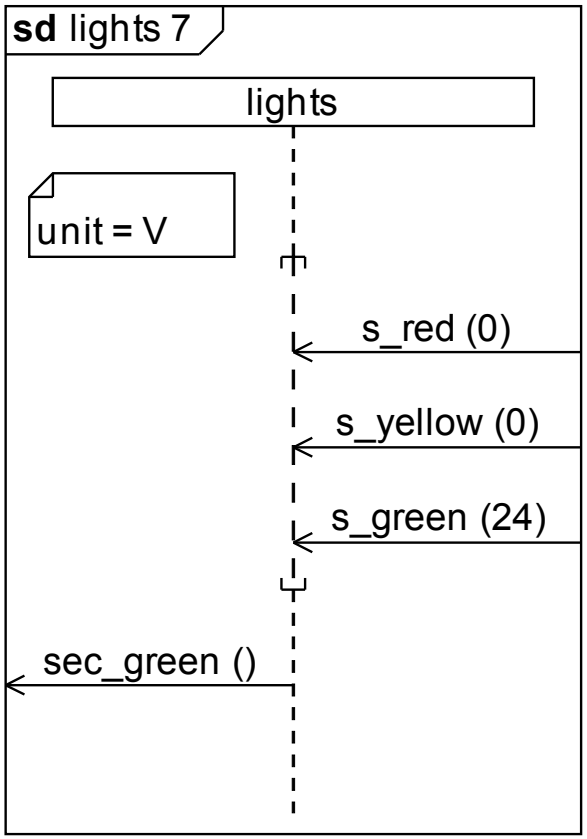
Domain Knowledge of lights domain (Sequence diagram) III



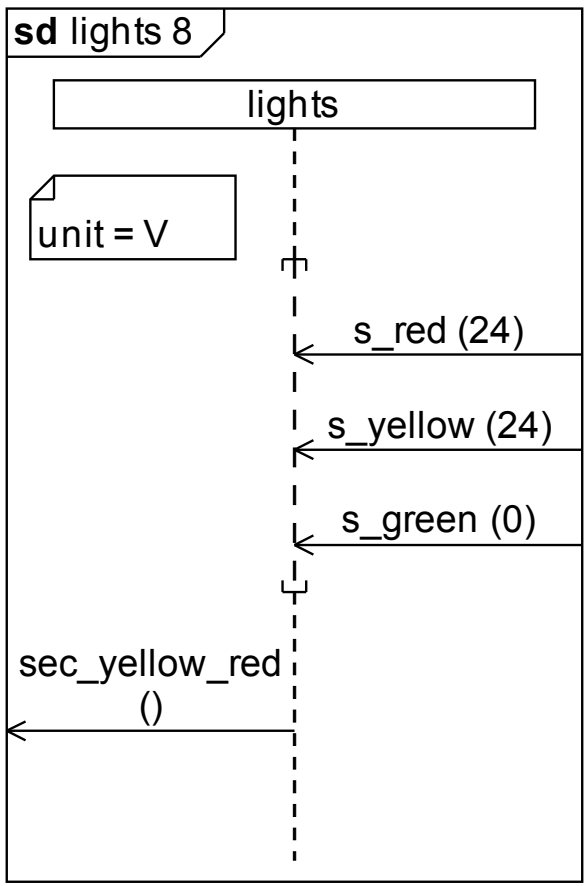


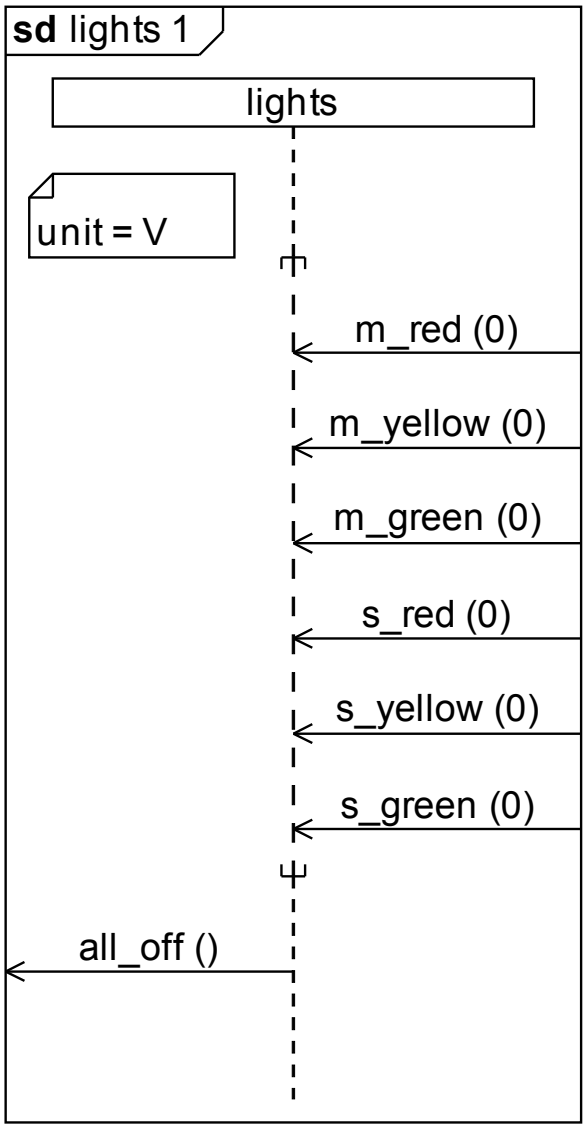
Domain Knowledge of lights domain (Sequence diagram) IV



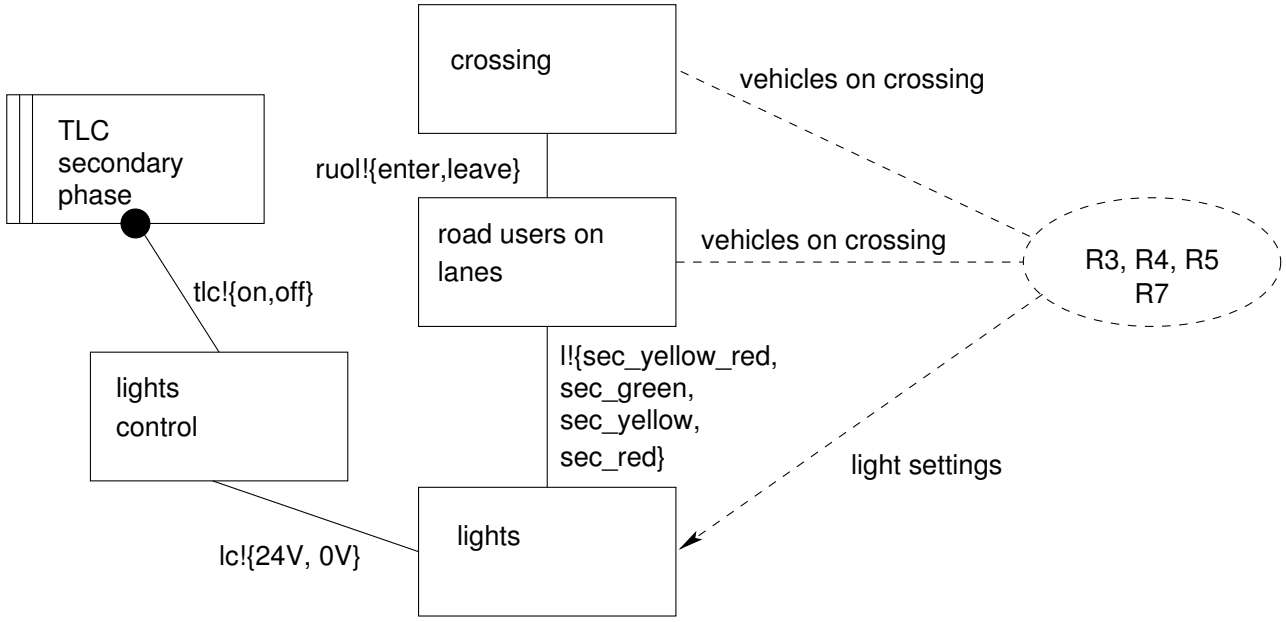


Domain Knowledge of lights domain (Sequence diagram) V





SecondaryRoadPassing Problem Diagram I



R3 : Vehicles on the main road should be allowed to pass the crossing for a longer period of time than from the secondary road (if not emergency-case).

R4 : While vehicles on one road are allowed to pass, the others should be stopped.

SecondaryRoadPassing Problem Diagram II

R5 : The lights should switch in the following order: red - red+yellow - green - yellow - red. Other combinations (except "all off", yellow blinking, and green - yellow - green in emergency case[7]) are not allowed.

R7 : After switching to red, the traffic flow of both roads should be stopped for a period of time

[7] Added later to eliminate contradictions

SecondaryRoadPassing – Derive Specification I

allowed to pass the crossing is no interface phenomenon of the machine (controlled by the environment, not observable by the machine). $\Rightarrow$ Transformed into *s_green*(24) or *m_green*(24) followed by *s_green*(0) or *m_green*(0) using $F2$ and $A1$.
should be stopped is no interface phenomenon of the machine (controlled by the environment, not observable by the machine). $\Rightarrow$ Transformed into *s_red*(24) or *m_red*(24) followed by *s_red*(0) or *m_red*(0) using $F1$ and $A1$.
longer period of time should be refined into concrete values using $F4$ and $F12$ (definition of fairness, maximum time for waiting).

S3 The traffic light should switch on the green light bulb for the main road (m_green(24), m_green(0)) for at least 20 s and for the secondary road (*s_green*(24), *s_green*(0)) 10 s (if not emergency-case).

S4a While green is shown for the main road (*m_green*(24)), the secondary road lights should show red (*s_red*(24)).

S4b While green is shown for the secondary road (*s_green*(24)), the main road lights should show red (*m_red*(24)).

SecondaryRoadPassing – Derive Specification II

S5a The lights should be switched in the following order:
red ($m\_red(24), m\_yellow(0), m\_green(0)$) –
red/yellow ($m\_red(24), m\_yellow(24), m\_green(0)$) –
green ($m\_red(0), m\_yellow(0), m\_green(24)$) –
yellow ($m\_red(0), m\_yellow(24), m\_green(0)$) –
red ($m\_red(24), m\_yellow(0), m\_green(0)$).
Other combinations (except "all off"
($m\_red(0), m\_yellow(0), m\_green(0)$) and yellow blinking
($m\_red(0), m\_yellow(24/0), m\_green(0)$) are not allowed.

S5b The lights should be switched in the following order:
red ($s\_red(24), s\_yellow(0), s\_green(0)$) –
red/yellow ($s\_red(24), s\_yellow(24), s\_green(0)$) –
green ($s\_red(0), s\_yellow(0), s\_green(24)$) –
yellow ($s\_red(0), s\_yellow(24), s\_green(0)$) –
red ($s\_red(24), s\_yellow(0), s\_green(0)$) .
Other combinations (except "all off"
($s\_red(0), s\_yellow(0), s\_green(0)$) and yellow blinking
($s\_red(0), s\_yellow(24/0), s\_green(0)$) are not allowed.

SecondaryRoadPassing – Derive Specification III

S7 After switching to red ($s\_red(24)$ or $m\_red(24)$) the lights traffic show red for both roads for 3 s ($s\_red(24)$ and $m\_red(24)$) before ($s\_red(0)$ or $m\_red(0)$).

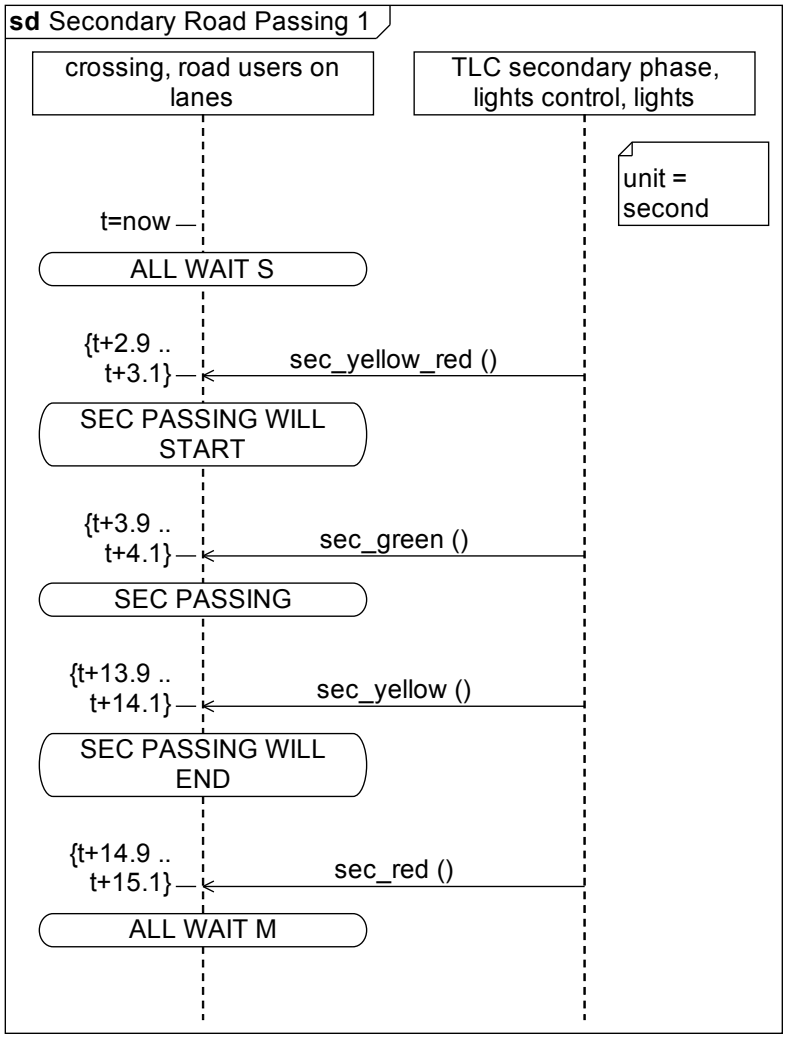
Sequence diagram for SecondaryRoadPassing I

For the TLC instead of the specification, the requirements are expressed as sequence diagrams:

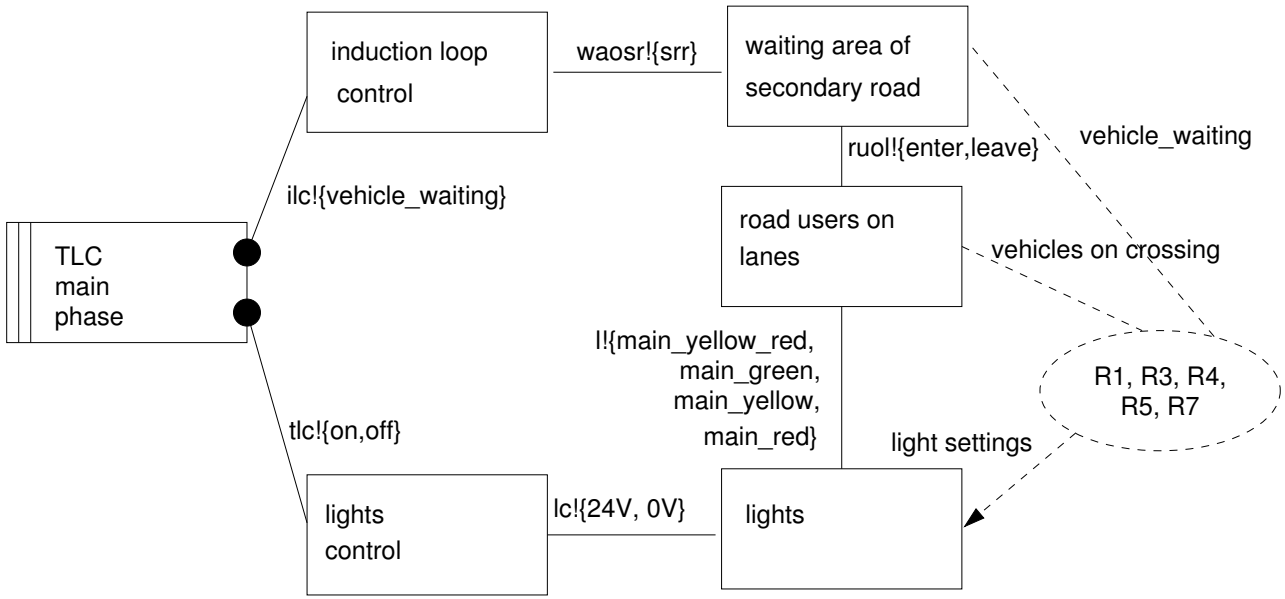
- The domains *crossing* and *road users on lanes* are merged.
- The domains *TLC secondary phase*, *lights control*, and *lights* are also merged to express requirements.

In this step the requirement is refined by adding timing constraints, e.g., the state *SECONDARY PASSING* should take 10 seconds (see $F4$ and $F12$).

Sequence diagram for SecondaryRoadPassing II



MainRoadPassing Problem Diagram I



R1 : When there is a car waiting on the secondary road, the traffic lights should stop the flow of traffic on the main road for a period of time and allow the traffic flow on the secondary road.

R3 : Vehicles on the main road should be allowed to pass the crossing for a longer period of time than from the secondary road (if not emergency-case).

MainRoadPassing Problem Diagram II

R4 : While vehicles on one road are allowed to pass, the others should be stopped.

R5 : The lights should switch in the following order: red - red+yellow - green - yellow - red. Other combinations (except "all off", yellow blinking, and green - yellow - green in emergency case[8]) are not allowed.

R7 : After switching to red, the traffic flow of both roads should be stopped for a period of time

[8] Added later to eliminate contradictions

MainRoadPassing – Derive Specification I

allowed to pass the crossing is no interface phenomenon of the machine (controlled by the environment, not observable by the machine). $\Rightarrow$ Transformed into *s_green*(24) or *m_green*(24) followed by *s_green*(0) or *m_green*(0) using $F2$ and $A1$.
should be stopped and *allow the traffic flow* are no interface phenomena of the machine (controlled by the environment, not observable by the machine). $\Rightarrow$ Transformed into *s_red*(24) or *m_red*(24) followed by *s_red*(0) or *m_red*(0) using $F1$ and $A1$.
car waiting on the secondary road is no interface phenomenon of the machine (controlled by the environment, not observable by the machine). $\Rightarrow$ Transformed into (*srr*) using $F9$ and $A4$.

S1 When a secondary road request occurs (*srr*), the traffic lights should should switch on the red light for the main road for a period of time (*m_red*(24) followed by *m_red*(0)) and switch on the green light for the secondary road for a period of time (*s_green*(24) followed by *s_green*(0)).

S3 see SecondaryRoadPassing.

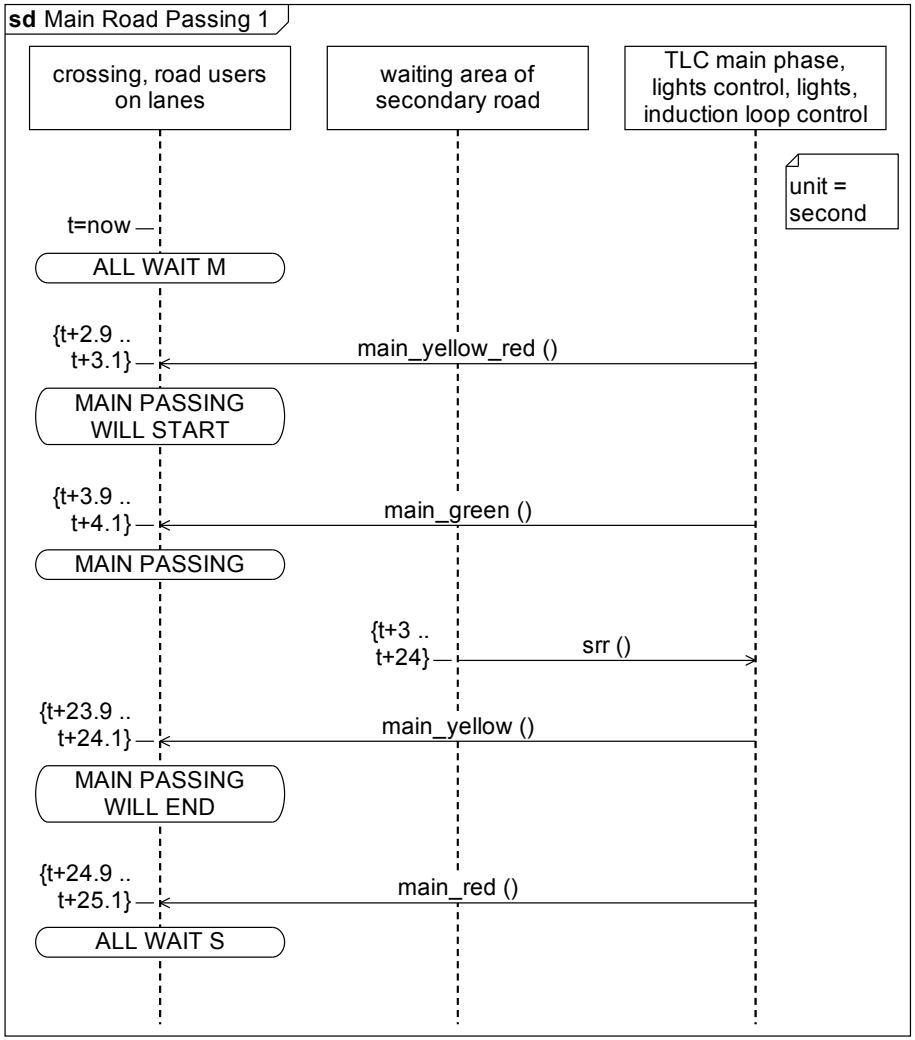
MainRoadPassing – Derive Specification II

S4a/b see SecondaryRoadPassing.

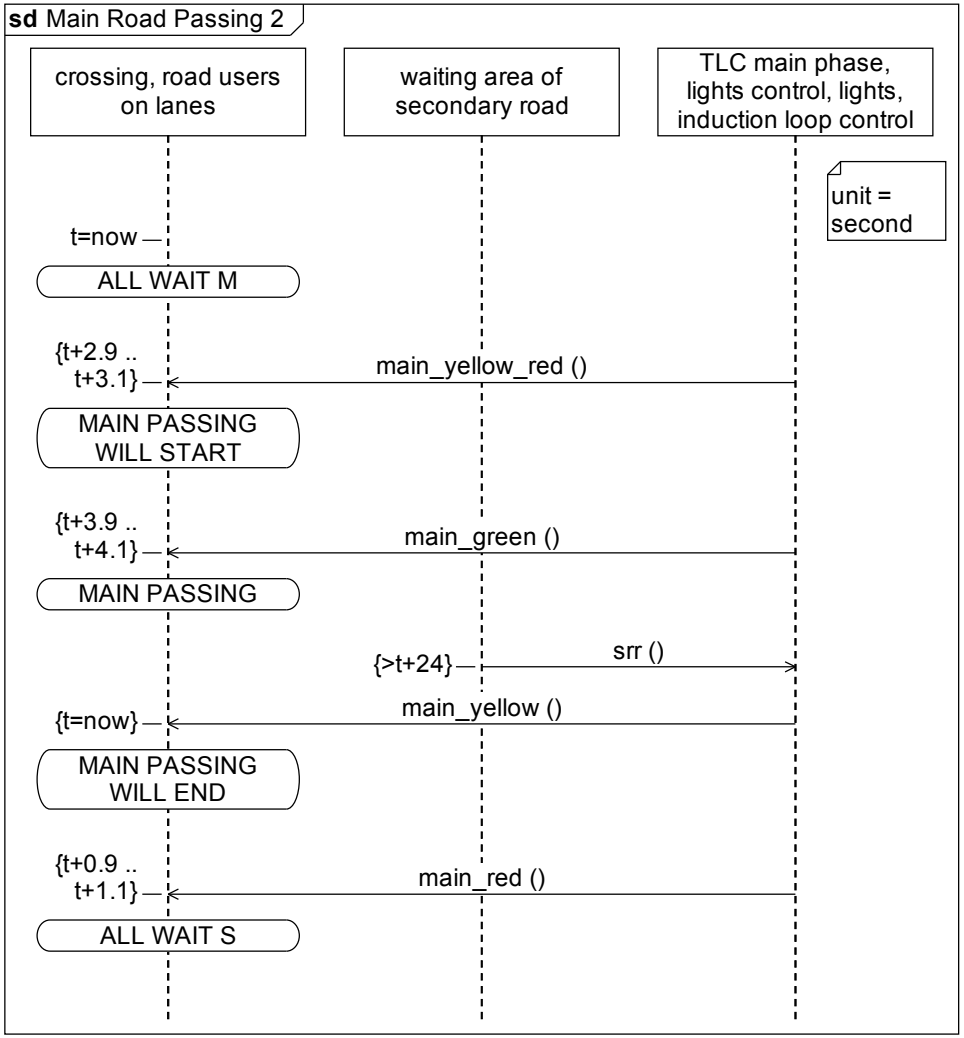
S5a/b see SecondaryRoadPassing.

S7 see SecondaryRoadPassing.

Sequence diagrams for MainRoadPassing I



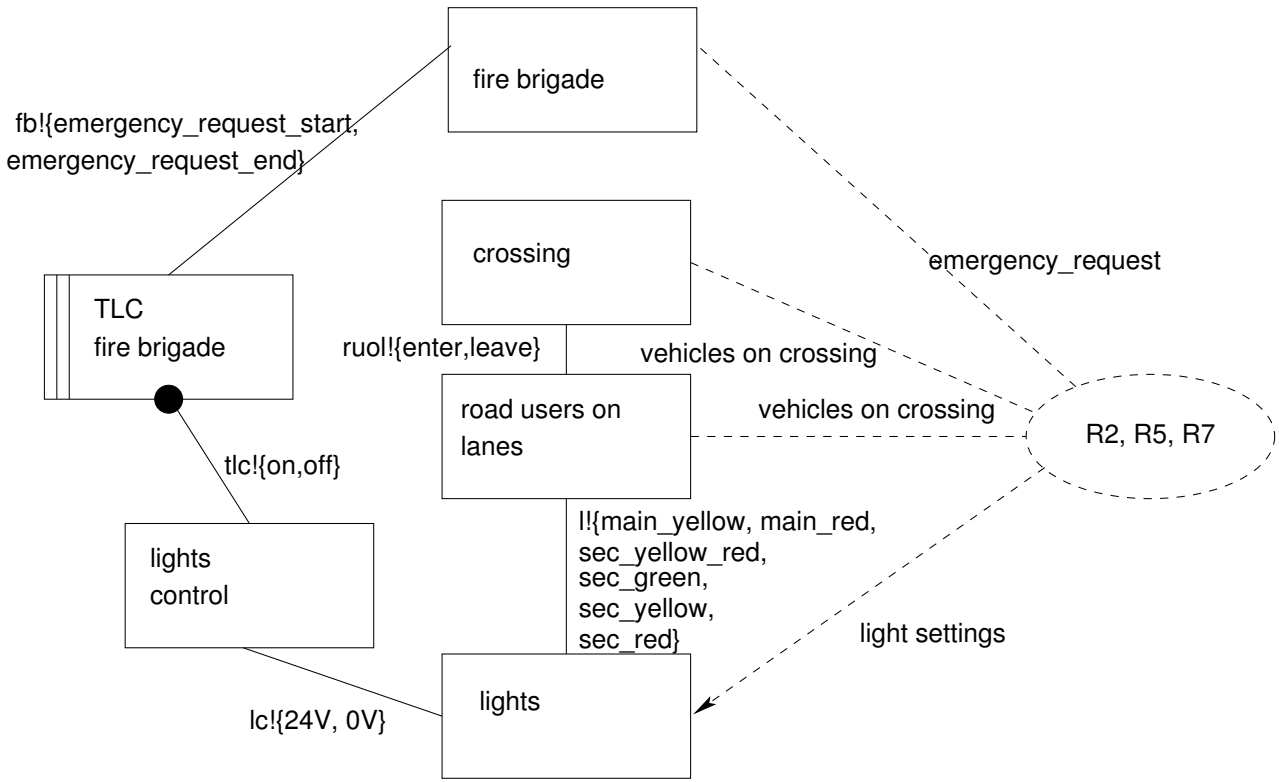
Sequence diagrams for MainRoadPassing II



Sequence diagrams for MainRoadPassing III

- The first sequence diagram expresses that the state *MAIN PASSING* takes at least 20 seconds, and therefore the requirement $R3$ is considered.
- The domains *crossing* and *road users on lanes* are merged.
- The domains *induction loop control*, *TLC main phase*, *lights control*, and *lights* are merged to express requirements.

EmergencyRequestSecondaryRoadPassing Problem Diagram I



R2 : As long as the emergency button is activated, the flow of traffic on the main road should be stopped and the flow of traffic on the secondary road should be allowed.

EmergencyRequestSecondaryRoadPassing Problem Diagram II

R5 : The lights should switch in the following order: red - red+yellow - green - yellow - red. Other combinations (except "all off", yellow blinking, and green - yellow - green in emergency case[9]) are not allowed.

R7 : After switching to red, the traffic flow of both roads should be stopped for a period of time

[9] Added later to eliminate contradictions

EmergencyRequestSecondaryRoadPassing – Derive Specification I

should be allowed is no interface phenomenon of the machine (controlled by the environment, not observable by the machine). $\Rightarrow$ Transformed into *s_green*(24) or *m_green*(24) followed by *s_green*(0) or *m_green*(0) using $F2$ and $A1$.
should be stopped is no interface phenomenon of the machine (controlled by the environment, not observable by the machine). $\Rightarrow$ Transformed into *s_red*(24) or *m_red*(24) followed by *s_red*(0) or *m_red*(0) using $F1$ and $A1$.

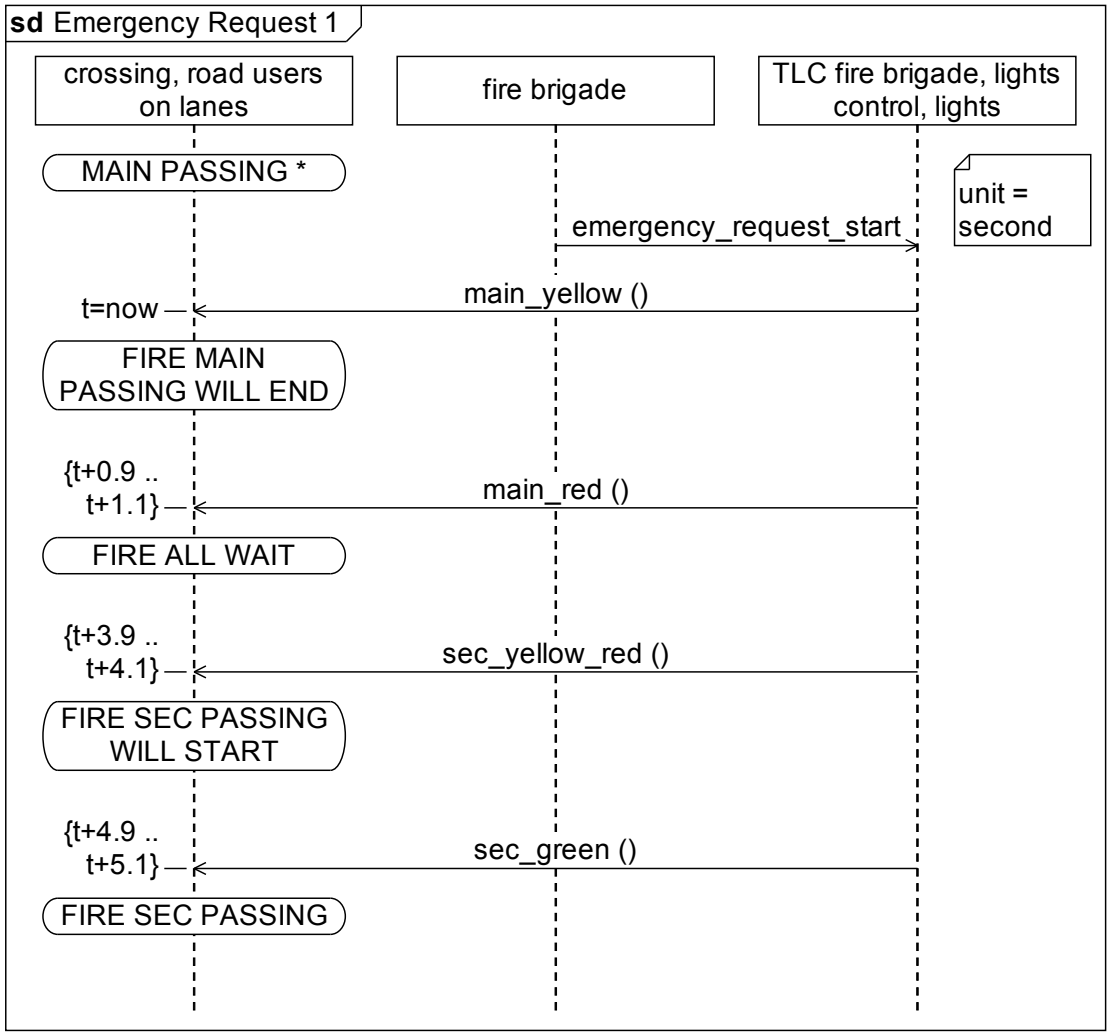
S2 As long as the emergency button is activated (*emergency_request_start*, *emergency_request_end*), the lights for the main road should be red (*m_red*(24)) and the lights for the secondary road should be green (*s_green*(24)).

S5a see SecondaryRoadPassing.

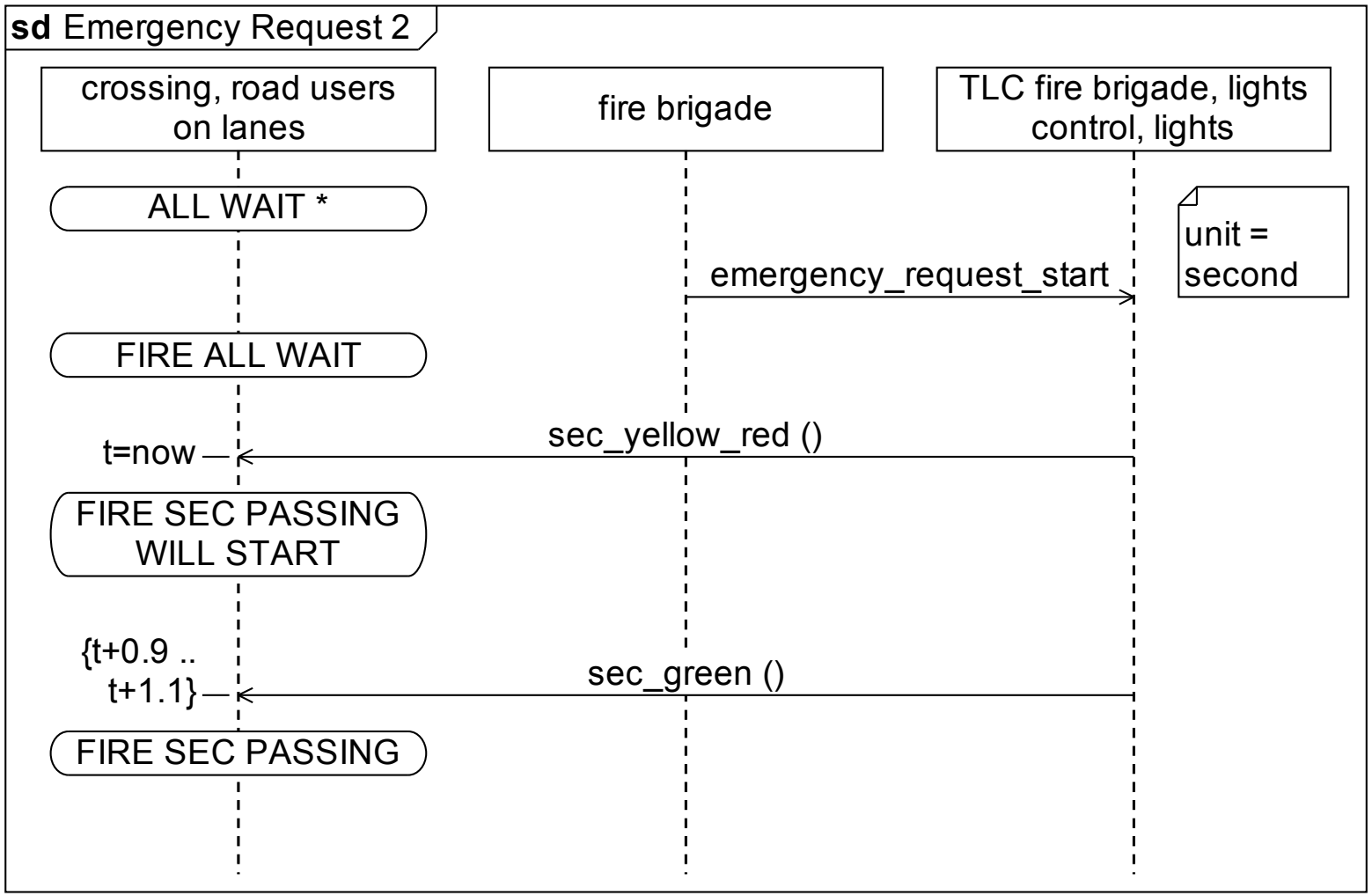
S5b see SecondaryRoadPassing.

S7 see SecondaryRoadPassing.

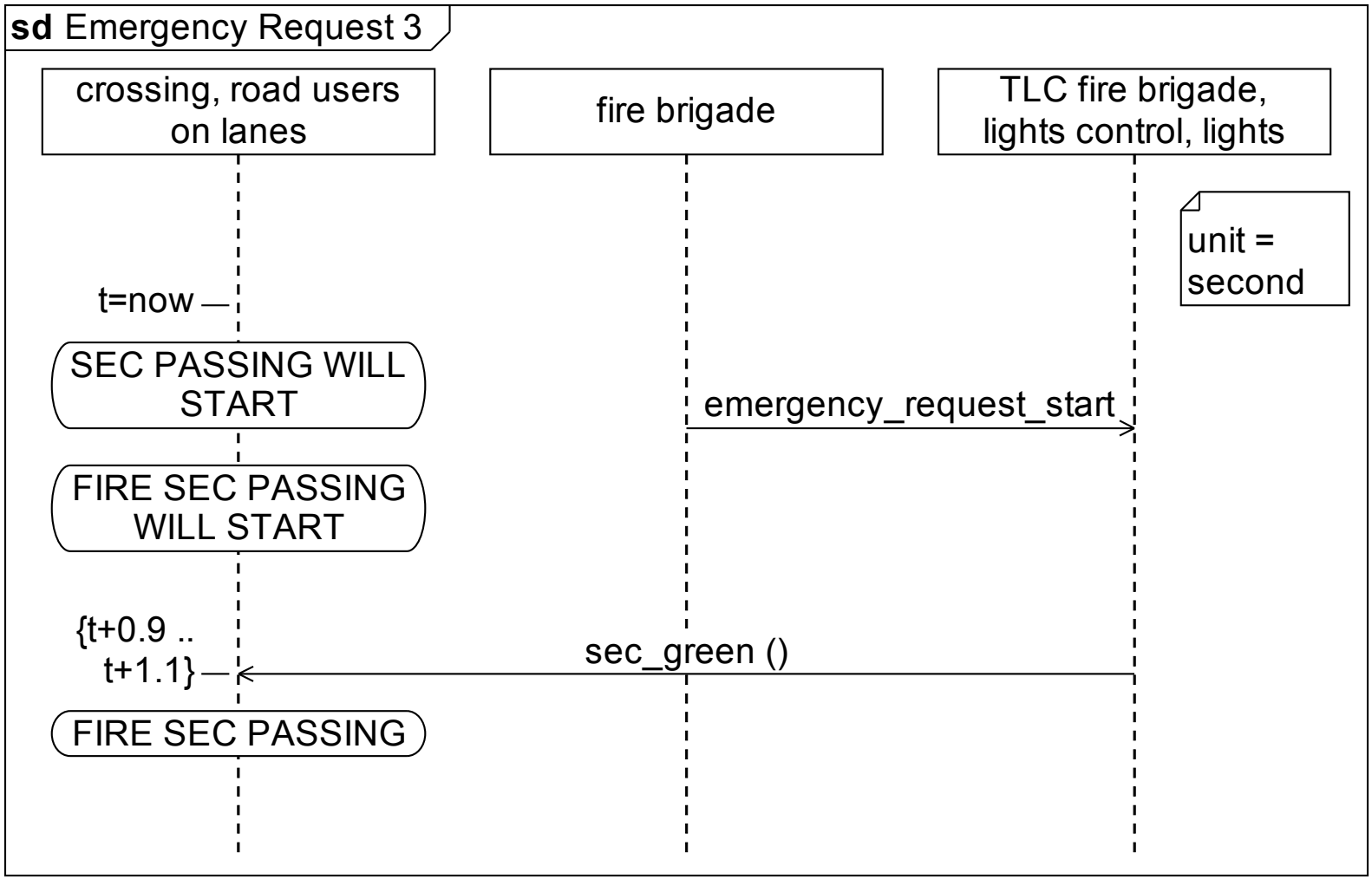
Sequence diagrams for EmergencyRequestSecondaryRoadPassing I



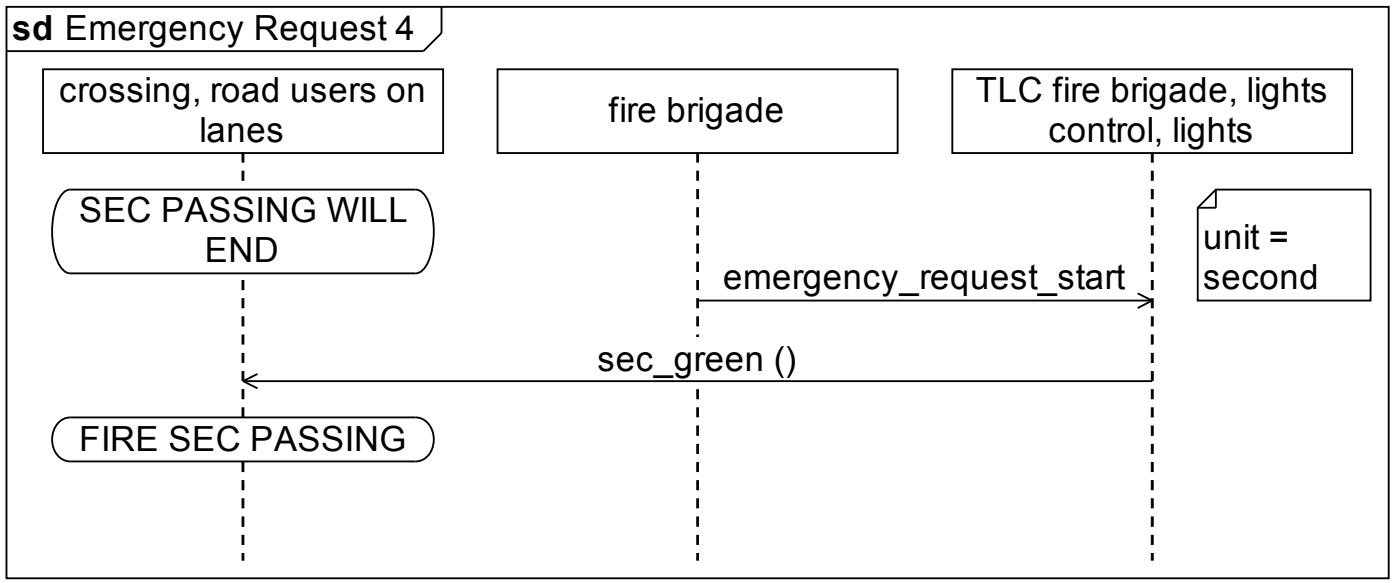
Sequence diagrams for EmergencyRequestSecondaryRoadPassing II



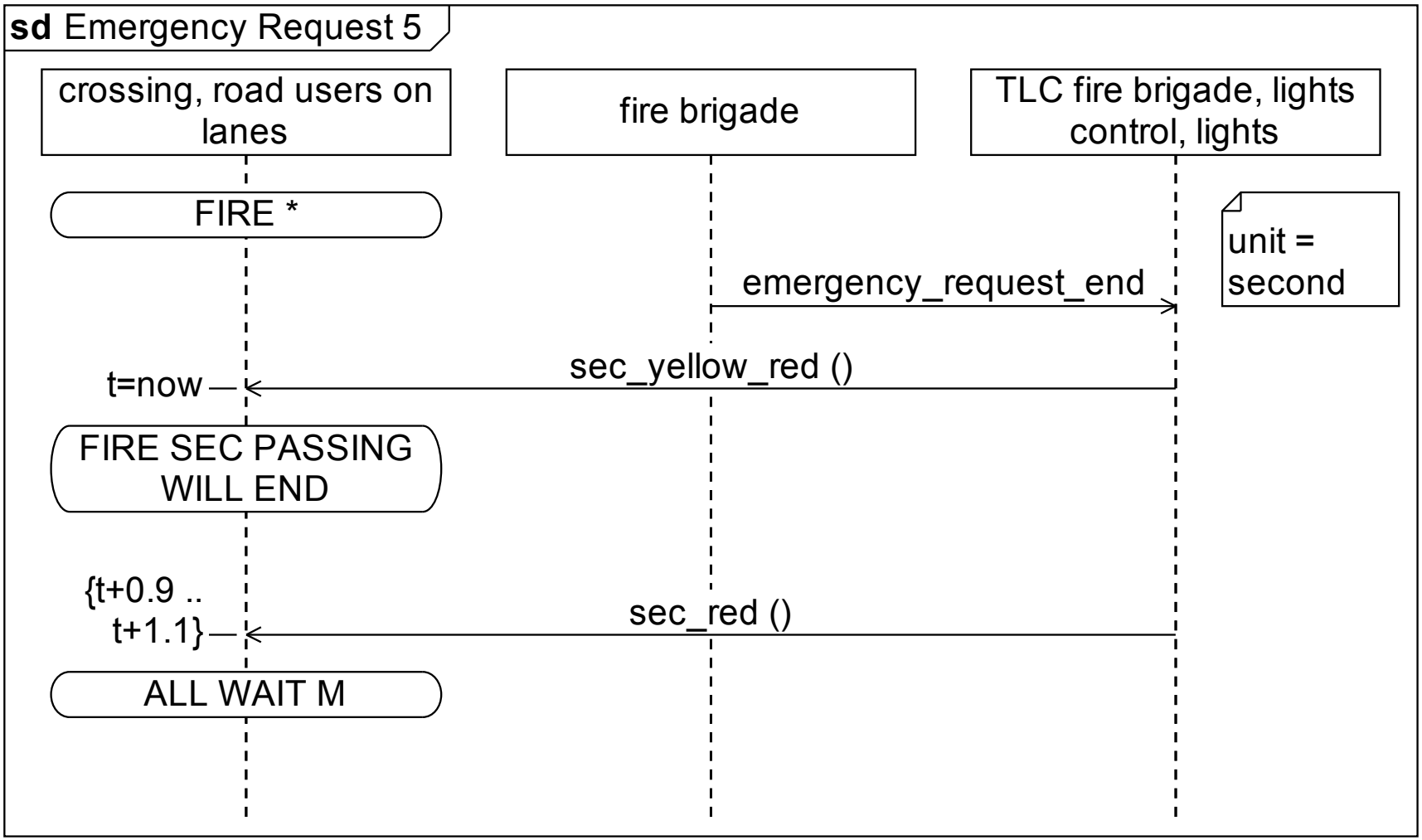
Sequence diagrams for EmergencyRequestSecondaryRoadPassing III



Sequence diagrams for EmergencyRequestSecondaryRoadPassing IV



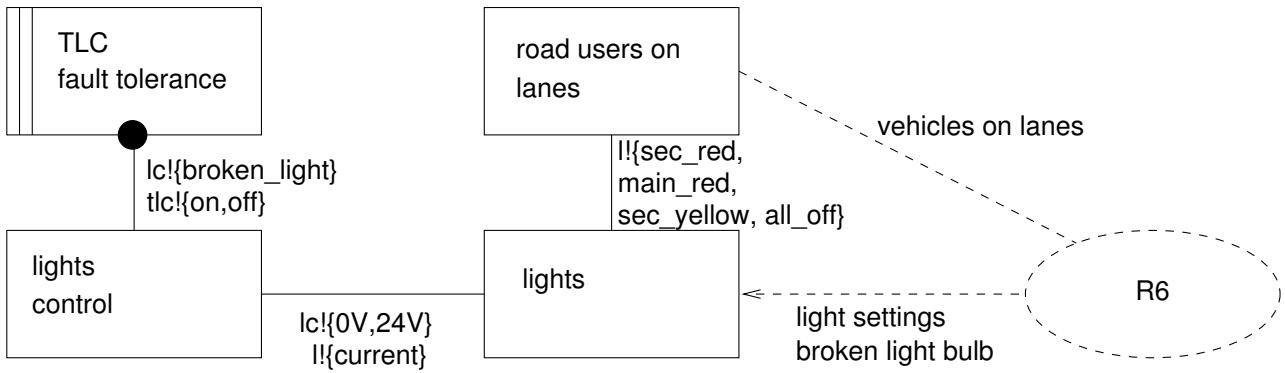
Sequence diagrams for EmergencyRequestSecondaryRoadPassing V



Sequence diagrams for EmergencyRequestSecondaryRoadPassing VI

- The emergency request can occur at any time; therefore all possible starting states have to be considered.
- The star (*) indicates that the diagram can be applied for all states, whose name begins with the given string.
- The domains *crossing* and *road users on lanes* are merged.
- The domains *TLC fire brigade*, *lights control*, and *lights* are merged to express requirements.

BrokenLightSafeState Problem Diagram I



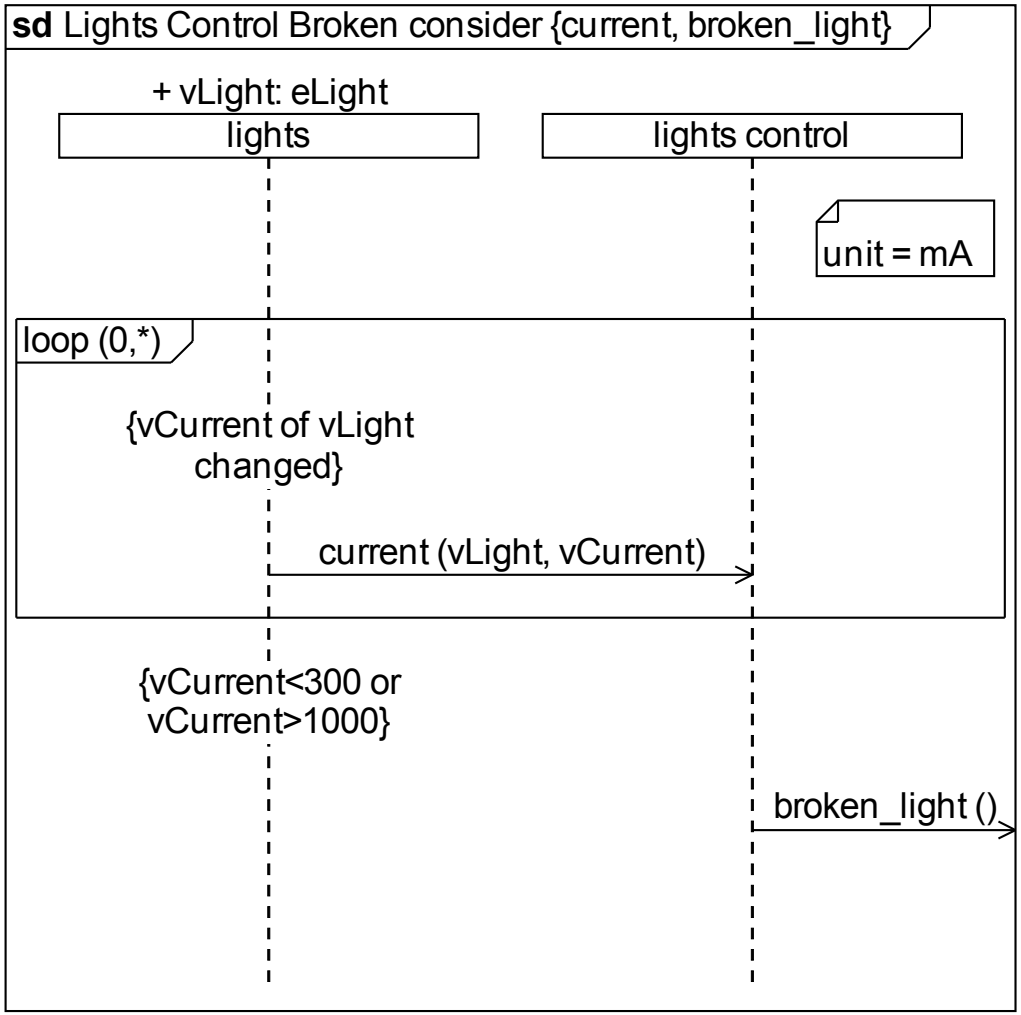
R6 : In case of a broken light bulb the traffic lights should blink in yellow for the secondary road, after all red lights have been switched on for a period of time.

BrokenLightSafeState – Derive Specification I

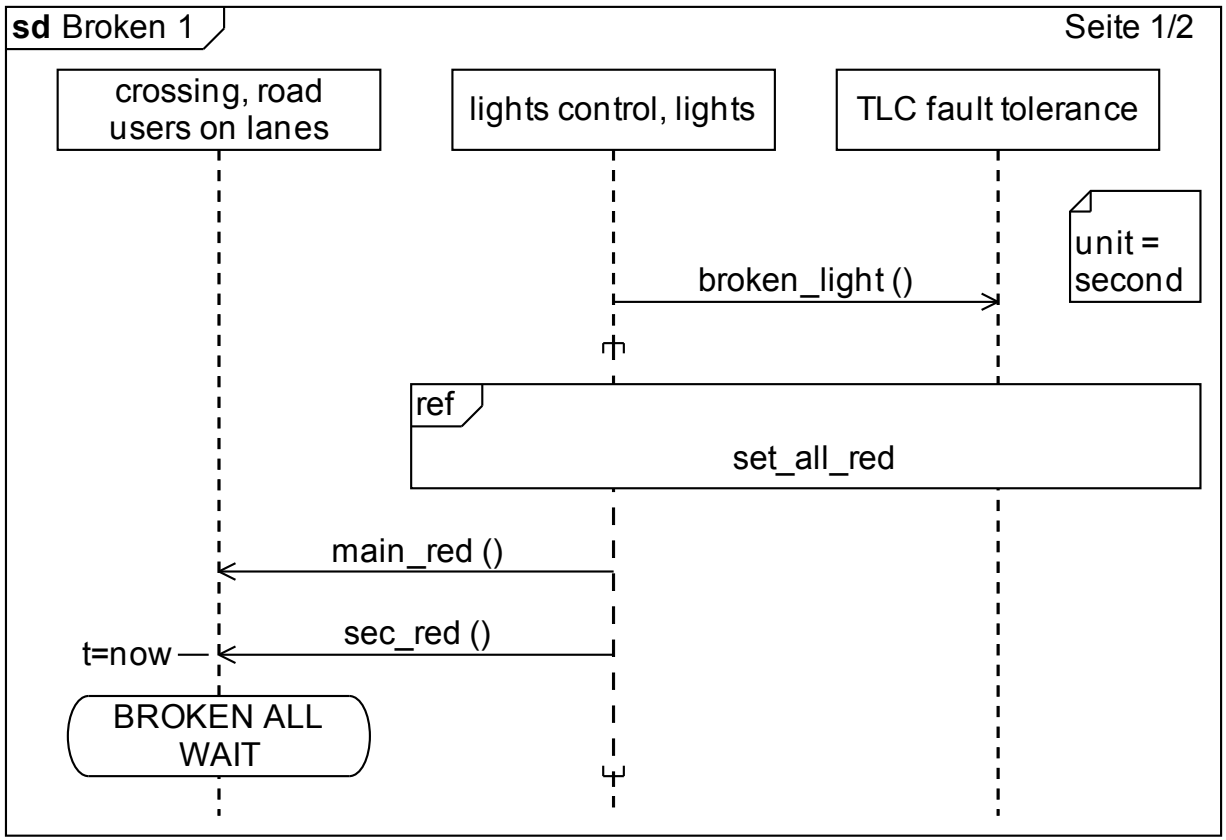
broken light bulb is no interface phenomenon of the machine (controlled by the environment, not observable by the machine). $\Rightarrow$ Transformed into *current* not beween 300 mA and 1 A using $F7$.

S6 In case of a *current* below 300 mA or above 1 A for a light bulb that is switched on, the traffic lights should blink in yellow for the secondary road ($s\_yellow(24)$, $s\_yellow(0)$), after all red lights have been switched on ($s\_red(24)$ or $m\_red(24)$) for a period of time.

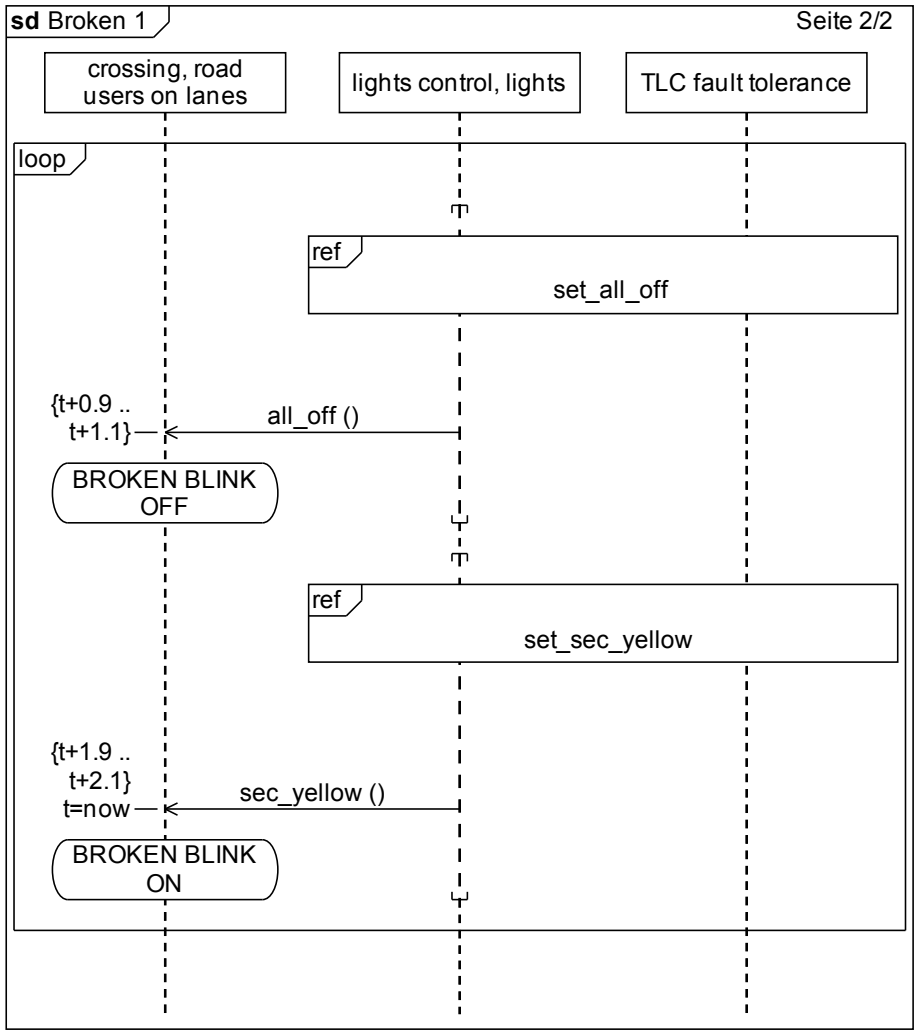
Domain knowledge about *broken_light*()



Sequence diagrams for BrokenLightSafeState I



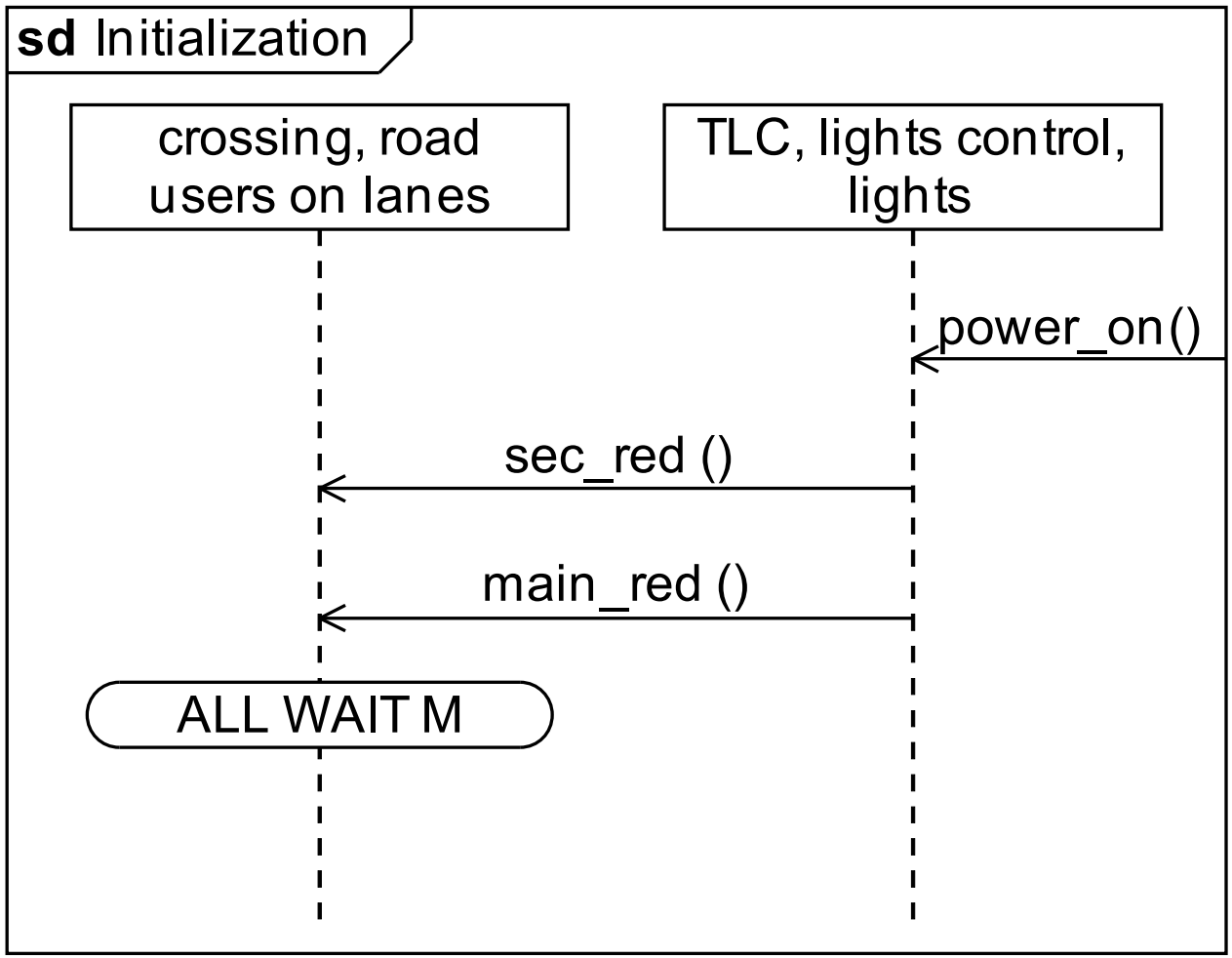
Sequence diagrams for BrokenLightSafeState II



Sequence diagrams for BrokenLightSafeState III

- The phenomenon *broken_light* can occur in every state. It is detected by a very high or very low current for one light bulb.
- Although the domain *lights control* is part of the machine, it is included in this diagram because the phenomenon *broken_light* is more abstract. A diagram using the more technical phenomenon *current* is hard to understand.
- The references *set_all_red* and *set_sec_yellow* are not specified here since the behavior is described in Phase 6.
- The safe state is realized by periodically switching on and off the yellow light of the secondary road. It is not specified how to repair the traffic lights, i.e., how to leave the safe state.

Sequence diagrams for Initialization



Validation I

- no contradictions found in $F \wedge A \wedge S$
- $F \wedge A \wedge S \Longrightarrow R'$
  - $S1 \wedge F1 \wedge F9 \wedge A1 \wedge A4 \Longrightarrow R1$
  - $S2 \wedge F1 \wedge F2 \wedge A1 \Longrightarrow R2$
  - $S3 \wedge F2 \wedge A1 \wedge F4 \wedge F12 \Longrightarrow R3$
  - $S4a \wedge S4b \wedge F1 \wedge F2 \wedge A1 \Longrightarrow R4$
  - $S5a \wedge S5b \wedge A1 \Longrightarrow R5$
  - $S6 \wedge F7 \wedge A1 \Longrightarrow R6$
  - $S7 \wedge F1 \wedge A1 \Longrightarrow R7$
- All requirements are captured. They are assigned to the subproblems as described in Phase 3 and therefore also assigned to the corresponding sequence diagrams.

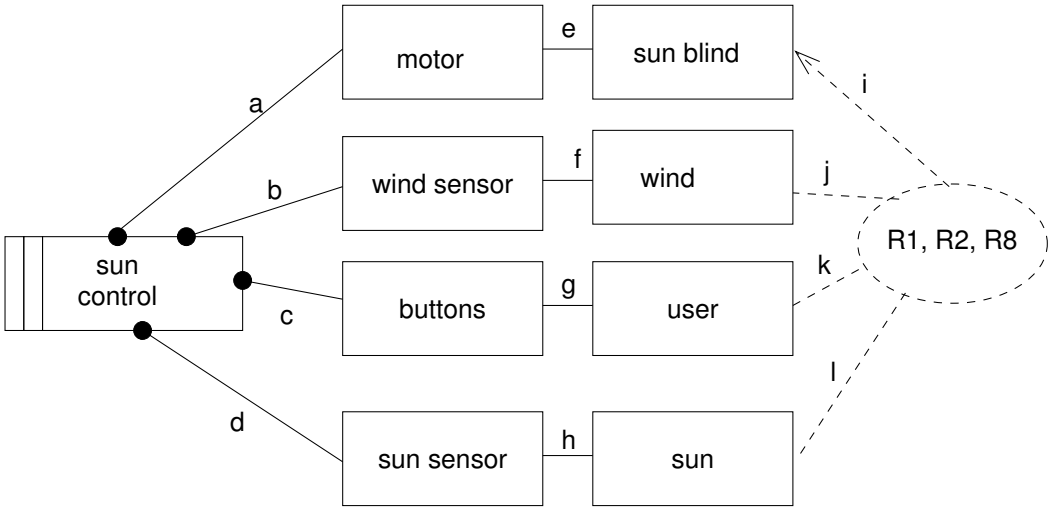
Validation II

- In the sequence diagrams, exactly the phenomena of the problem diagrams are used, and the direction of signals is consistent with the control of the shared phenomena.
- The signals connect domains as connected in the problem diagram.
- The specification can be easily derived from the requirements and the domain knowledge expressed as sequence diagrams.
- The relationships of Phase 3 are consistent with the state invariants:

Subproblem	**Start State**	**End State**
MainRoadPassing	*ALL_WAIT_M*	*ALL_WAIT_S*
SecondaryRoadPassing	*ALL_WAIT_S*	*ALL_WAIT_M*
EmergencyRequest SecondaryRoadPassing	*all*	*ALL_WAIT_M*
BrokenLightSafeState	*all*	*none*

Example 2: sun blind control

SunControl Problem Diagram



R1 If there is *sunshine* for more than one minute but *no heavy wind*, the sun blind will be lowered (*lower sun blind*). *(Parts of R3 included to prevent contradictions. R8 has priority!)*

R2 If there is *no sun shine* for more than 5 minutes, the sun blind will be pulled up (*pull up sun blind*). *(R8 has priority!)*

R8 If the user interacts with the sun blind (*manually opens the sun blind*, *manually closes the sun blind*, *manually rotate fins with positive degree* or *manually rotate fins with negative degree*), *sun shine* and *no sun shine* are ignored within the next 4 hours.

SunControl – Derive Specification

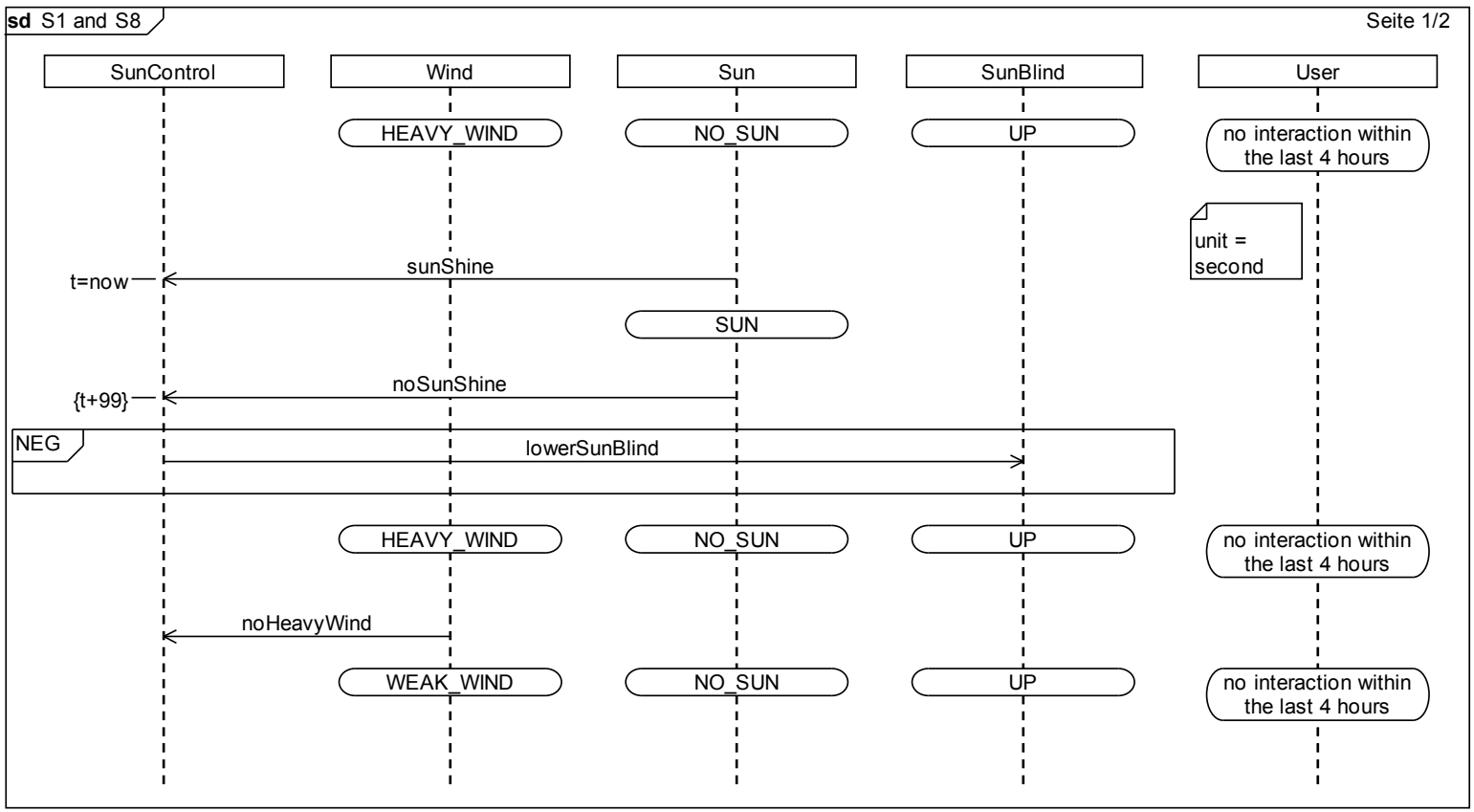
SunShine, *noSunShine*, *heavyWind*, *noHeavyWind*, *manuallyOpenSunBlind*, *manuallyCloseSunBlind*, *manuallyRotateFinsWithPositiveDegree* and *manuallyRotateFinsWithNegativeDegree* are controlled by the environment and observable by the machine.
lowerSunBlind, *pullUpSunBlind* are controlled by the machine and observable by the environment.
R8 expresses conditions that can only be decided in the future.

S1 = R1

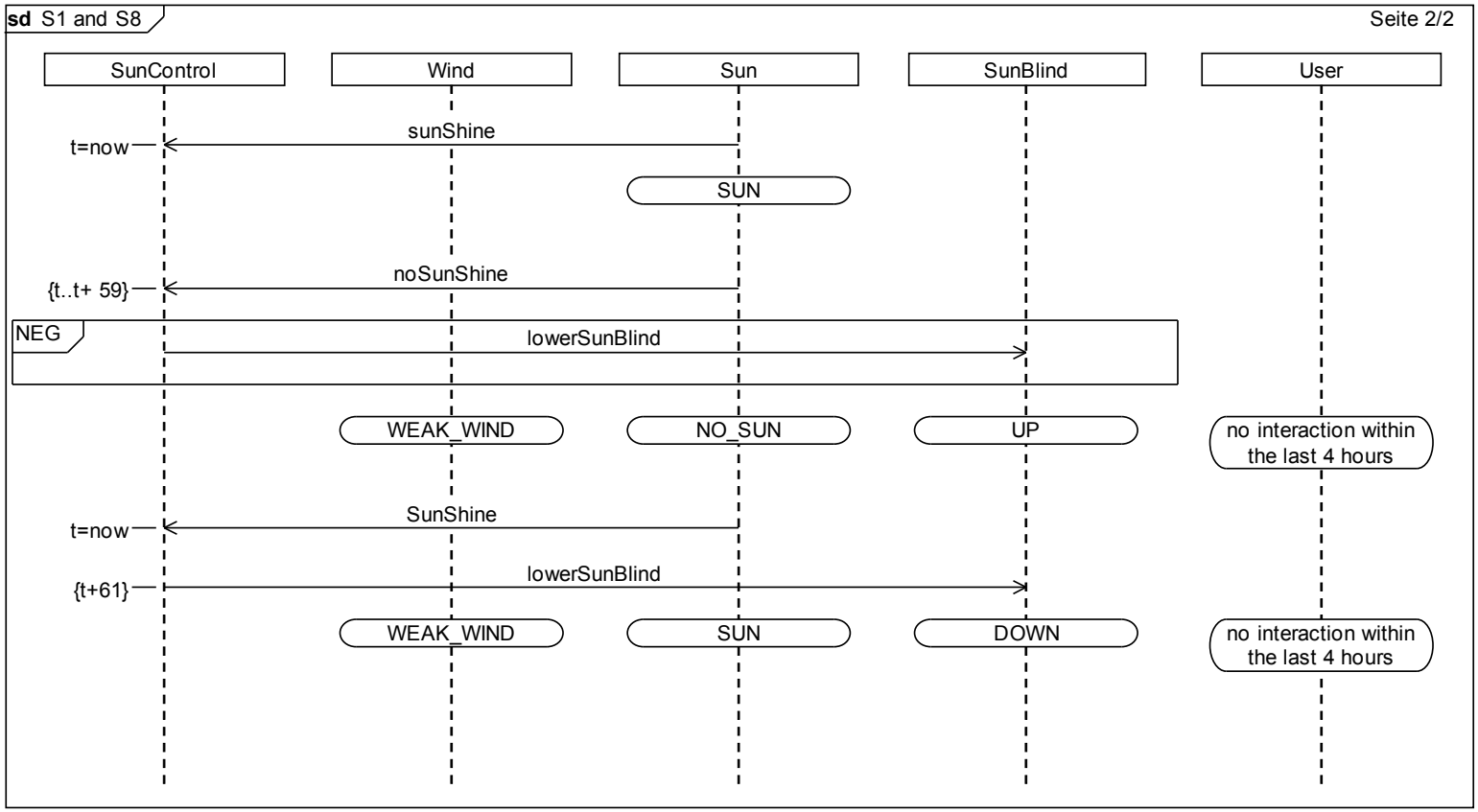
S1 = R2

S8 The sunblind should not be lowered or pulled up (*lowerSunBlind*, *pullUpSunBlind*) when the user interacted with the sun blind (*manually opens the sun blind*, *manually closes the sun blind*, *rotate fins with positive degree* or *rotate fins with negative degree*) within the last 4 hours.

SunControl Sequence Diagrams I

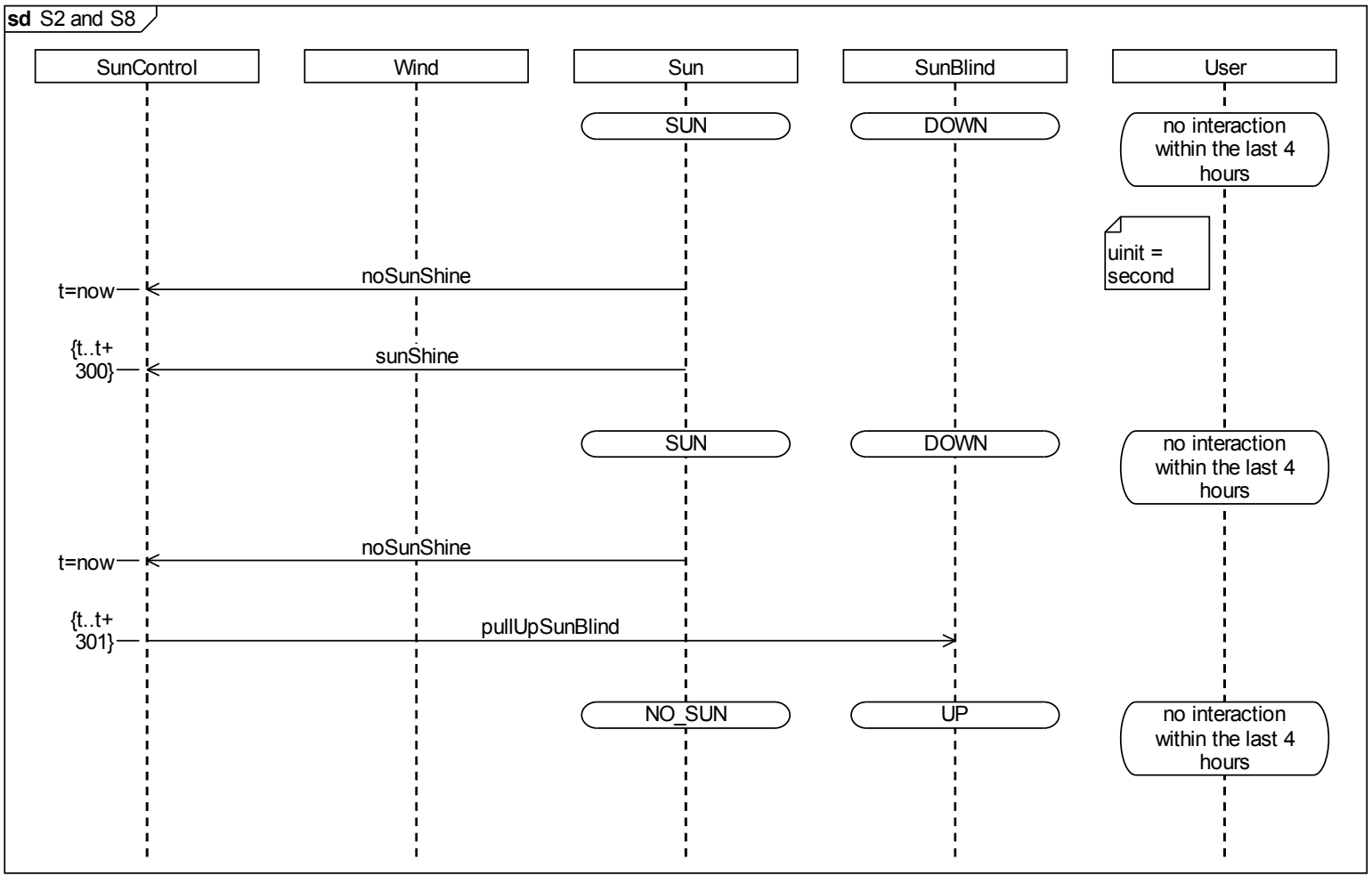


SunControl Sequence Diagrams II

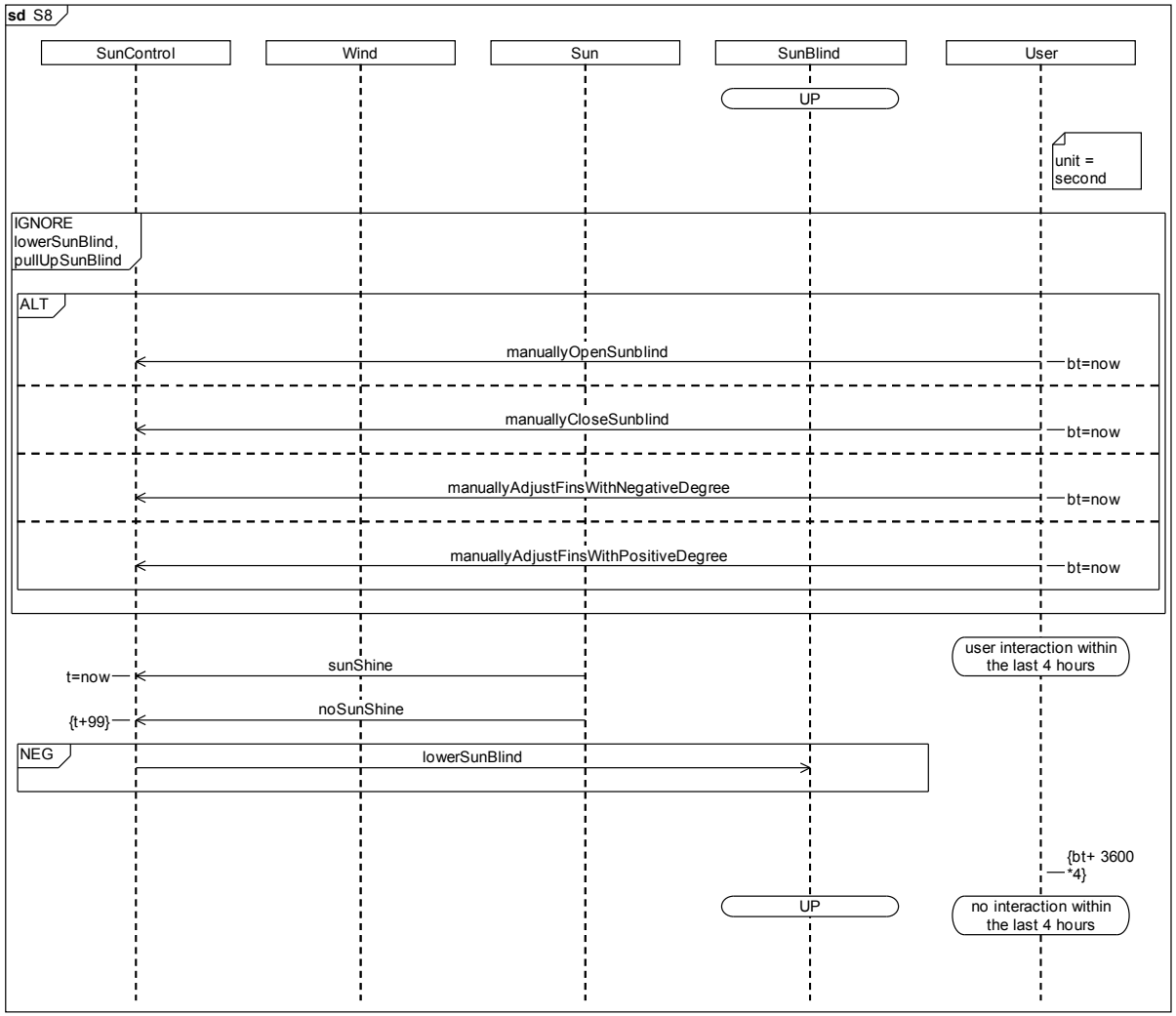


10

SunControl Sequence Diagrams III

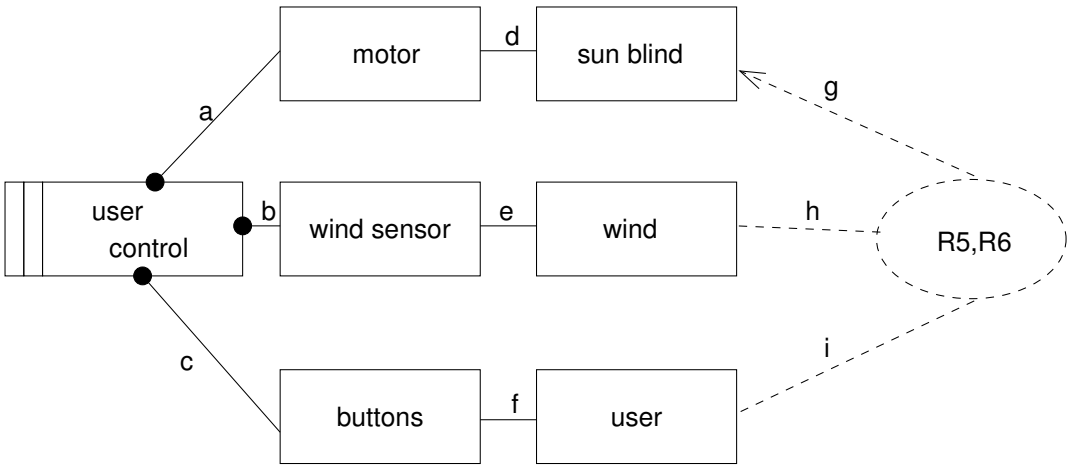


SunControl Sequence Diagrams IV



[10] First the exceptions are specified.

UserControl Problem Diagram



R5 If the user *manually opens the sun blind*, the sun blind will be pulled up (*pull up sun blind*).

R6 If the user *manually closes the sun blind* and there is *no heavy wind*, the sun blind will be lowered (*lower sun blind*). *(Parts of R3 included to prevent contradictions.)*

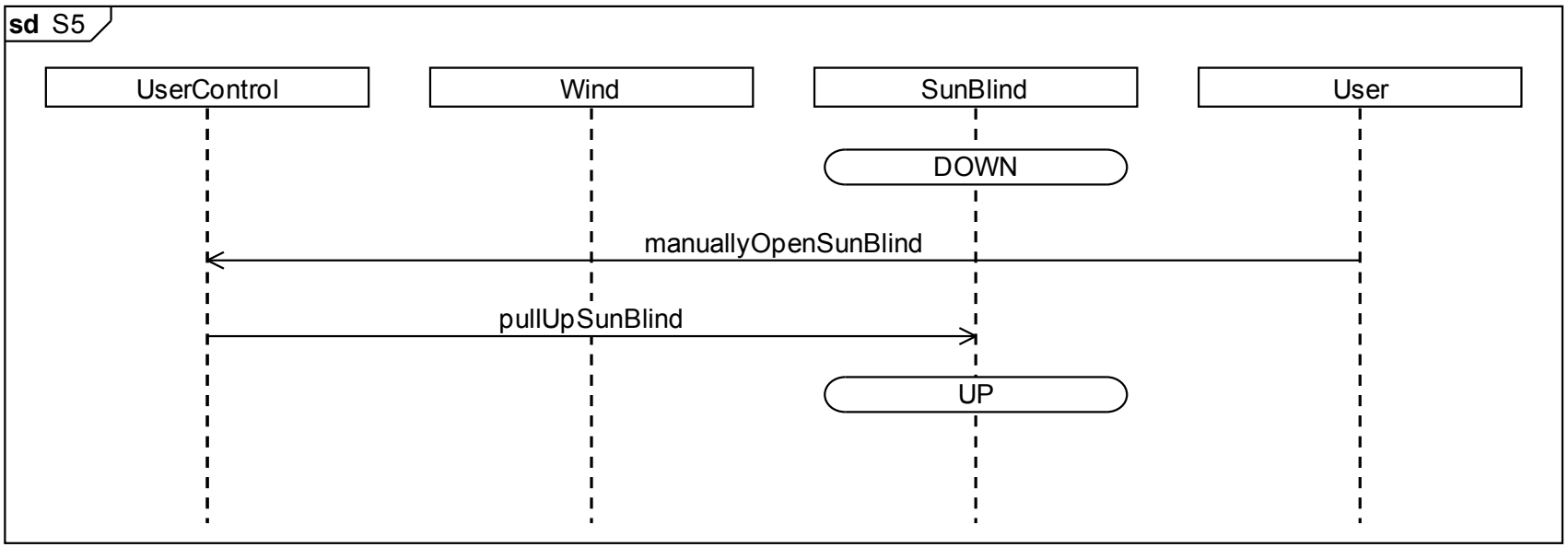
UserControl - Derive Specification I

heavyWind, *noHeavyWind*, *manuallyOpenSunBlind*, and *manuallyCloseSunBlind* are controlled by the environment and observable by the machine.
lowerSunBlind, *pullUpSunBlind* are controlled by the machine and observable by the environment.

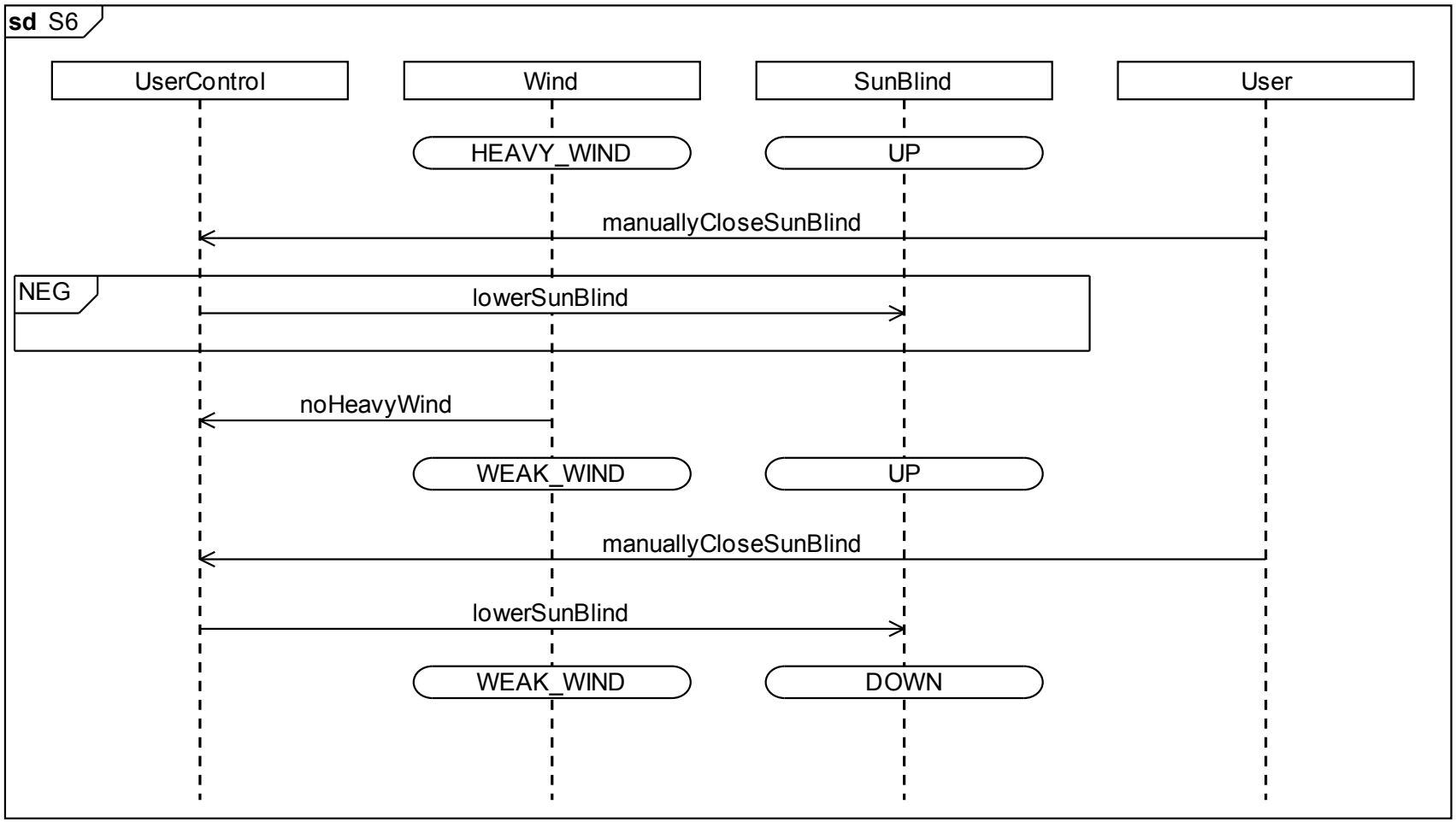
S5 = R5

S6 = R6

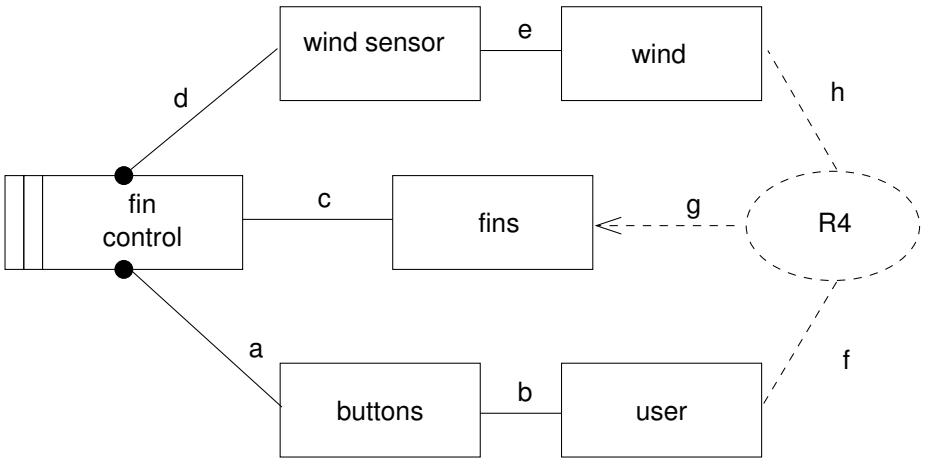
UserControl Sequence Diagrams I



UserControl Sequence Diagrams II



FinsControl Problem Diagram



R4 If there is *no heavy wind* and the user *manually adjusts the fins with positive degree* the fins are rotated with positive degree (*rotate fins with positive degree*).
If there is *no heavy wind* and the user *manually adjusts the fins with negative degree* the fins are rotated with negative degree (*rotate fins with negative degree*).

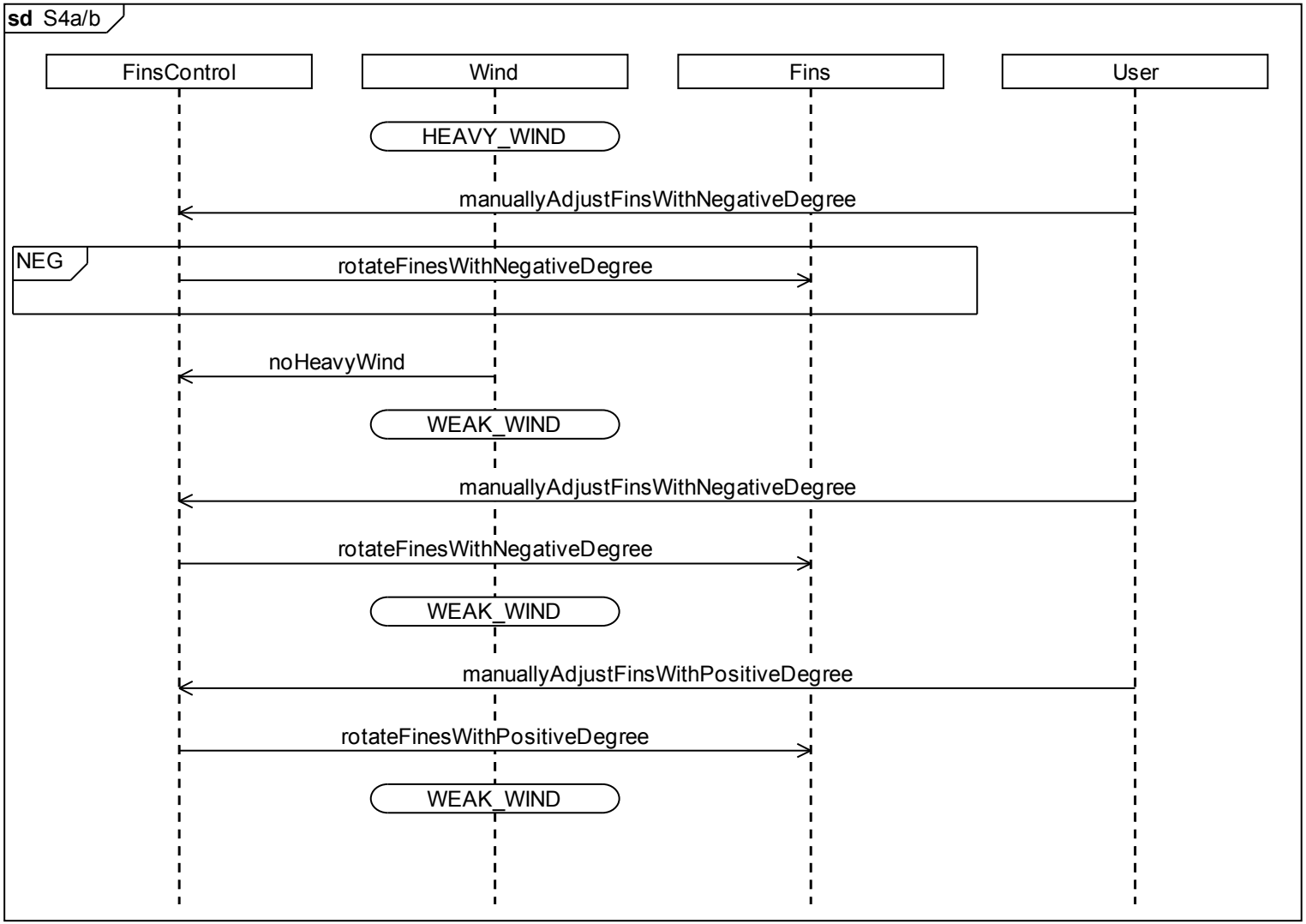
FinsControl - Derive Specification

heavyWind, *noHeavyWind*, *manuallyRotateFinsWithPositiveDegree* and *manuallyRotateFinsWithNegativeDegree* are controlled by the environment and observable by the machine.
rotateFinsWithPositiveDegree and *rotateFinsWithNegativeDegree* are controlled by the machine and observable by the environment.

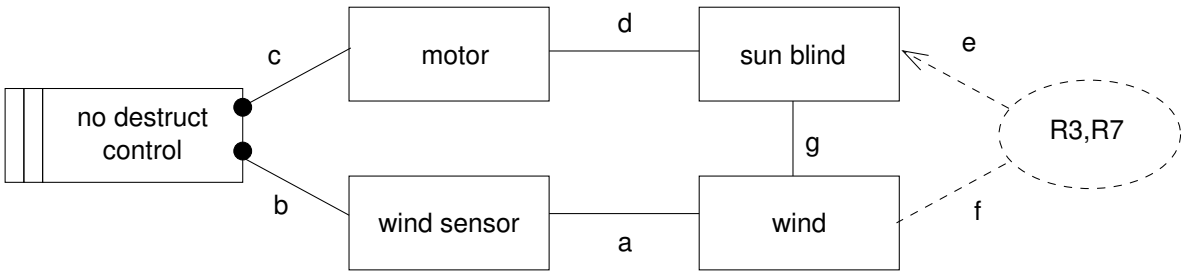
S4a If there is *no heavy wind* and the user *manually adjusts the fins with positive degree* the fins are rotated with positive degree (*rotate fins with positive degree*).

S4b If there is *no heavy wind* and the user *manually adjusts the fins with negative degree* the fins are rotated with negative degree (*rotate fins with negative degree*).

FinsControl Sequence Diagram I



NoDestructControl Problem Diagram



R3 The sun blind should not be destroyed: If there is *heavy wind*, the sun blind will be pulled up (*pull up sun blind*).

R7 The sun blind should not be destroyed: When the sun blind is in its lowest position (*sun blind is lowered*) or in it highest position (*sun blind is pulled up*) the sun blind should stop (*stop sun blind*).

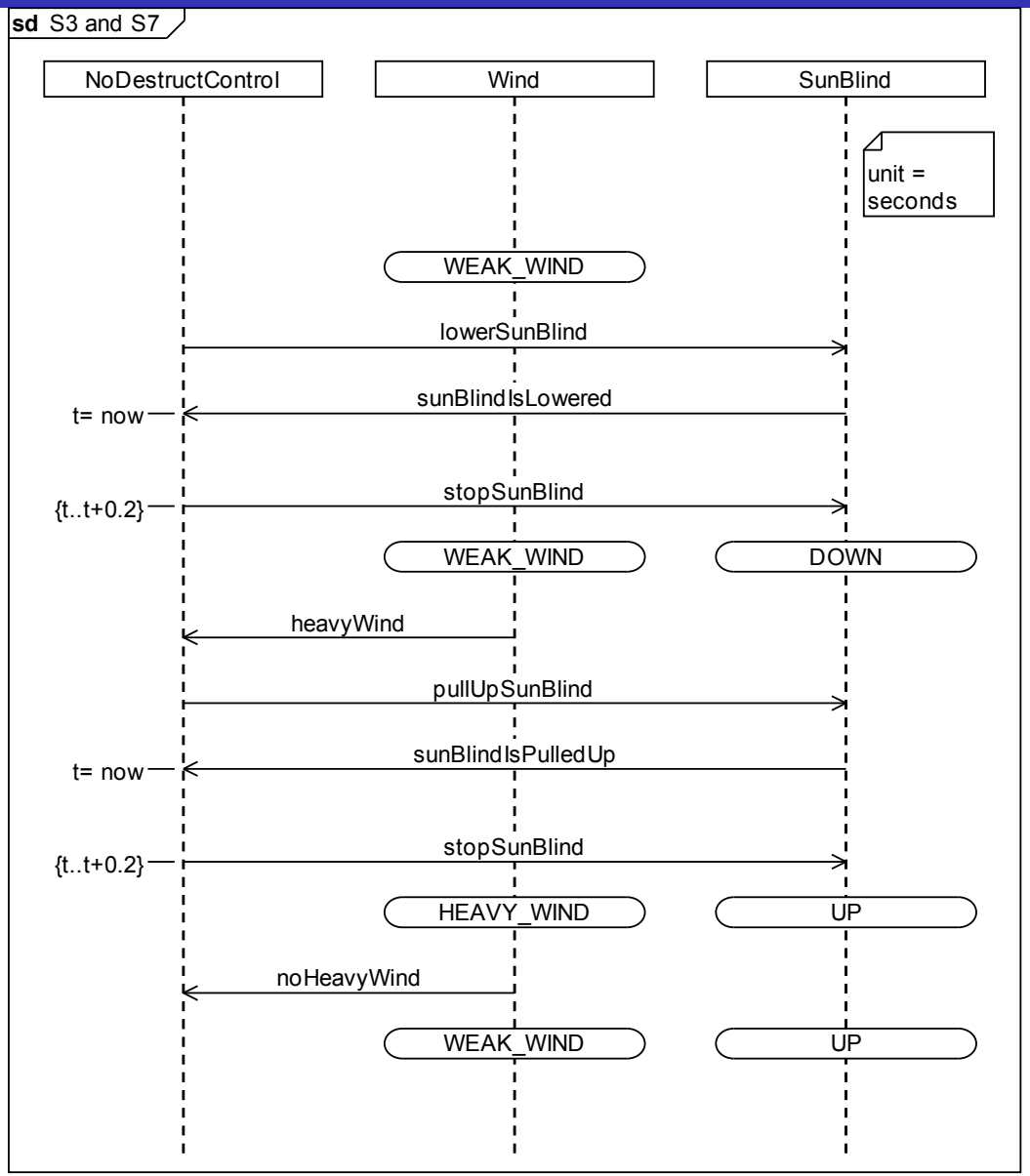
NoDestructControl - Derive Specification

heavyWind, *noHeavyWind*, *sunBlindIsLowered*, and *sunBlindIsPulledUp* are controlled by the environment and observable by the machine. *rotateFinsWithPositiveDegree* and *rotateFinsWithNegativeDegree* are controlled by the machine and observable by the environment.

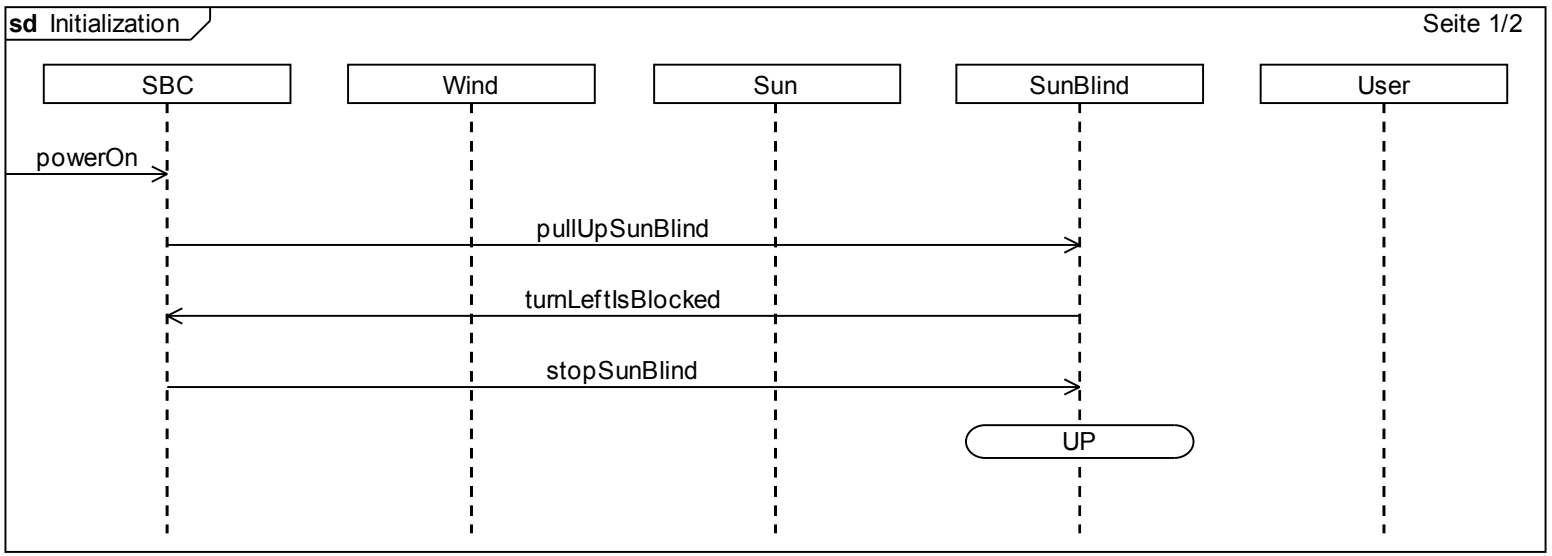
S3 = R3

S7 = R7

NoDestructControl Sequence Diagram



Initialization Sequence Diagram



Validation I

- no contradictions found in $D \wedge S$
- $D \wedge S \Longrightarrow R$
  - $S1 \Longleftrightarrow R1$
  - $S2 \Longleftrightarrow R2$
  - $S3 \Longleftrightarrow R3$
  - $S4a \wedge S4b \Longleftrightarrow R4$
  - $S5 \Longleftrightarrow R5$
  - $S6 \Longleftrightarrow R6$
  - $S8 \Longrightarrow R8$ (other time reference)
  - $S7 \Longleftrightarrow R7$
- all requirements are captured
- in the sequence diagrams exactly the phenomena of the problem diagrams are used (space + letter converted to capital letters)

Validation II

- direction of signals is consistent with control of shared phenomena
- signals connect domains as connected in problem diagram
- all phases are parallel, and therefore it is not checked that the relationships are consistent with the state invariants

Phase 5: Design global system architecture subproblem

...

4. Derive machine behavior specification for each subproblem
5. Design global system architecture
6. Derive specifications for all components of the global system architecture
7. Design a software architecture for all components of the global system architecture that should be implemented in software
8. Specify the behavior of all components of all software

...

Phase 5: Design global system architecture I

input:	context diagram from Phase 2	ext. Jackson
	problem diagrams from Phase 3	Jackson with dot-notation
	sequences of interactions between machine and environment of all subproblems from Phase 4	sequence diagrams
	expression of the subproblem relationships from Phase 3	grammar
output:	system architecture	composite structure diagram
	perhaps subcomponents (recursively)	composite structure diagrams
	purpose of each component	natural language
	specification of external interfaces	interface classes
	specification of interfaces between the components	interface classes
	technical description of hardware interfaces	natural language, figures
	expression of the subproblem relationships for all components	grammars

Phase 5: Design global system architecture II

validation:	all machine interfaces of the problem diagrams must be captured	
	the signals in the sequence diagrams must be the same as the signals in the external interfaces	
	to each programmable component at least one problem diagram must be associated	
	each problem diagram must be associated to at least one component	
	all domains in the problem diagrams being part of the machine must be associated to a component	
	each machine domain in the context diagram must occur in the architecture	
	purpose must be consistent with the associated requirements	
	the grammar for each component must describe a subset of the grammar in Phase 3	

Notations and concepts

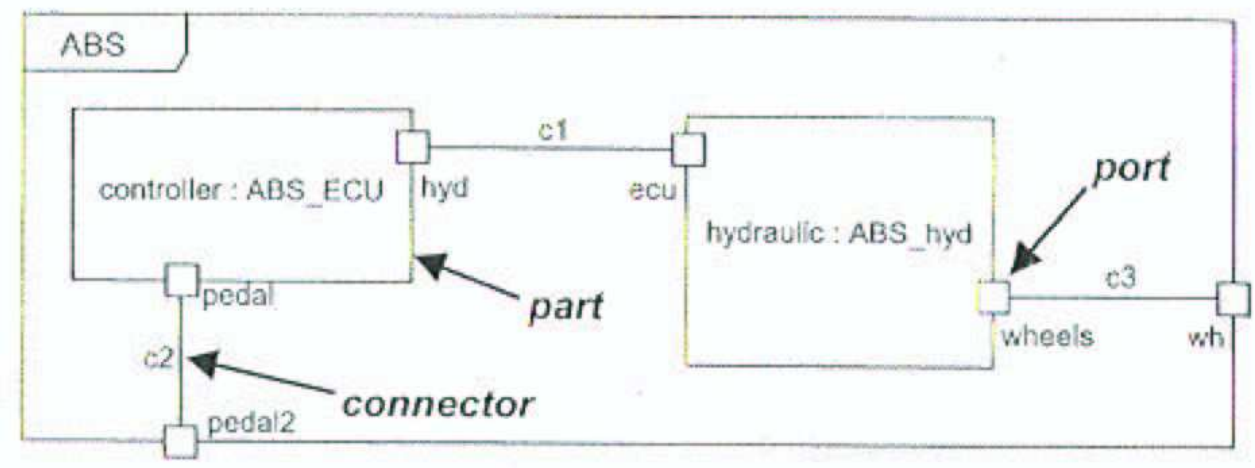
- UML composite structure diagrams

Composite structure diagrams

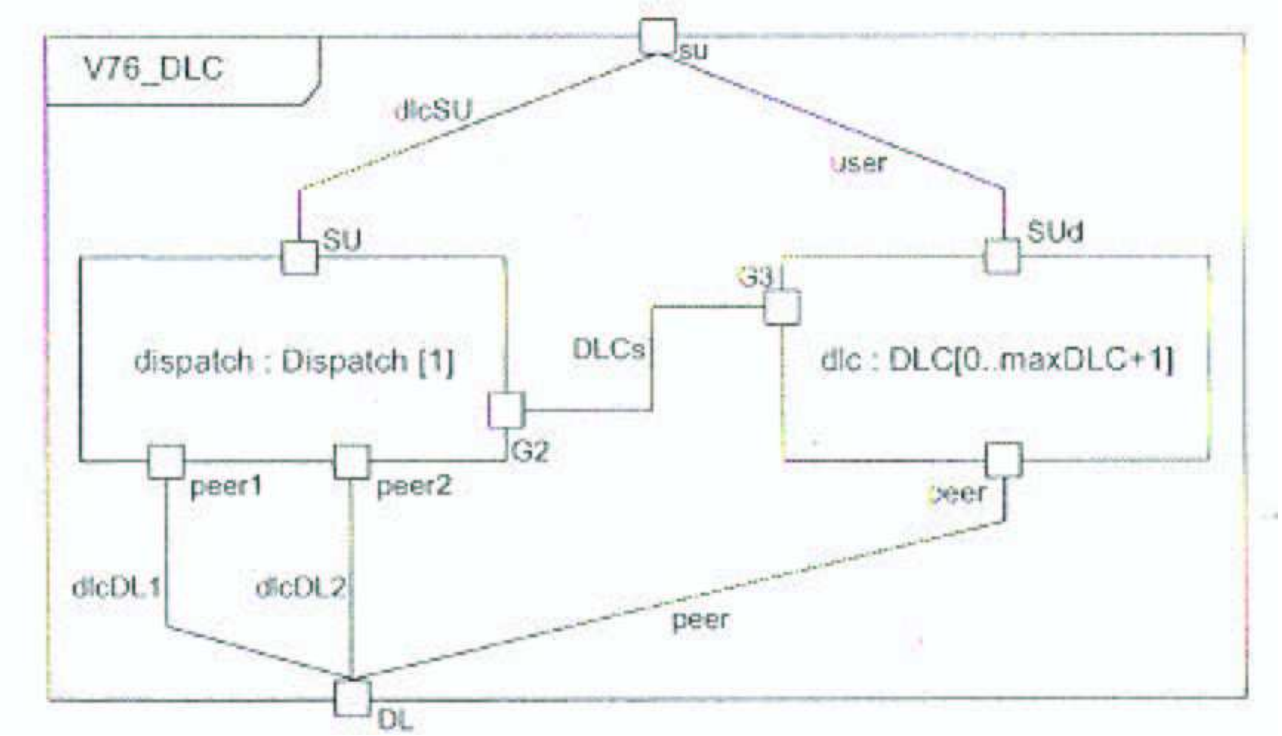
- Represent internal structure of a component
- Also called architecture diagrams
- Answers question "How are components structured and how do they work together"?

Basic elements

- parts: named rectangles, denote architectural components
- ports: small rectangles, denote interaction points of a part with its environment; may have names
- connectors: lines between two ports; may have names



Part multiplicity

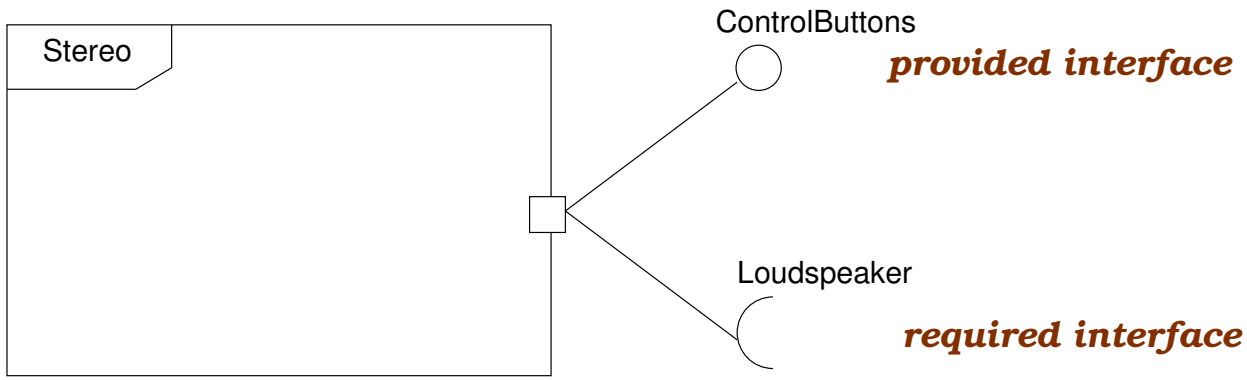


There is exactly one instance of part *dispatch* and between 0 and $maxDLC + 1$ instances of part *dlc*.

Interfaces

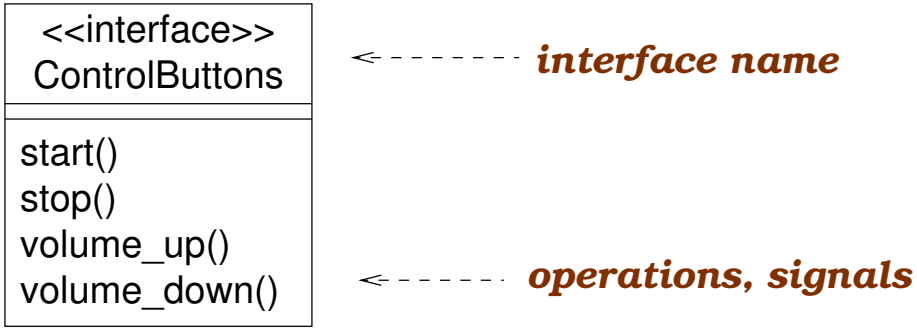
May be associated to ports

- required interface: "socket" notation
- provided interface: "lollipop" notation

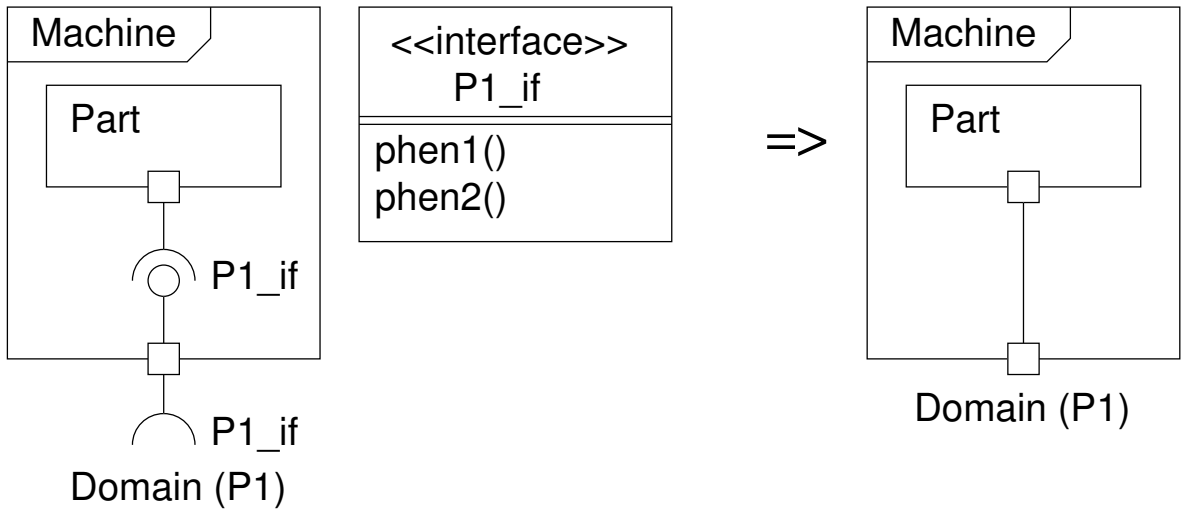


Interface classes

- Serve to describe interfaces
- Notation: class diagrams, with stereotype "<<interface>>"
- Contain no attributes

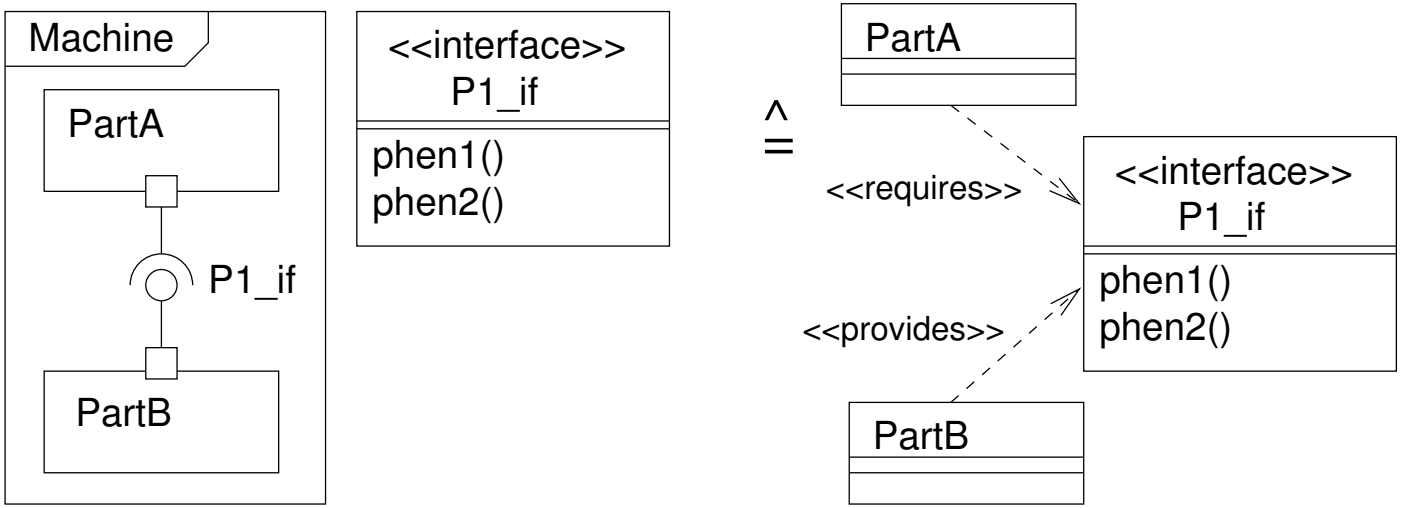


Notation for architectures



Parts = Components = Objects or Classes (Classifiers)

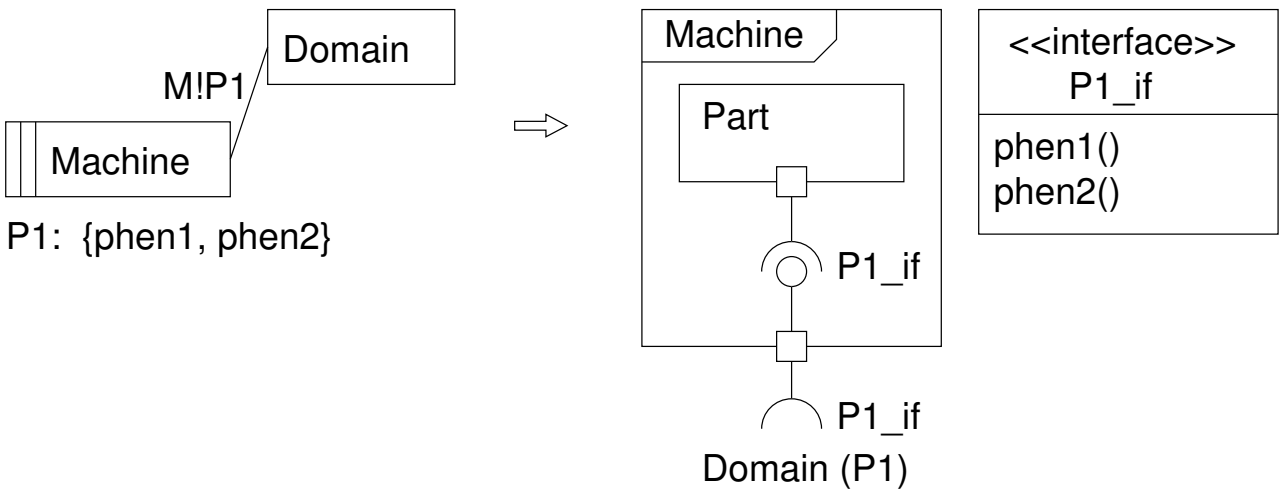
Notation for interfaces



Composite structure diagrams can be transformed into class diagrams.

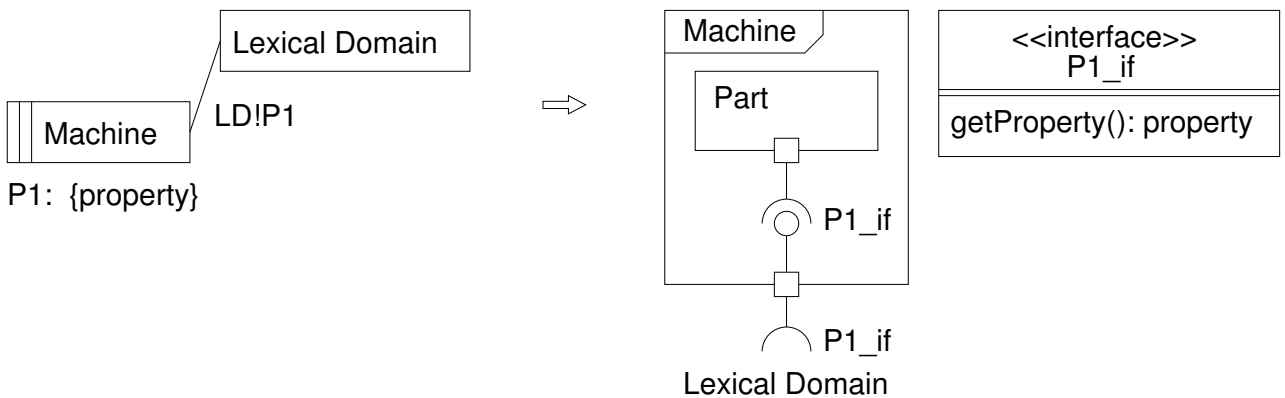
Problem diagrams with phenomena vs. composite structure diagrams with interface classes I

Phenomena controlled by the machine become part of a required interface of the machine.



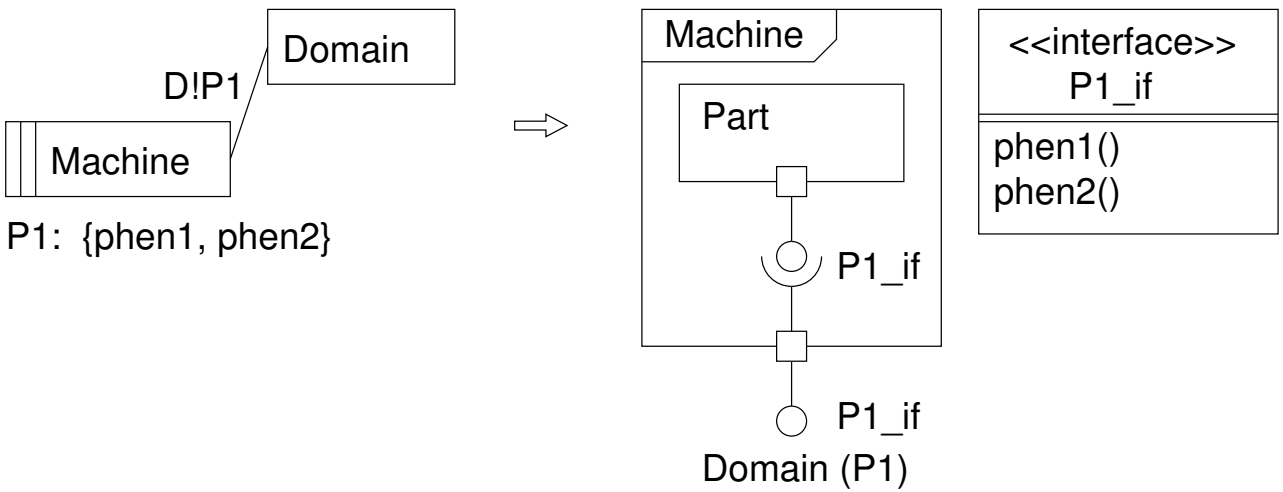
Problem diagrams with phenomena vs. composite structure diagrams with interface classes II

Phenomena controlled by lexical domains in the environment can become part of a required interface of the machine, if the lexical domain returns a value.



Problem diagrams with phenomena vs. composite structure diagrams with interface classes III

Phenomena controlled by the environment become part of a provided interface of the machine.



Phase 5: Design global system architecture I

input:	context diagram from Phase 2	ext. Jackson
	problem diagrams from Phase 3	Jackson with dot-notation
	sequences of interactions between machine and environment of all subproblems from Phase 4	sequence diagrams
	expression of the subproblem relationships from Phase 3	grammar
output:	system architecture	composite structure diagram
	perhaps subcomponents (recursively)	composite structure diagrams
	purpose of each component	natural language
	specification of external interfaces	interface classes
	specification of interfaces between the components	interface classes
	technical description of hardware interfaces	natural language, figures
	expression of the subproblem relationships for all components	grammars

Phase 5: Design global system architecture II

validation:	all machine interfaces of the problem diagrams must be captured	
	the signals in the sequence diagrams must be the same as the signals in the external interfaces	
	to each programmable component at least one problem diagram must be associated	
	each problem diagram must be associated to at least one component	
	all domains in the problem diagrams being part of the machine must be associated to a component	
	each machine domain in the context diagram must occur in the architecture	
	purpose must be consistent with the associated requirements	
	the grammar for each component must describe a subset of the grammar in Phase 3	

Executing Phase 5 I

- Find out which hardware and software components are necessary.
  - Existing components can become parts of the machine. We distinguish between programmable components (e.g., Microcontroller, Embedded PC with Operating System) and hardware components (e.g., Network and Interconnection Components, Analog-Digital-Converter, Clocks). (Software components will be considered in Phase 7).
  - If distributed processing is required, several components being part of the machine are necessary.
  - Domains in the problem diagrams that are part of the machine (marked with a big dot) will become separate components inside the architectual diagram.

Executing Phase 5 II

- Add ports with their provided and required interfaces.
  - The signals and parameters of the operations for the external interfaces can be extracted from the sequence diagrams.
  - The other interfaces must be designed according to the desired functionality of the connected components.
- In addition to the interface description using interface classes, the technical realization of the interfaces must be described. Natural language or figures from the application domain can be used for these technical descriptions.
- For each component, its purpose should be described in one or two sentences. This description must be clear enough to distinguish between the different components.

Executing Phase 5 III

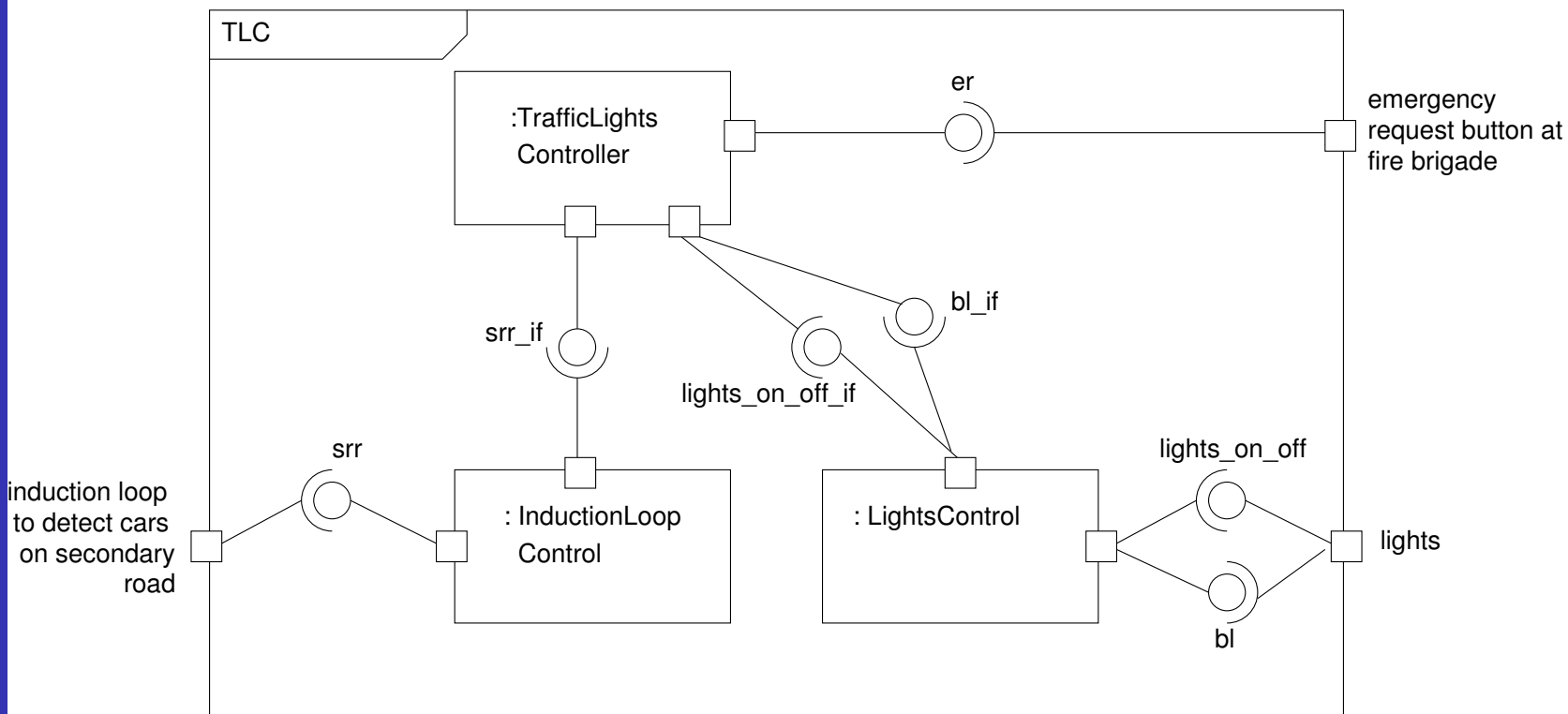
- State for each programmable component which (parts of) subproblems are solved by the component and what their relatenships are.
  - The subproblem relationships are described for each component with the same notation as introduced in Phase 3.
  - Parallel problems constraining different domains can be easily distributed to different components.
  - Sequential and alternative problems must be associated to the same component or a new component must be introduced that decides which of the machines should be activated.

Remarks

- For all programable components, the architecture diagram will be refined in Phase 7.
- For hardware components to be developed, the architecture diagram is the starting point for the hardware development.
- Clock components should only be included if the programable component does not provide a clock signal or timer functionality and at least one time-dependent requirement exists. Operating systems often provide some kind of timer functionality. In this case, all software components can use the given functionality, and no dedicated interface for the clock signal must be described.

Example 1: traffic light control

TrafficLightsControl System Architecture



Purpose of each component

TrafficLightsController Decides on the signaling shown by the physical traffic lights.

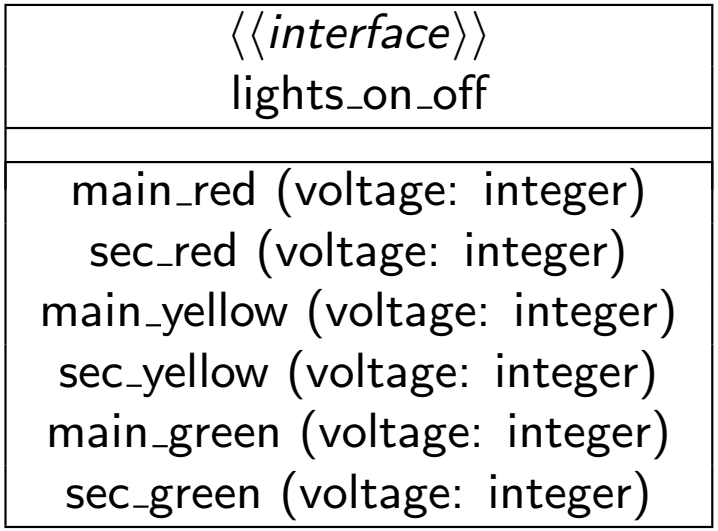
LightsControl Connects the *TrafficLightsController* to the physical lights. We buy this component.

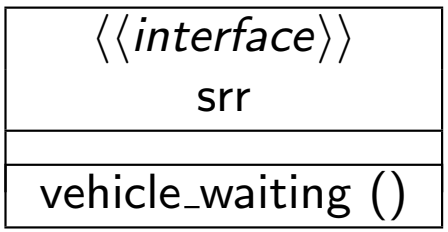
InductionLoopControl Connects the *TrafficLightsController* to the induction loop. We buy this component.

Subcomponents

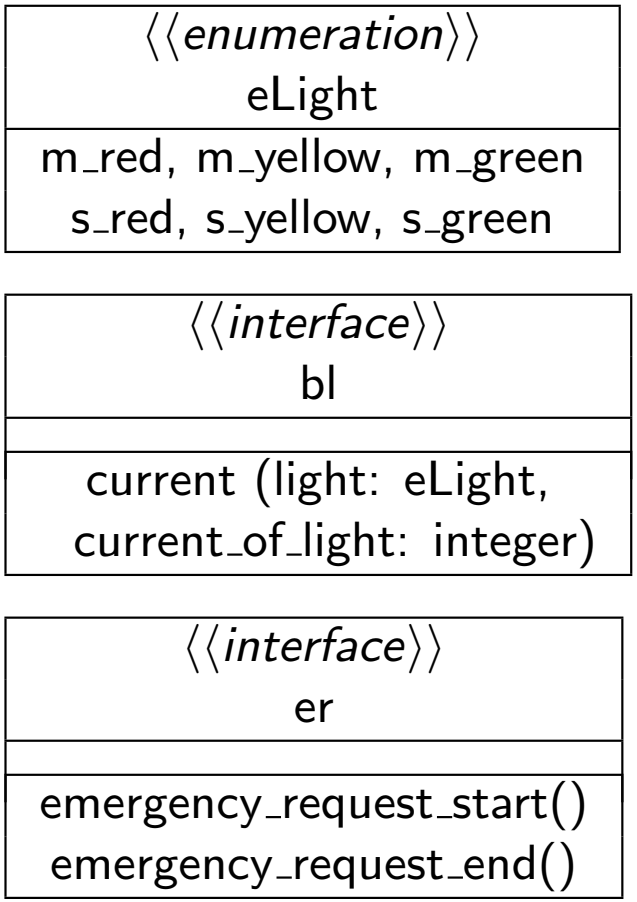
No subcomponents are necessary for this problem.

TrafficLightsControl System Architecture - External Interfaces I

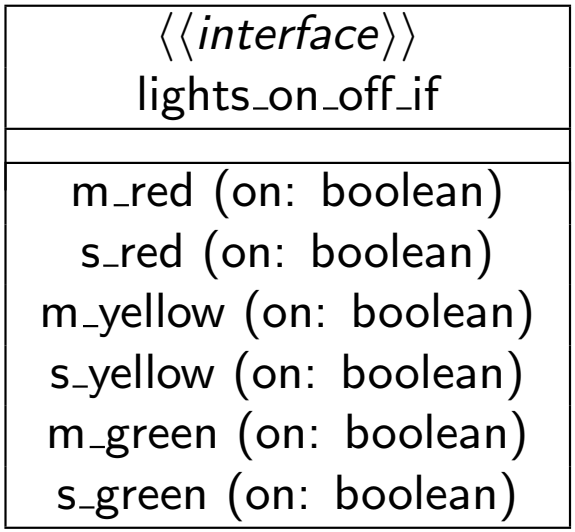


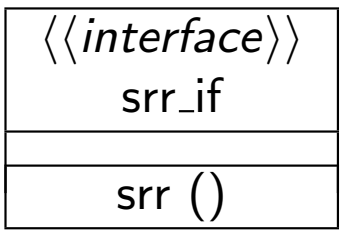


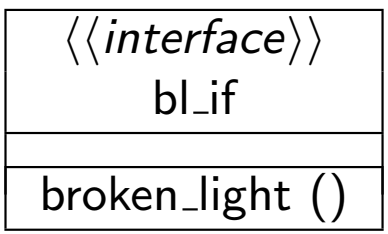
TrafficLightsControl System Architecture - External Interfaces II



TrafficLightsControl System Architecture - Internal Interfaces







TrafficLightsControl System Architecture – Technical interface description

Example: Interface *bl*

The signal of the interface *bl* describes the measurement of the electric current for each light. If the electric current is not in the range from 300 mA to 1000 mA, the signal *broken light()* of the interface *bl* is sent to the *TrafficLightsController* as a single event.

Subproblem relationships

We decided to implement all subproblems in the component *TrafficLightsController*. The subproblem relationship of the component *TrafficLightsController* is therefore the same as for the overall machine (Step 3).

The other components are hardware components.

Validation I

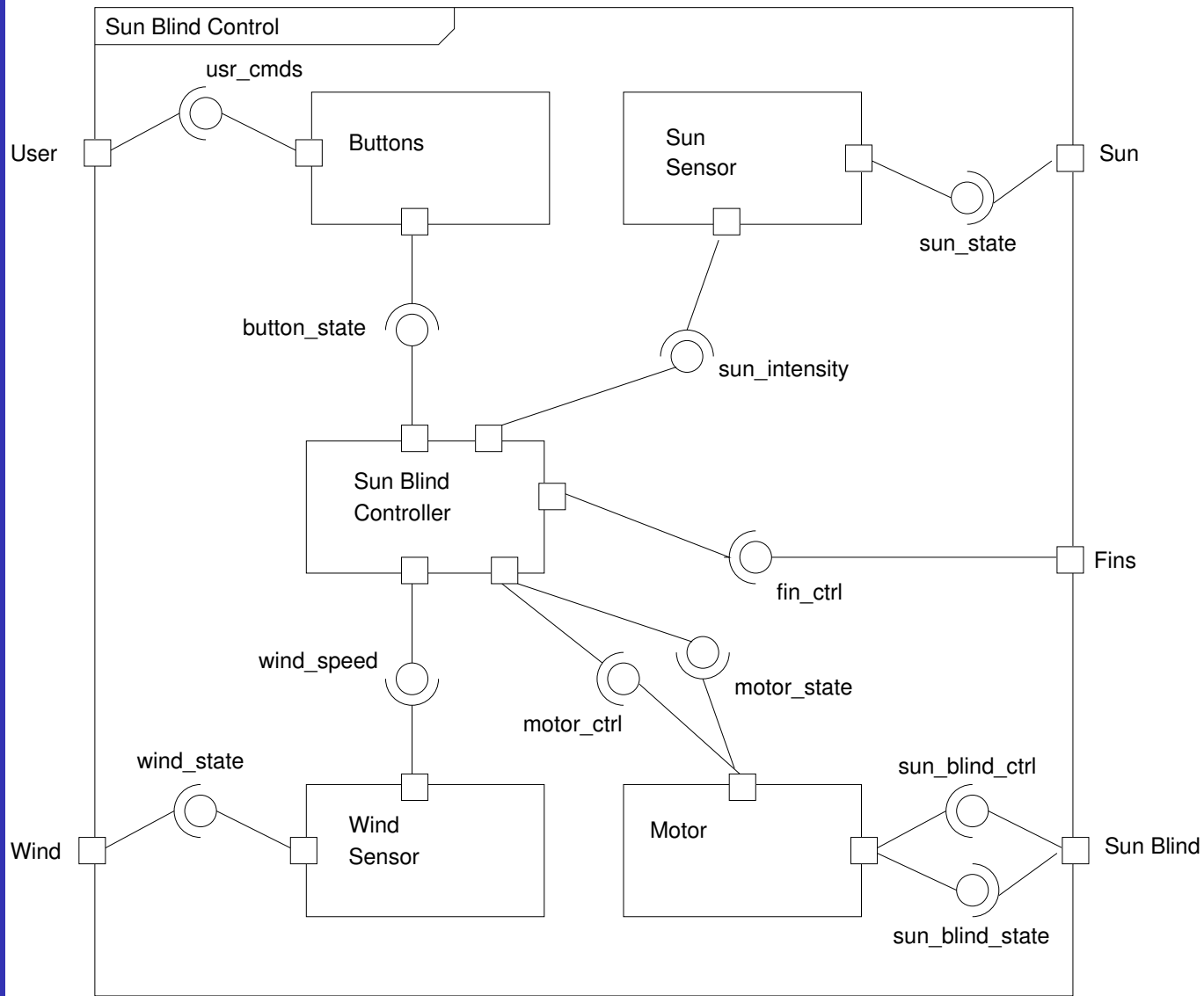
- The external interfaces of the components cover the interfaces of all problem diagrams.
- The signals in the sequence diagrams are the same as in the external interfaces.
- All subproblems are associated to the component *TrafficLightsController*. (At least one must be associated.)
- All domains in the problem diagrams being part of the machine are associated to a component (domain *lights control* – component *LightsControl*, domain *induction loop control* – component *InductionLoopControl*).
- *LightsControl* and *InductionLoopControl* are hardware components.

Validation II

- Only one machine domain in the context diagram exists. Its structure is given by the architecture.
- The purpose of each component is consistent with the associated requirements.

Example 2: sun blind control

SunControl System Architecture



Purpose of each component I

Button Transforms the user commands into a button state.

SunSensor Measures the sun intensity and transforms it into a lux-value.

WindSensor Measures the speed of the wind and transforms it into a number of pulses per minute proportional to the speed of the wind.

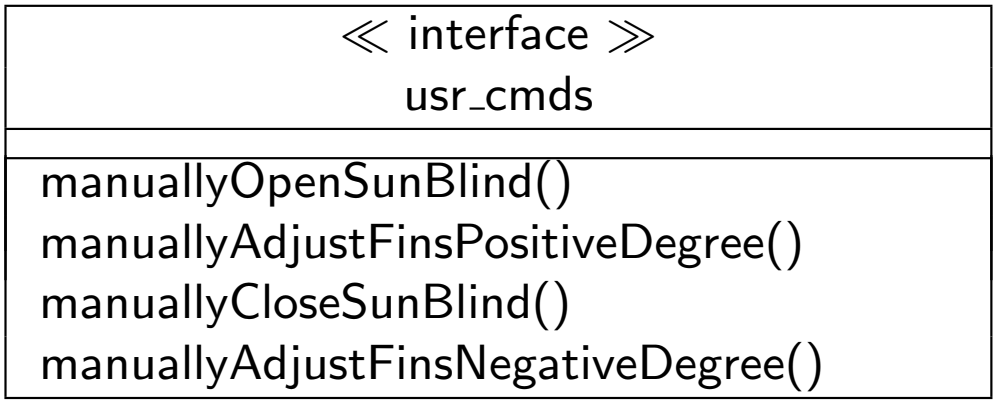
Motor Pulls up and lowers the sunblind according to its turning direction. The direction can be controlled by the *SunBlindController*.

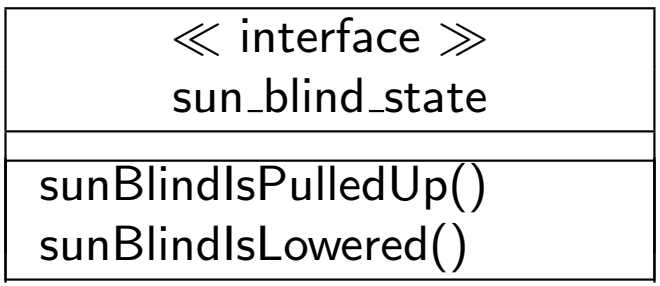
SunBlindController Controlls the sun blind and the fins. It should react to sunshine and user commands, and it prevents the sun blind from taking damage caused by heavy wind.

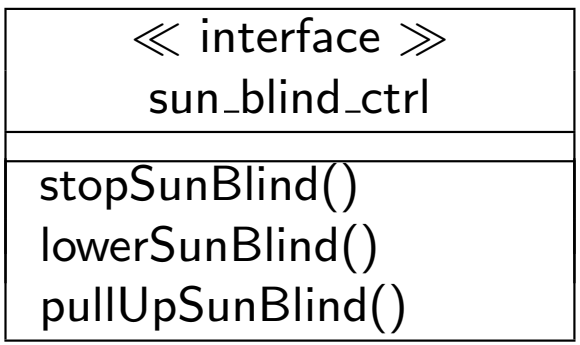
Subcomponents I

No subcomponents are necessary for this problem.

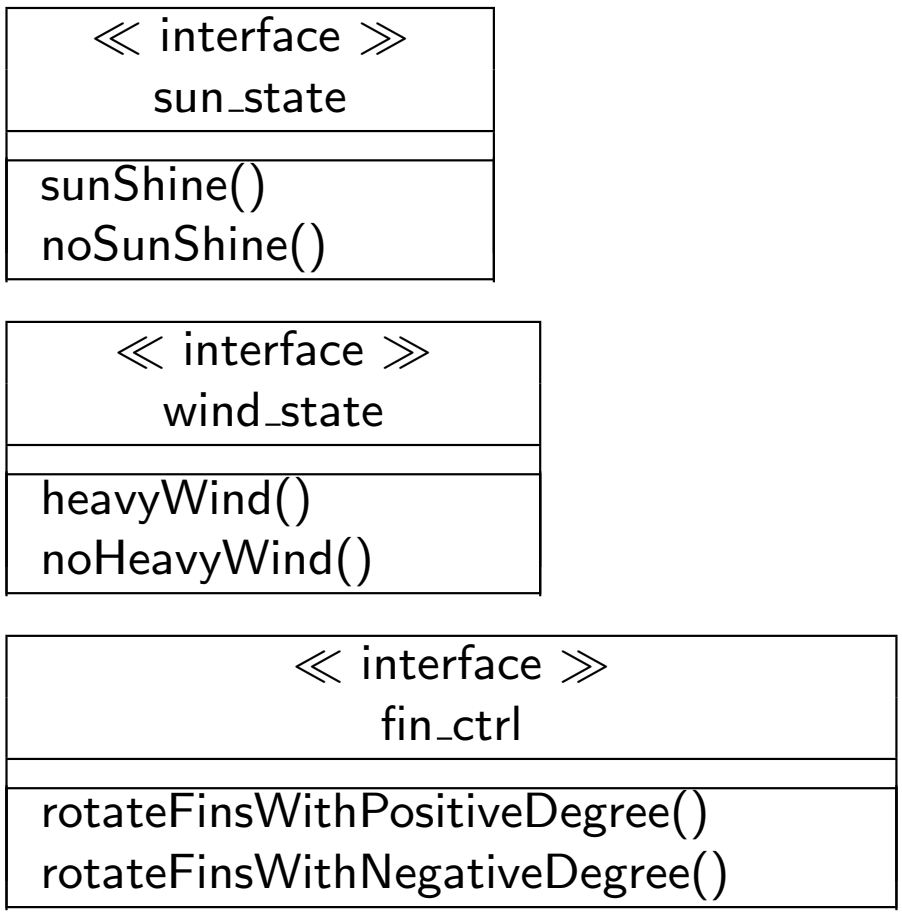
SunControl System Architecture - External Interfaces I



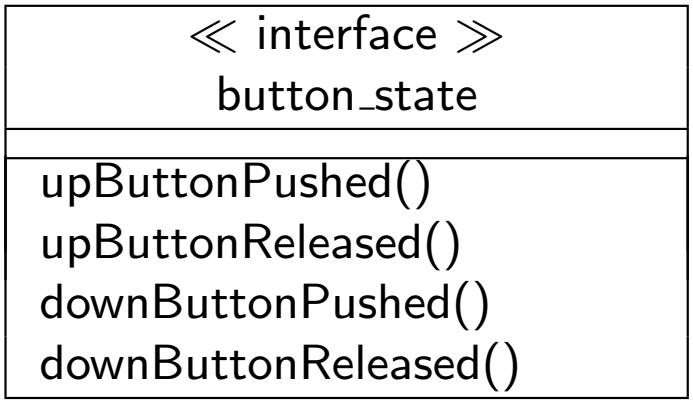


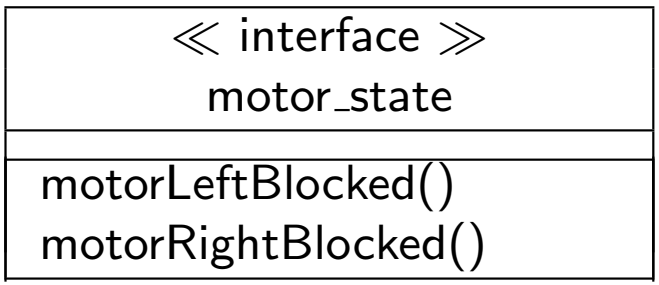


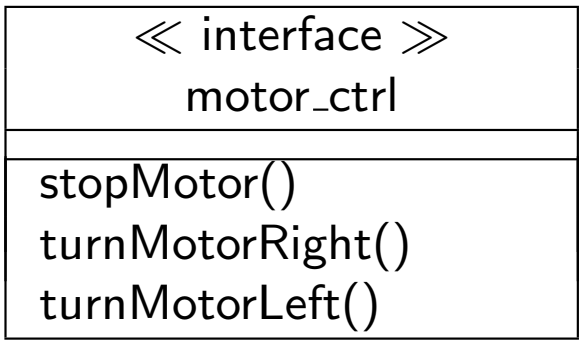
SunControl System Architecture - External Interfaces II



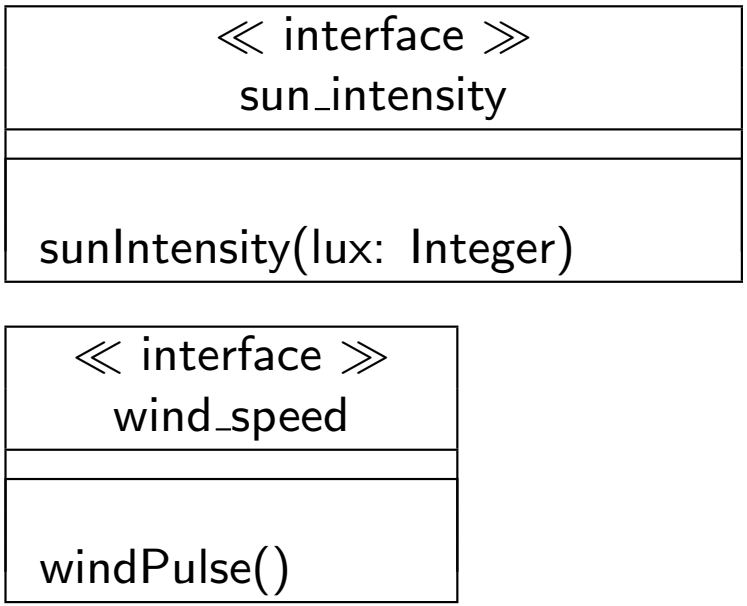
SunControl System Architecture - Internal Interfaces I







SunControl System Architecture - Internal Interfaces II



SunControl System Architecture - Wind sensor description

wind speed	pulses per minute	delay between two pulses
1 km/h	10	6000 ms
2 km/h	20	3000 ms
5 km/h	50	1200 ms
10 km/h	100	600 ms
20 km/h	200	300 ms
50 km/h	500	120 ms
80 km/h	800	75 ms
100 km/h	1000	60 ms
200 km/h	2000	30 ms

Subproblem relationships I

The component *SunBlindController* is the only programmable component.

SunBlindController `<Start> ::= SunControl || NoDestructControl || FinsControl || UserControl`

(R1 – R8)

Validation I

- All machine interfaces of the problem diagrams are captured (e.g., lower sun blind – lowerSunBlind).
- The signals in the sequence diagrams are the same as in the external interfaces.
- To each programmable component at least one problem diagram is associated (see previous slide).
- All problem diagrams are associated to the component *SunBlindController*.
- All domains in the problem diagrams being part of the machine are associated to a component (Buttons – Buttons, SunSensor – SunSensor, WindSensor – WindSensor, Motor – Motor).

Validation II

- Only one machine domain in the context diagram exists (Sun Blind Control). Its structure is given by the architecture.
- The purpose of each component is consistent to the associated requirements.

Embedded Systems

WS 08/09

Maritta Heisel

Maritta.Heisel(AT)uni-duisburg-essen.de

Denis.Hatebur(AT)uni-duisburg-essen.de

University Duisburg-Essen – Faculty of Engineering
Department of Computer Science
Workgroup Software Engineering

Overview of development process (DePES) I

1. Describe system in use
2. Describe system to be built
3. Decompose problem
4. Derive a machine behavior specification for each subproblem
5. Design global system architecture
6. Derive specifications for all components of the global system architecture
7. Design an architecture for all programmable components of the global system architecture that will be implemented in software

Overview of development process (DePES) II

8. Specify the behavior of all components of all software architectures, using sequence diagrams
9. Specify the software components of all software architectures as state machines
10. Implement software components and test environment
11. Integrate and test software components
12. Integrate and test hardware and software

Phase 6: Derive specifications for all components of the global system architecture I

...

5. Design global system architecture
6. Derive specifications for all components of the global system architecture
7. Design a software architecture for all components of the global system architecture that should be implemented in software
8. Specify the behavior of all components of all software architectures, using sequence diagrams
9. Specify the software components of all software architectures as state machines

...

Phase 6: Derive specifications for all components of the global system architecture

For each subproblem:

input:	architecture from Phase 5	composite structure diagrams
	interface specifications from Phase 5	interface classes
	subcomponents (if defined) from Phase 5	composite structure diagrams
	sequences of interactions from Phase 4	sequence diagrams with annotated states or existing technical documentation
output:	interface behavior of all components (test specification)	sequence diagrams with annotated states
validation:	sequence diagrams together must describe the same interface behavior as in Phase 4	
	all signals in the interface classes of Phase 5 must be used in at least one sequence diagram	
	direction of signals must be consistent with the required and provided interfaces of Phase 5	
	signals must connect components as connected in the system architecture of Phase 5	
	it must be possible to map the new states to the states of Phase 4	

Executing Phase 6 I

To create the sequence diagrams, for each subproblem the following steps have to be performed:

- Draw a lifeline for all components of the architecture that are necessary to describe the interface behavior of the subproblem and one or more lifelines for the environment. If the diagram becomes too complex, components can be merged in the sequence diagram, and the interaction between these components must be described separately.
- Alternatively, for each component the behavior can be described separately.
- Describe the interface behavior of all components using the signals from the system architecture (Phase 5). The behavior must refine the behavior described in Phase 4.

Executing Phase 6 II

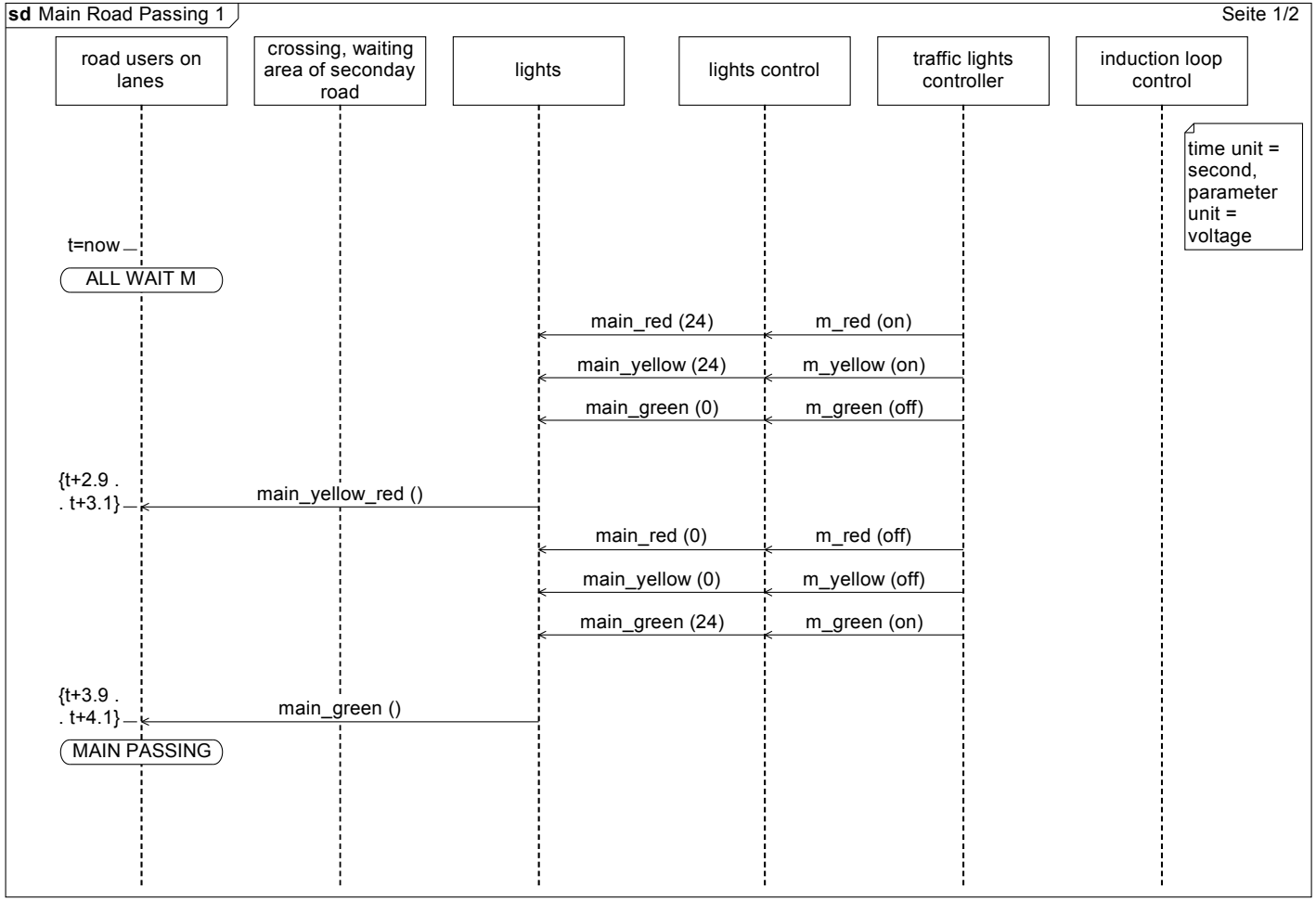
- Add states where they are relevant to describe the behavior.
- Add missing sequence diagrams to describe the behavior for all relevant states for all components.
- Add timing constraints if necessary.
- To describe complex interactions between two components, references to detailed sequence diagrams can be used.
- As for Phase 4: each diagram represents one concrete interaction sequence. Do not try to make the diagrams too general. It is better to draw further diagrams.

Remarks

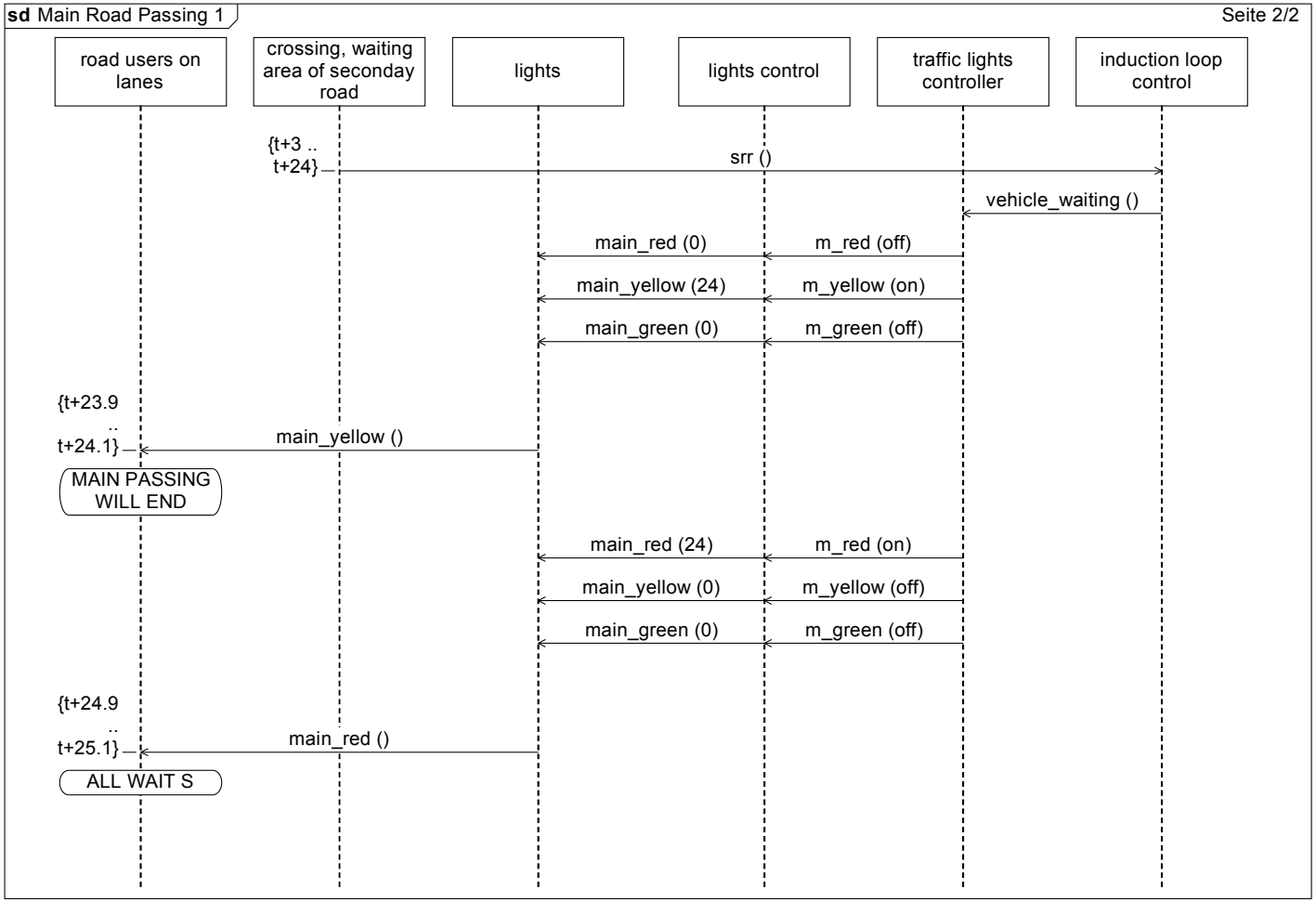
- The sequence diagrams together must describe the same behavior as in Phase 4.
- The signals at the external interfaces of this phase must be the same, have the same direction and the same order as in Phase 4.
- All signals in the interface classes specified in Phase 5 must be used in at least one sequence diagram of one subproblem, and the direction of signals must be consistent with the required or provided interfaces.
- It must be possible to map the new states to the states in Phase 4.

Example 1: traffic light control

Interface behavior for TrafficLightsControl, Subproblem MainRoadPassing 1 I



Interface behavior for TrafficLightsControl, Subproblem MainRoadPassing 1 II



Interface behavior for TrafficLightsControl, Subproblem MainRoadPassing 1 III

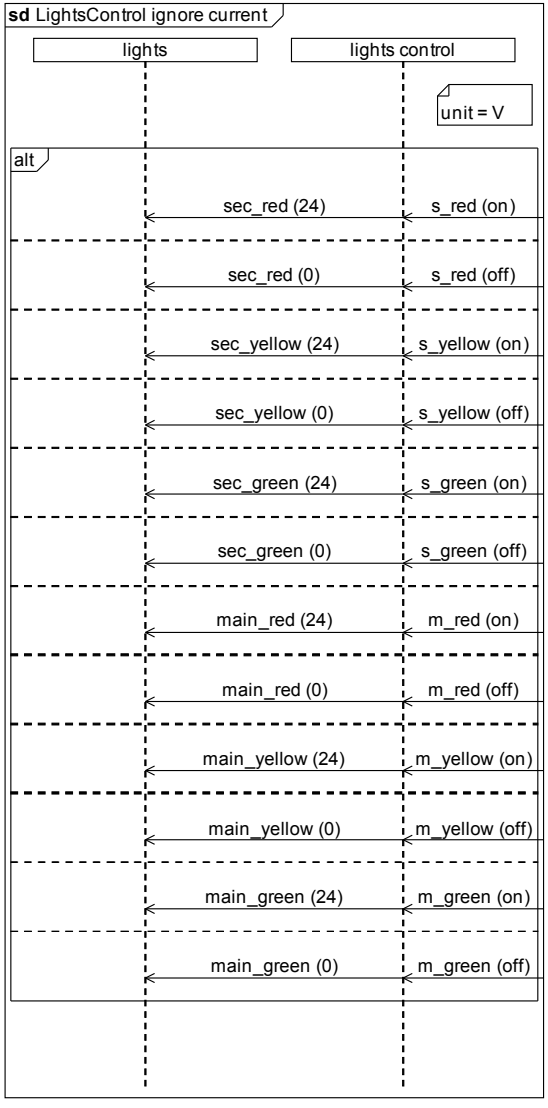
We suggest to use the alternative way of specification:

- In this phase, it is also possible to merge the domains *lights*, *lights control* and *TLC main phase* and express the interaction between these components separately.
- Since the diagrams become very complex in this case and only little additional information is given in the diagrams above, the specification of Step 4 can be re-used, and the behavior of the components *LightsControl* and *InductionLoopControl* can be specified separately on the next slides.

Interface behavior for LightsControl I

All subproblems

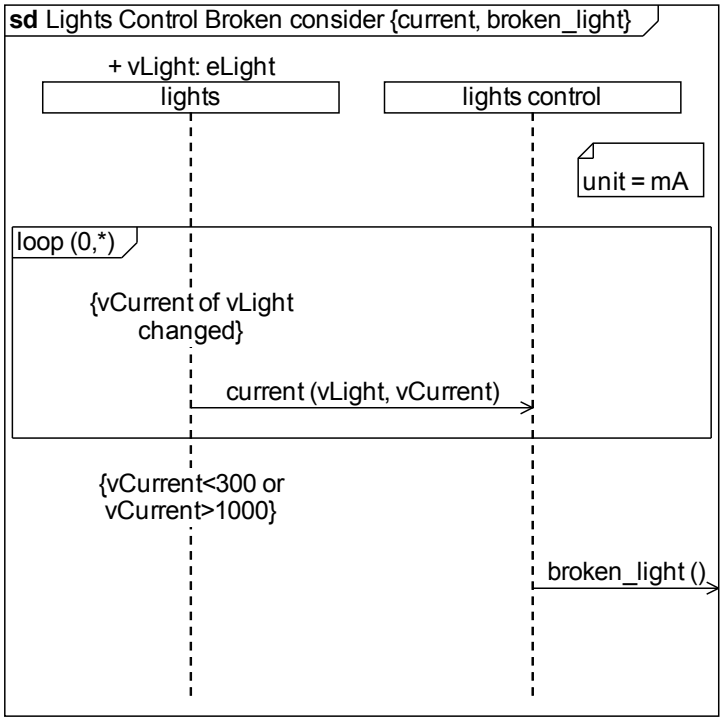
The component *lights control* converts the digital signals *(on/off)* into an analog voltage to control the lights.



Interface behavior for LightsControl II

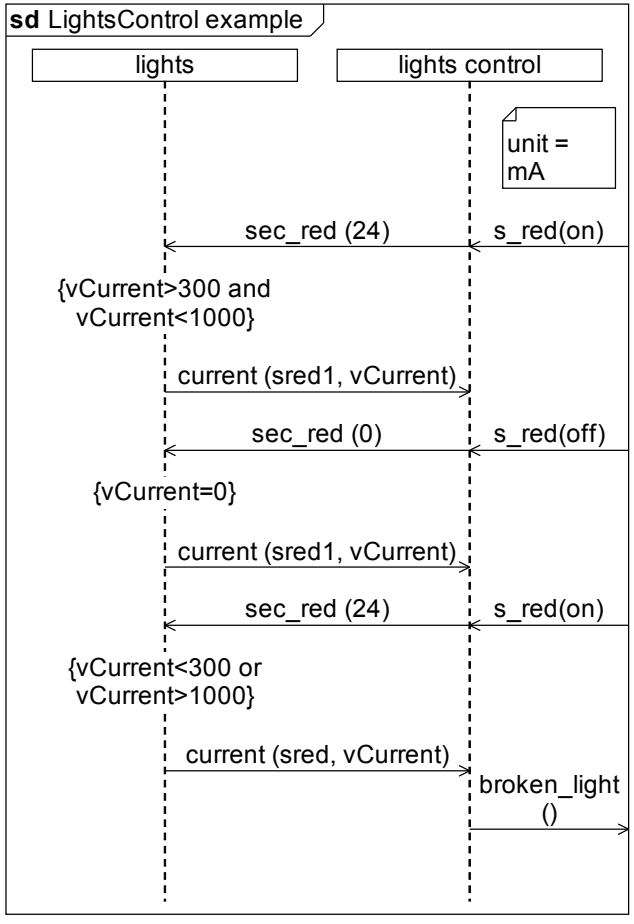
Subproblem BrokenLight-SafeState

When a light bulb is supplied with 24 V, a functioning lights bulb uses a current between 300 mA and 1000 mA. If another current can be measured for one light bulb, the *BrokenLight* signal is generated.



Interface behavior for LightsControl III

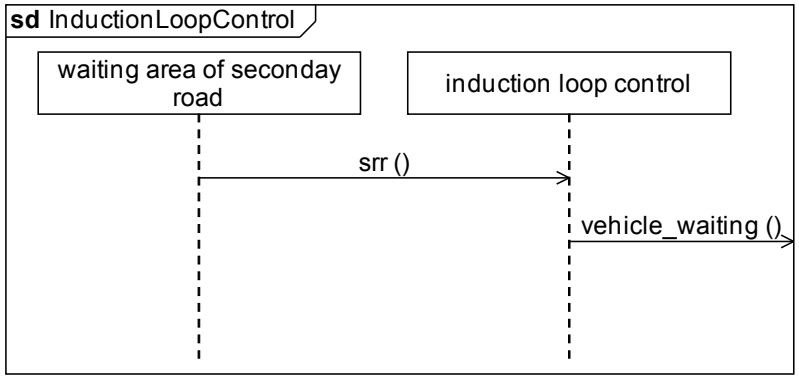
Subproblem
BrokenLightSafeState
(sample sequence)



Interface behavior for InductionLoopControl I

Subproblem MainRoadPassing

The secondary road request (*srr()*) is transformed into the signal *vehicle_waiting*. Since the abstract signal *srr* is used, an additional technical description is necessary (but not provided here).



Validation

- The sequence diagrams together describe the same behavior as in Phase 4, since all diagrams are re-used.
- All signals in the interface classes of Phase 5 are used in at least one sequence diagram.
- The direction of signals is consistent with the required or provided interfaces of Phase 5.
- The signals connect components as connected in the system architecture of Phase 5.

Example 2: sun blind control

Interface behavior for SunBlindControl

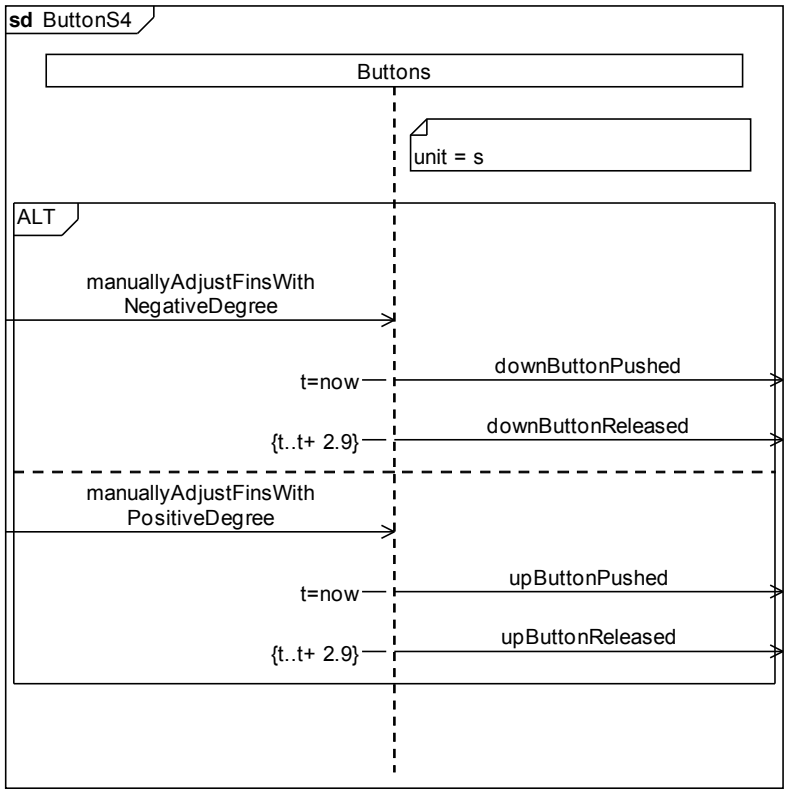
Subproblems SunControl, NoDestructControl, FinsControl, UserControl

The interface behavior can be directly derived from Phase 4 and the sequence diagrams of the other components.

Interface behavior for Button I

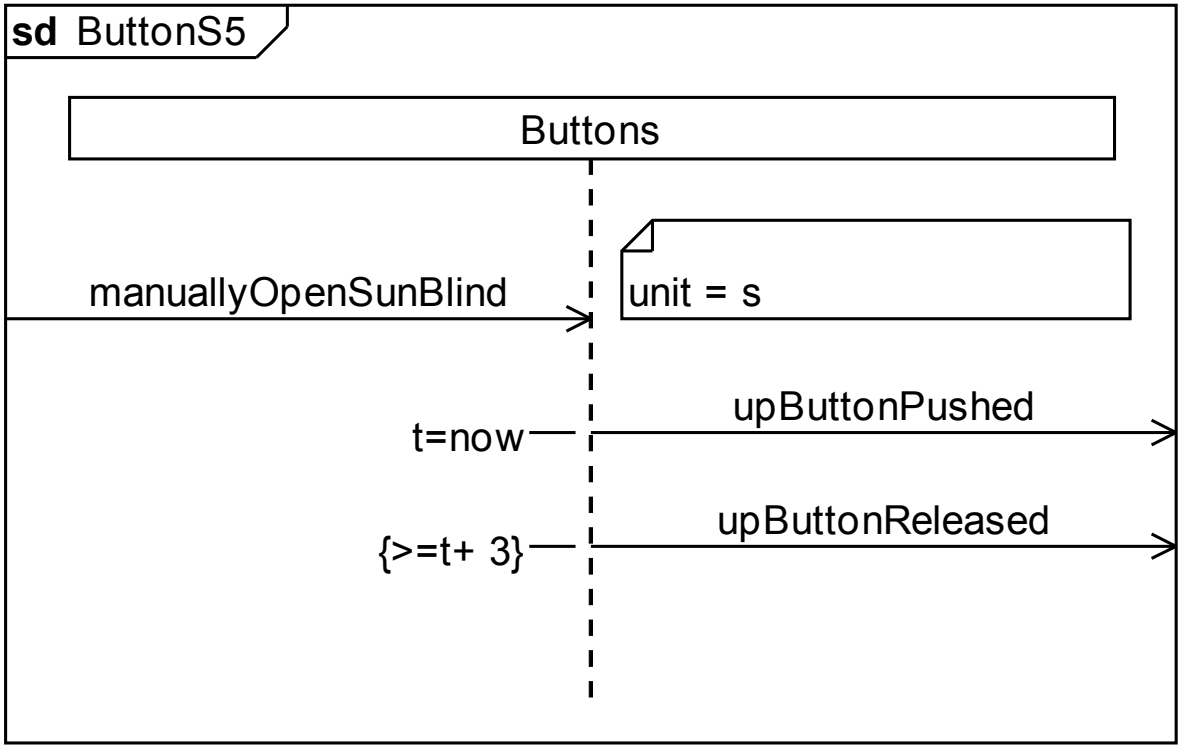
The signals to the *Buttons* are abstract and represents the intention of the user.

Subproblem FinsControl



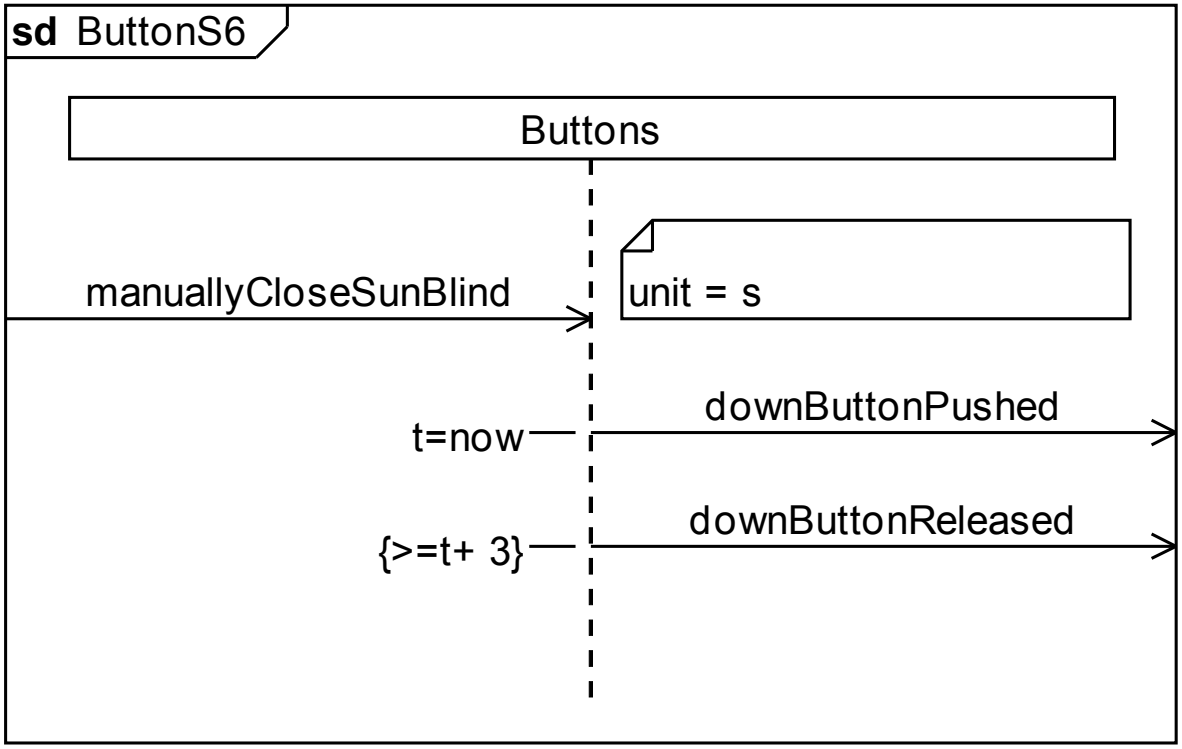
Interface behavior for Button II

Subproblem UserControl



Interface behavior for Button III

Subproblem UserControl



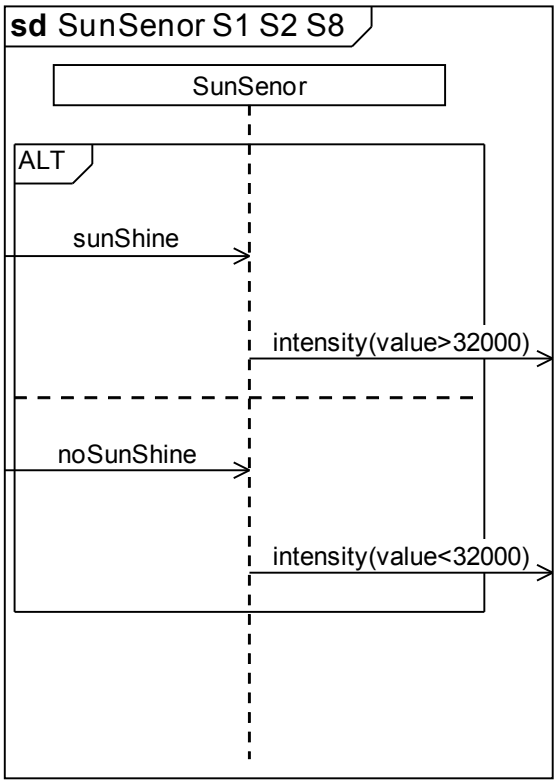
Interface behavior for Button IV

Subproblem SunControl

All sequence diagrams for *Button* together cover S8.

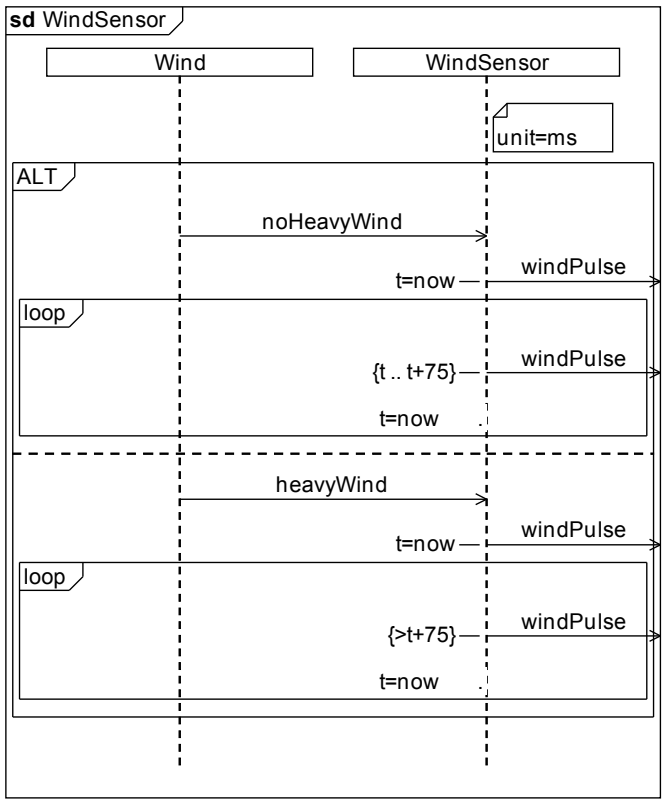
Interface behavior for SunSensor

Subproblem SunControl



Interface behavior for WindSensor

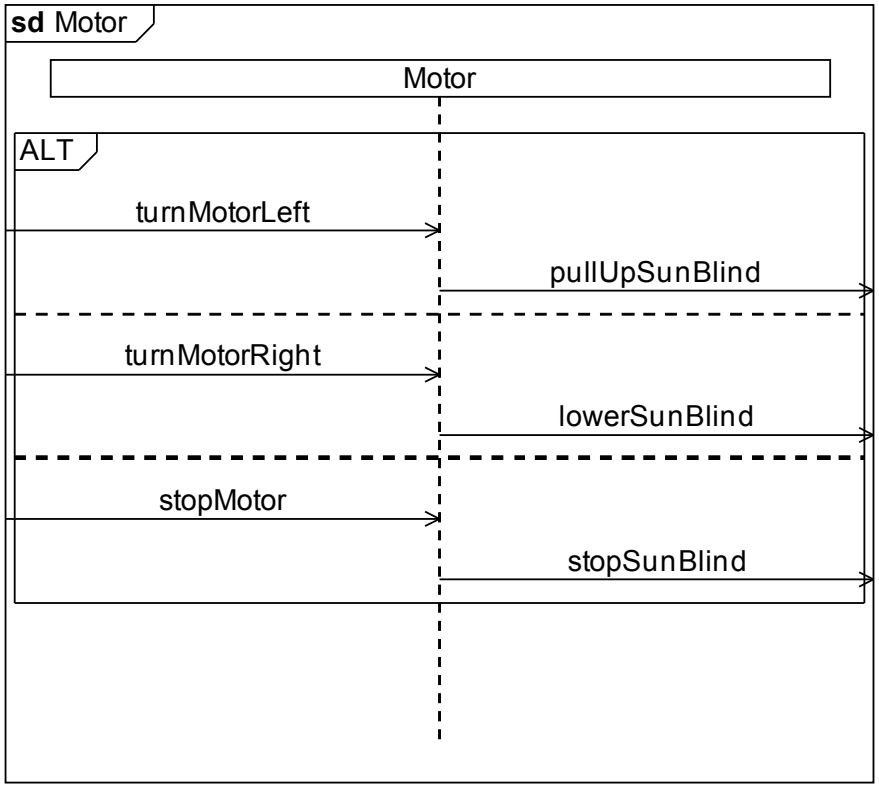
Subproblems SunControl, NoDestructControl, FinsControl, UserControl



The phenomena *WindSpeed* is an abstraction of a signal sequence consisting of *WindPulse*s.

Interface behavior for Motor

Subproblems SunControl, NoDestructControl, FinsControl, UserControl



Validation

- The sequence diagrams together describe the same behavior as in Phase 4, because all digrams are re-used.
- All signals in the interface classes of Phase 5 are captured in at least one sequence diagram. The phenomenon *WindSpeed* is an abstraction of a signal sequence consisting of *WindPulse*s.
- The direction of signals is consistent with the required or provided interfaces of Phase 5.
- The signals connect components as connected in the system architecture of Phase 5.
- No new state invariants are introduced.

Phase 7: Software architecture for all programmable components of the global system arch. I

...

5. Design global system architecture
6. Derive specifications for all components of the global system architecture
7. Design a software architecture for all components of the global system architecture that should be implemented in software
8. Specify the behavior of all components of all software architectures, using sequence diagrams
9. Specify the software components of all software architectures as state machines
10. Implement software components and test environment

...

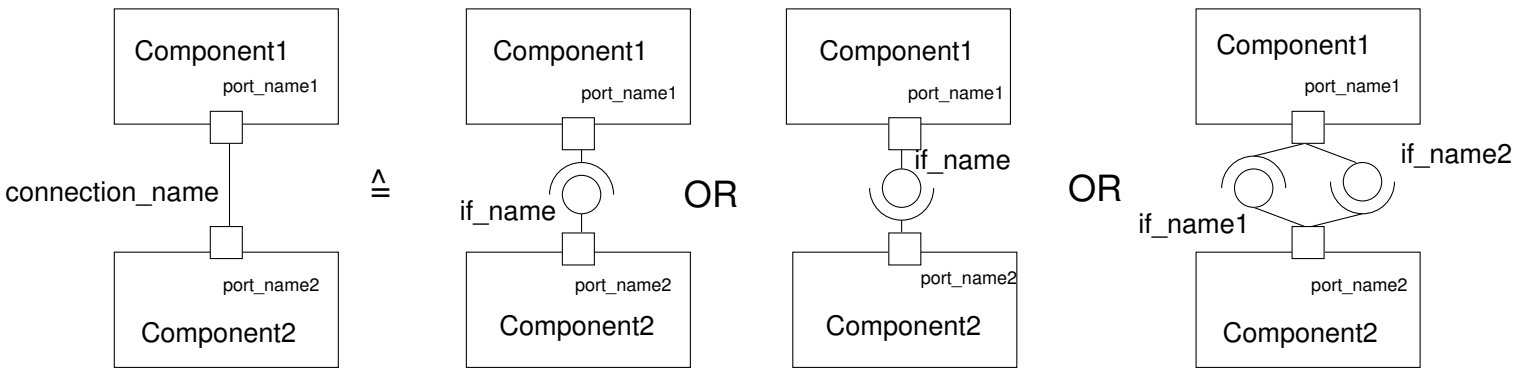
Phase 7: Software architecture for all programmable components of the global system arch.

input:	global system architecture from Phase 5	composite structure diagram
	problem diagrams from Phase 3	Jackson with do notation
	interface specifications from Phase 5	interfaces classes
	relationships between subproblems specified in Phase 5	grammars
	possibly reusable components from other projects (Phase 9)	active or passive class with interface classes
	machine behavior specifications from Phase 4	sequence diagrams wi annotated states
output:	layered software architecture for each subproblem	composite structure diagrams
	merged layered software architecture (with subcomponents)	composite structure diagrams
	purpose of each software component	natural language
	specification of interfaces between software components	interface classes
validation:	if no instantiation of architectural patterns: consistent with problem diagram	
	signals of Phase 4 sequence diagrams are interfaces of the application layer	
	direction of all signals consistent to each other and input	
	external interfaces must be consistent with the interfaces of the system architecture developed in Phase 5	

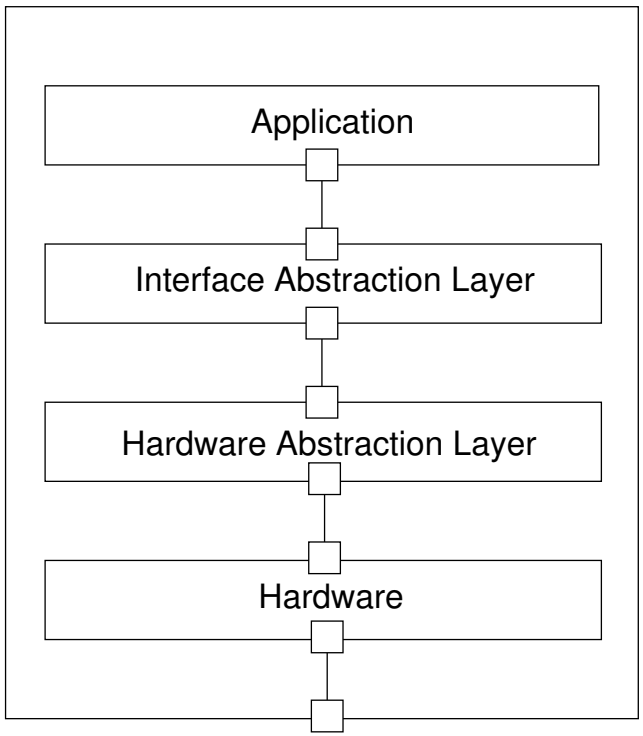
Notations and concepts

- Notation for connections and interfaces
- Four-variable model
- Architectural patterns

Notation for connections and interfaces



Layered architectures



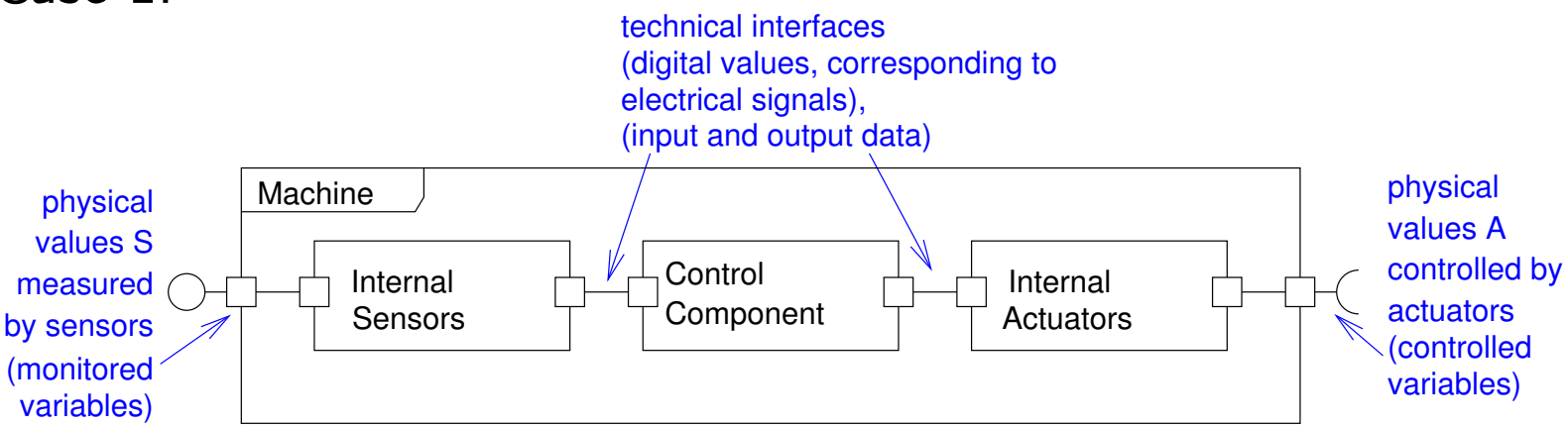
- Hierarchical organization of software plus hardware executing the software
- "Lower" layers provide services for "higher" layers
- Usually, only adjacent layers should be connected (no *layer bridging*)
- Advantage: modifications only affect adjacent layers
- Well-known example: ISO/OSI reference model for communication protocols

Four-variable model - basic idea

- Divide software into device-dependent and device-independent parts.
- Use (extended) four-variable model
  - Developed by David Parnas, extension by Connie Heitmeyer
  - Four Variables:
    1. Monitored variables: measured quantities (i.e., physical values, measured by sensors)
    2. Controlled variables: affected quantities (i.e., physical values, controlled by actuators)
    3. Input data: resources from which the values of monitored variables must be determined; submitted via a technical interface (electric signals corresponding to digital values)
    4. Output data: resources available to affect controlled variables; submitted via a technical interface (digital values corresponding to electric signals)

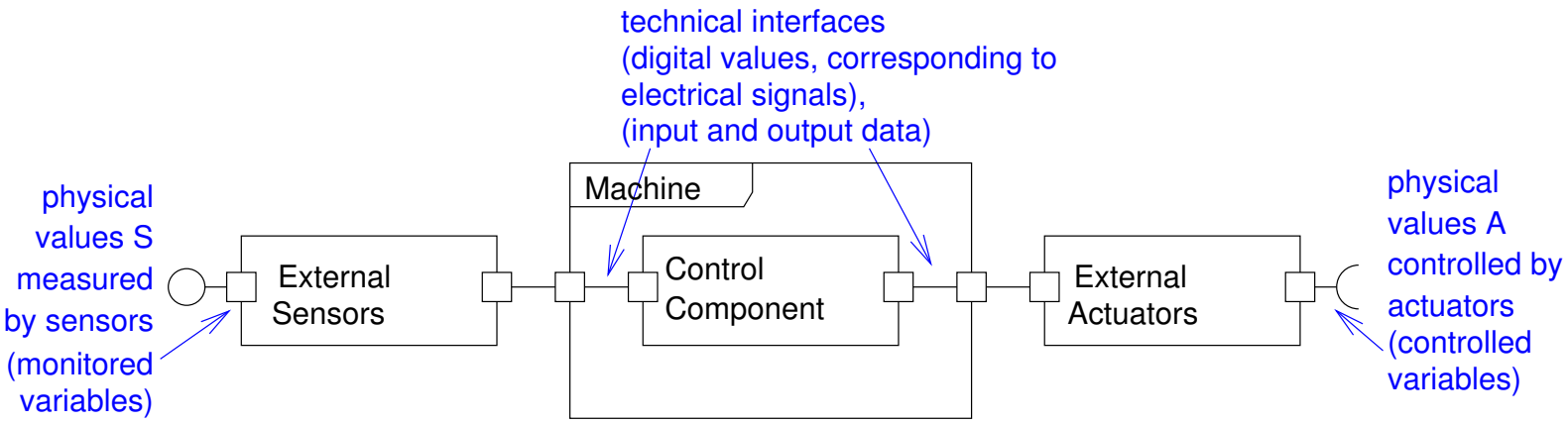
Four variable model: System architecture, I

Case 1:

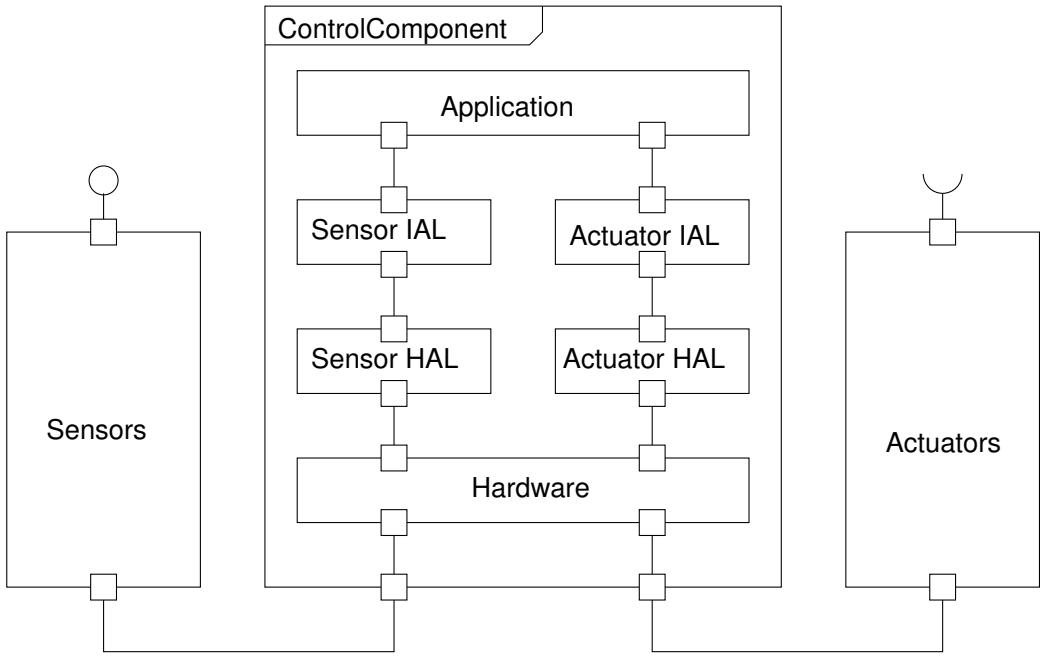


Four variable model: System architecture, II

Case 2:



Layered system architecture



Hardware Programmable hardware component

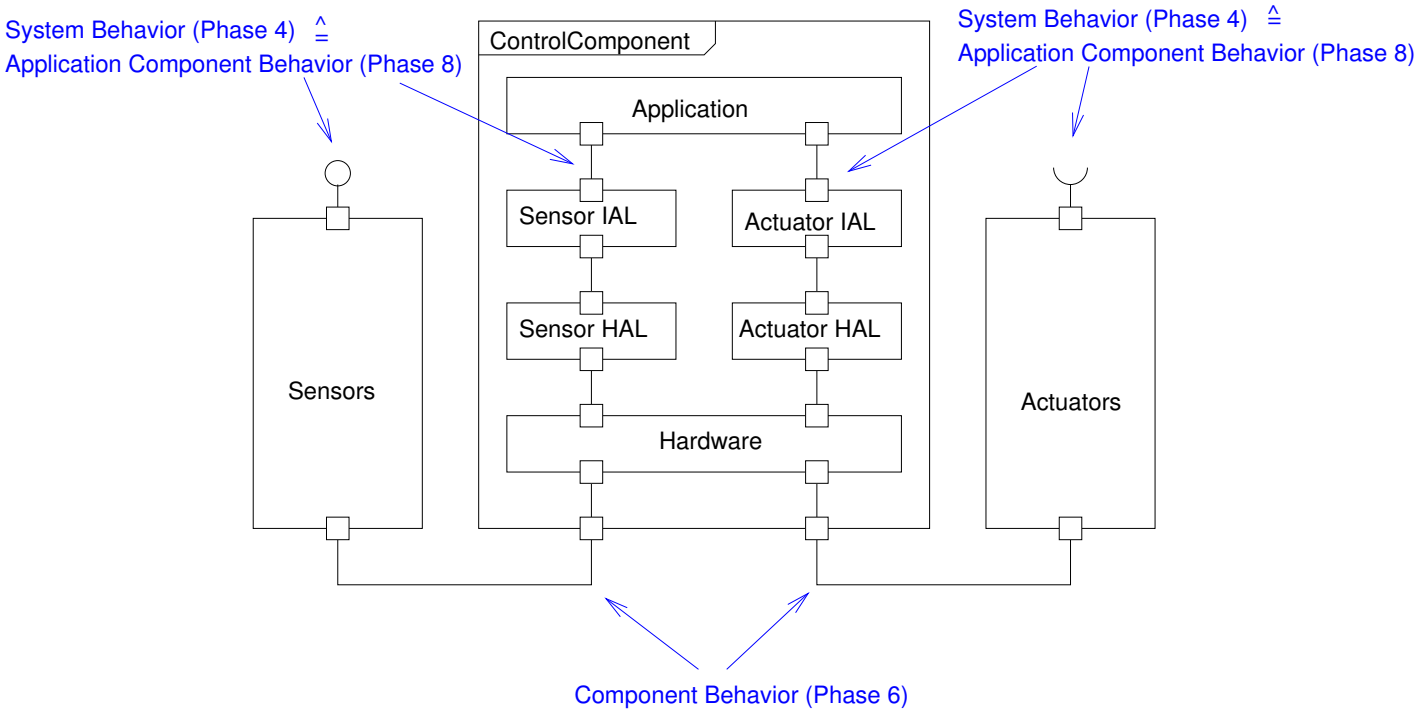
HAL Hardware Abstraction Layer: consists of drivers for external components; needed for portability

IAL Interface Abstraction Layer: provides input data or accepts output data, respectively

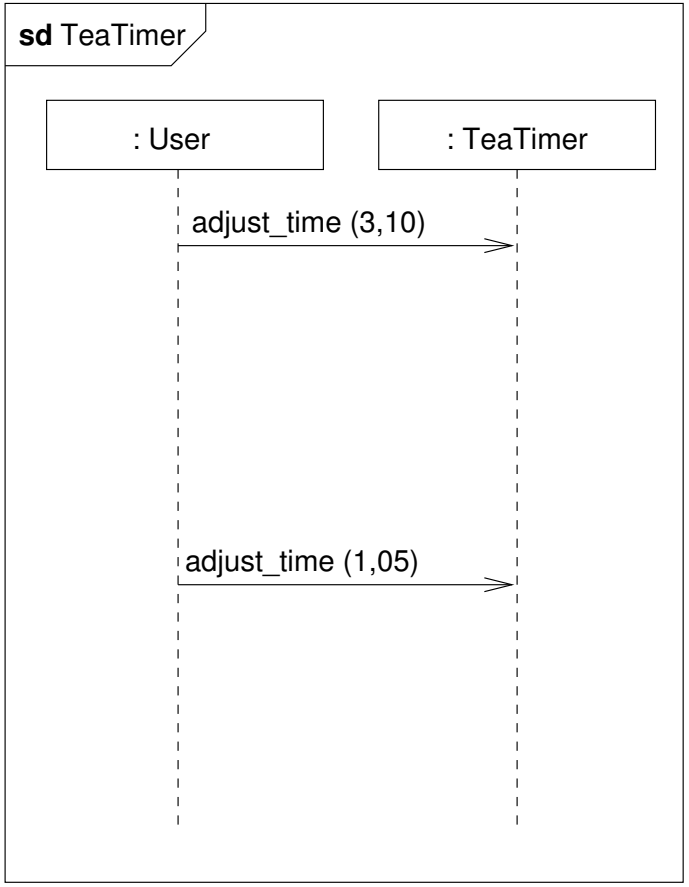
Application Layer: computes output data from input data

Extended four-variable model: Interfaces

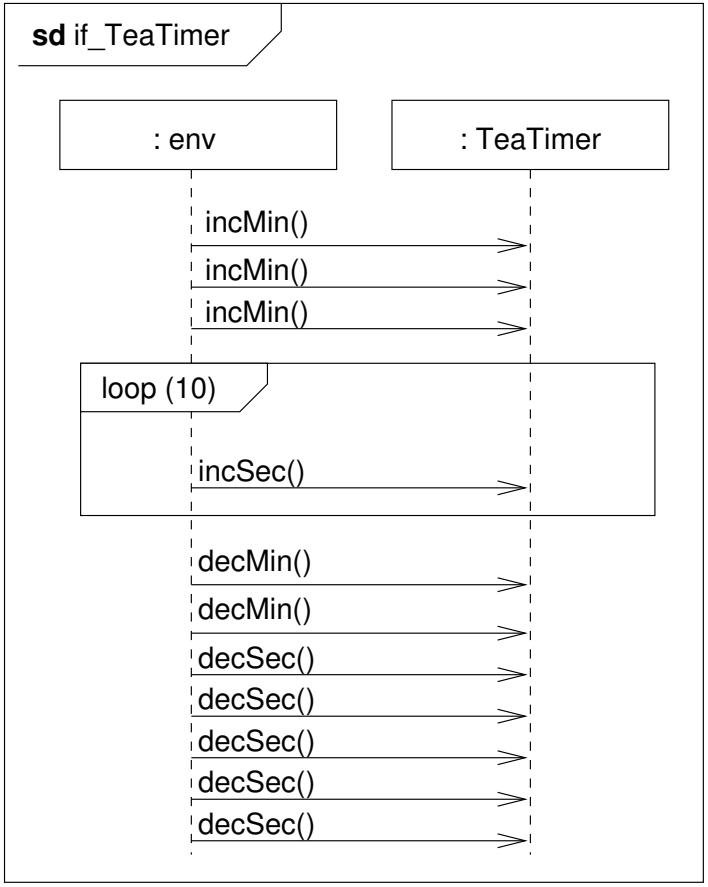
- Basic idea: application layer software should have the the same interfaces as the system, i.e., monitored and controlled variables
- Thus, application layer becomes device-independent, device dependencies are factored out in IALs and HALs.



Four-variable model: System vs. Component behavior I

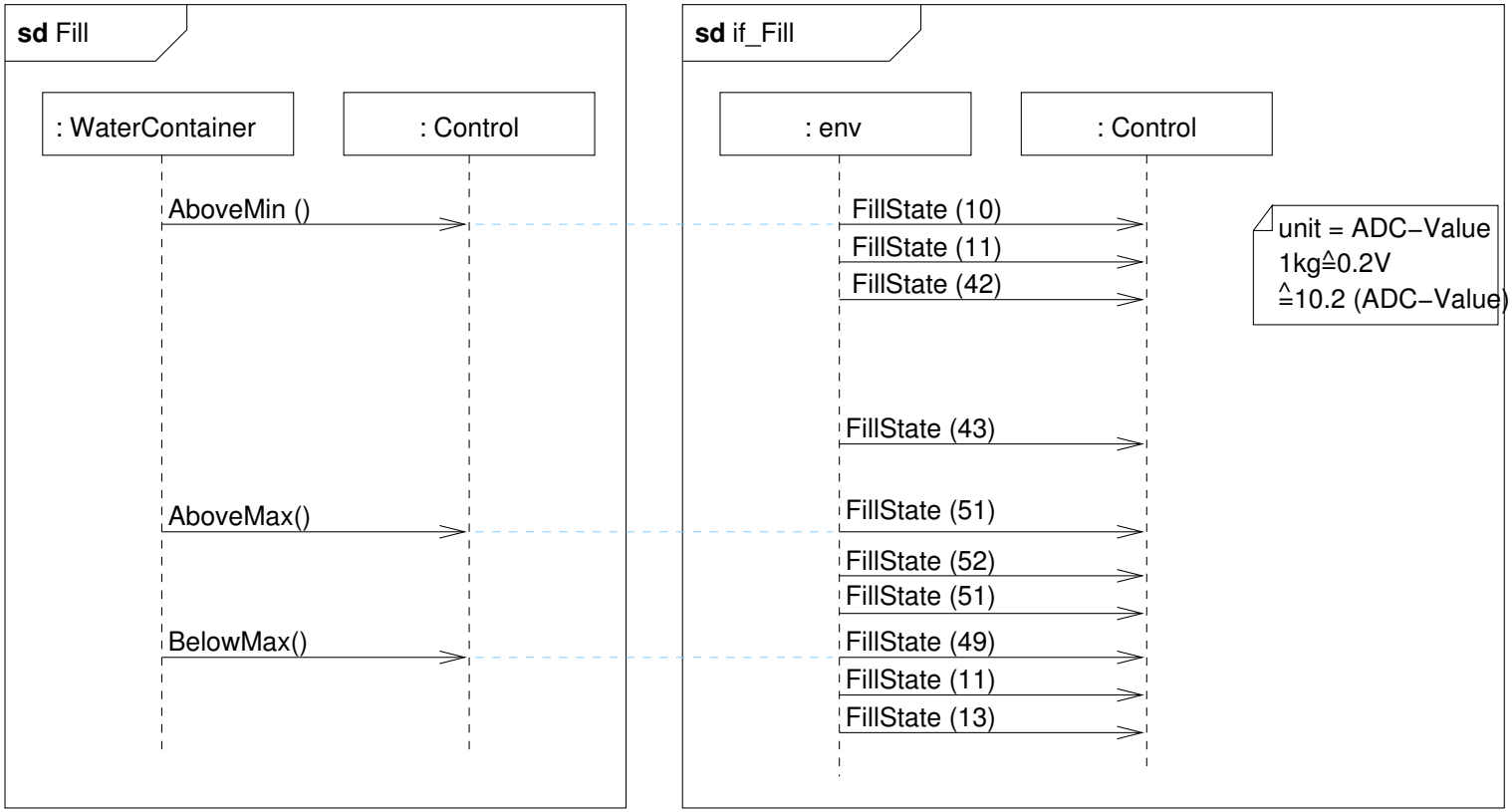


System behavior (Phase 4)



Component behavior (Phase 6)

Four-variable model: System vs. Component behavior II



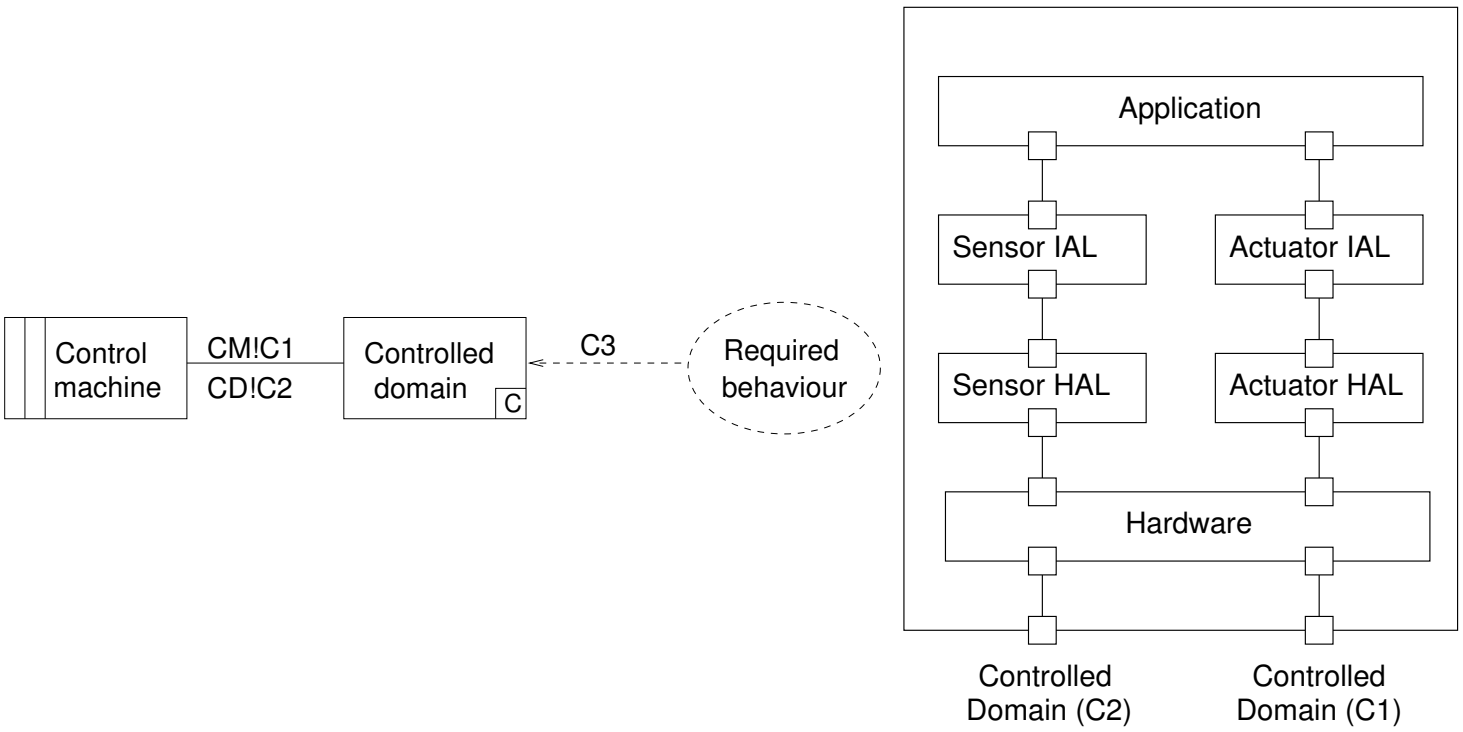
System behavior (Phase 4) Component behavior (Phase 6)

Analog-digital converter (ADV) to transform measured weight of vessel.
(5 V $\widehat{=}$ 255 $\widehat{=}$ 25 kg)

Architectural patterns

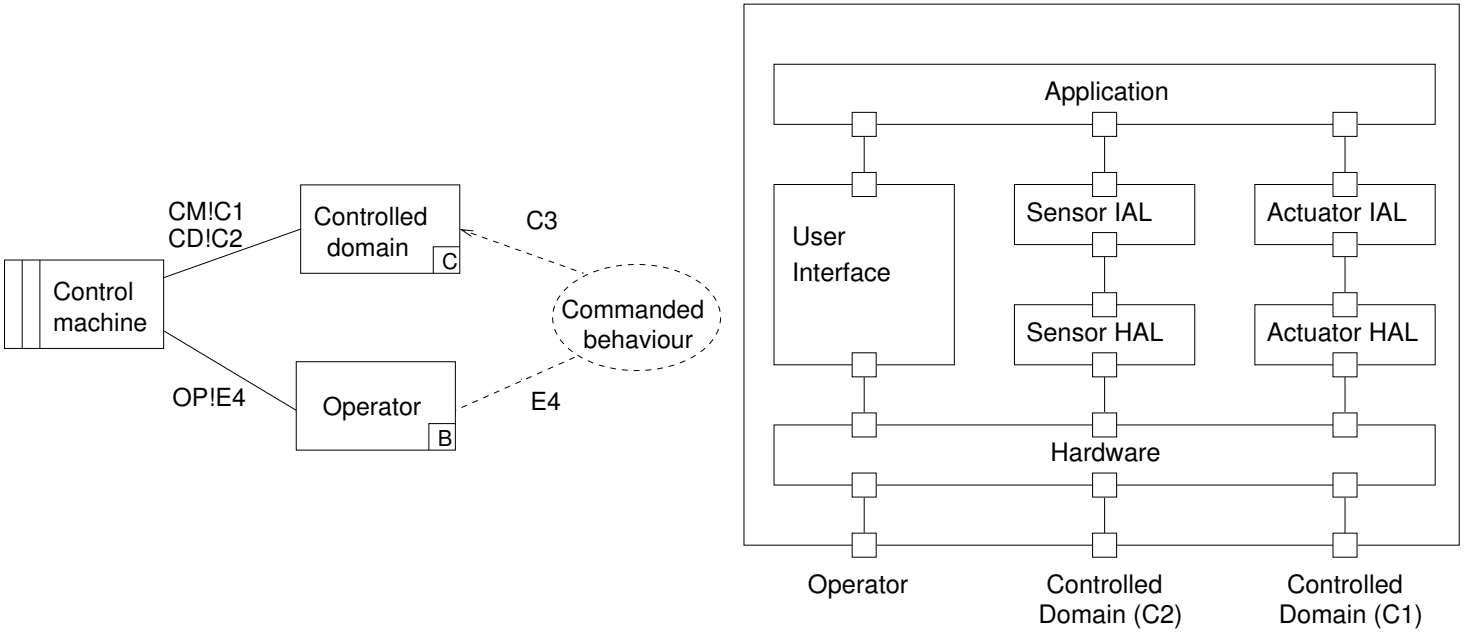
- For the most important problem frames the corresponding architectural patterns will be proposed.
- If a subproblem fits to a known problem frame, then a simple instantiation of the patterns will suffice.
- This architectural pattern is one possible solution and can be used as a starting point for further development.

Required behaviour frame diagram and architectural pattern



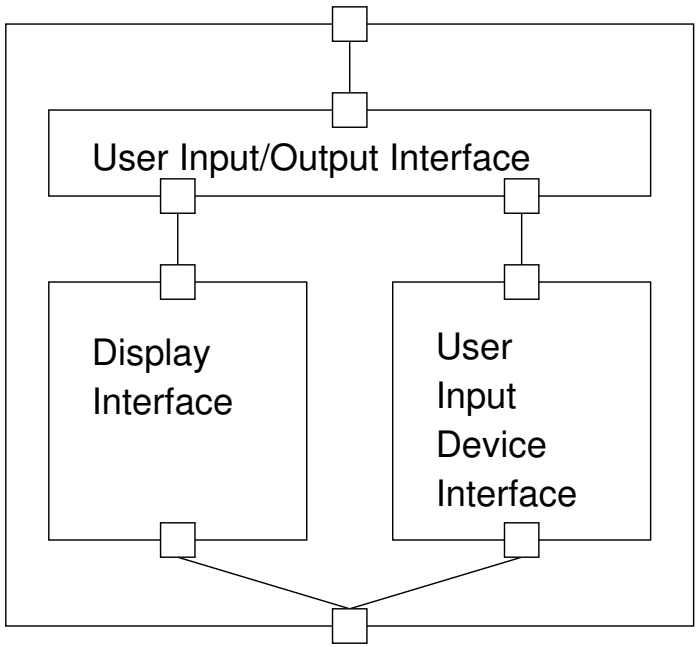
Fully automatic control system, no operator.

Commanded behaviour frame diagram and architectural pattern



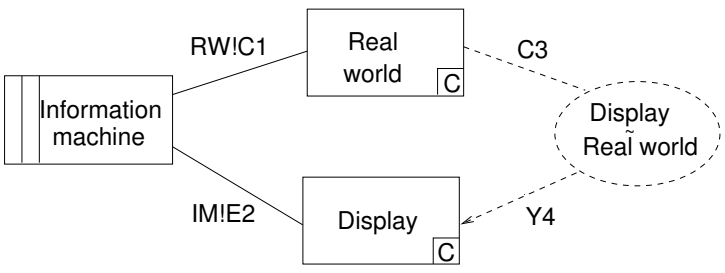
Control system with operator.

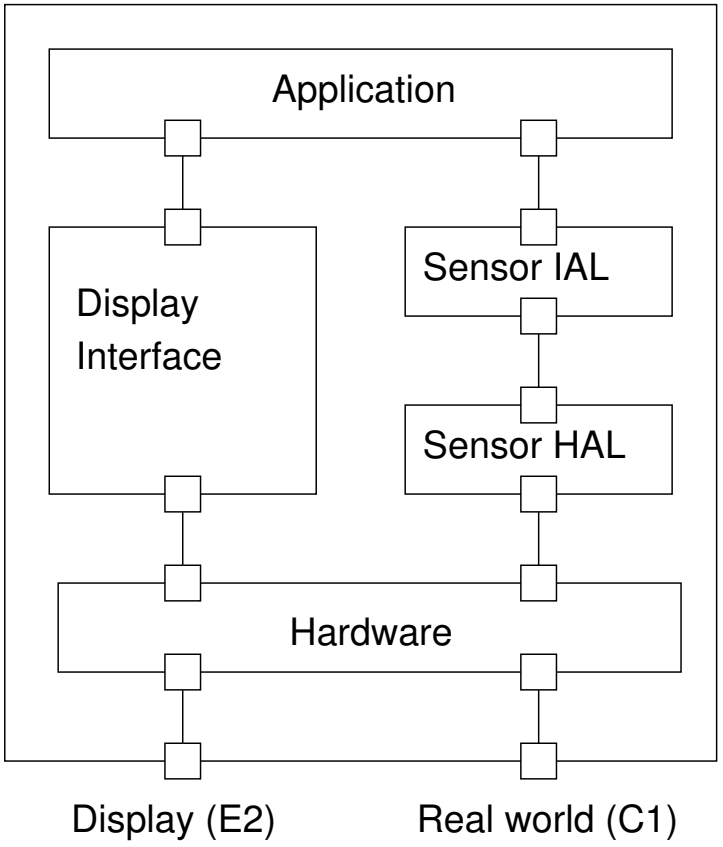
Detailed architectural pattern for user interface



Note: The architectual pattern contains a display to give feedback to the user (in contrast to the problem frame)

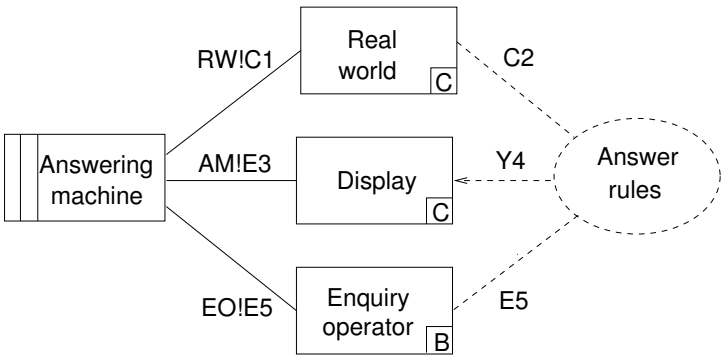
Information display frame diagram and architectural pattern

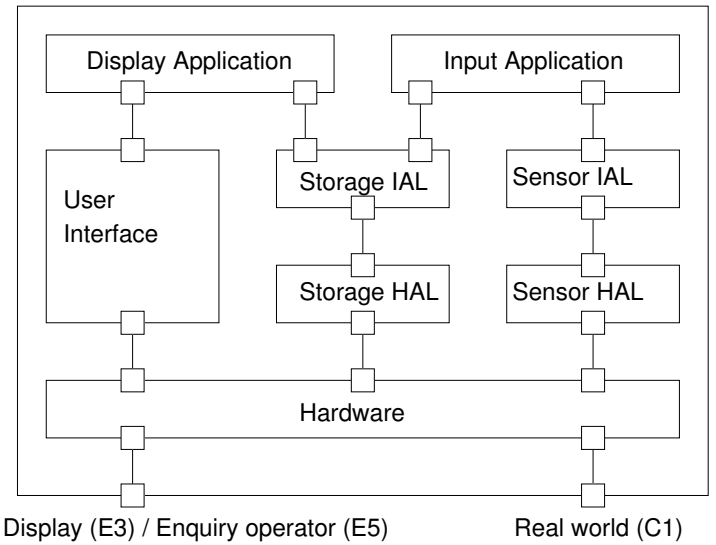




Display machine with application layer to process sensor values.

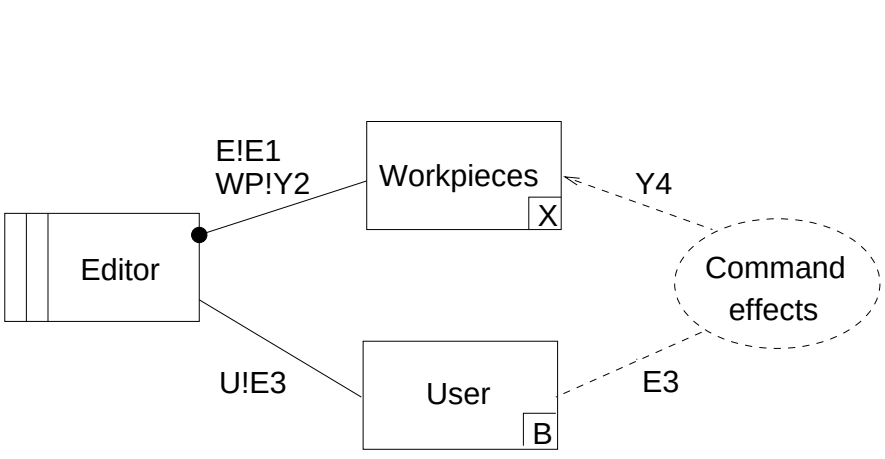
Commanded information frame diagram and architectural pattern

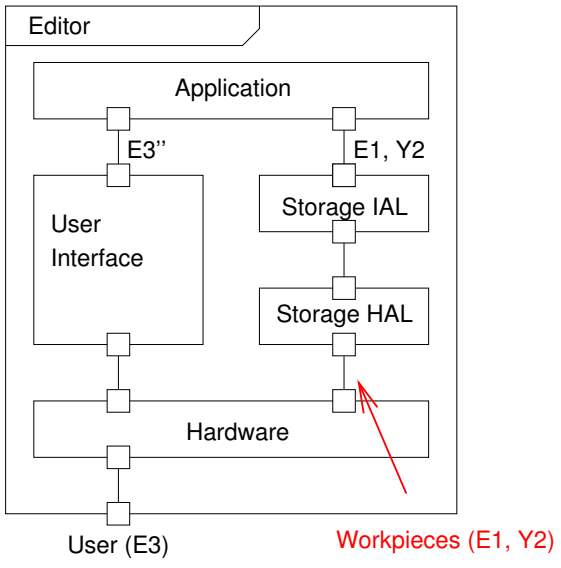




Display machine with operator. A data storage component serves to store information that can be queried by operator commands.

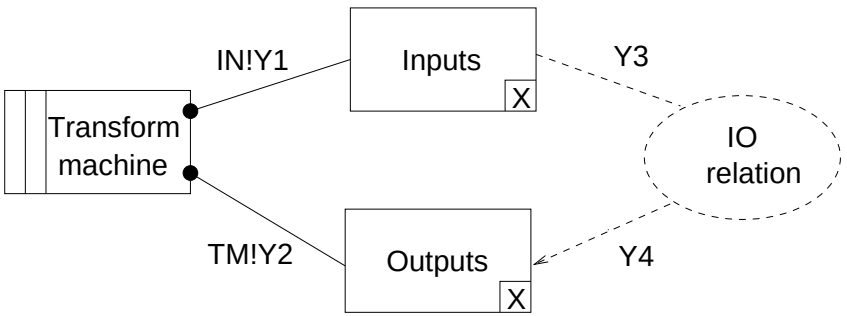
Workpieces frame diagram and architectural pattern

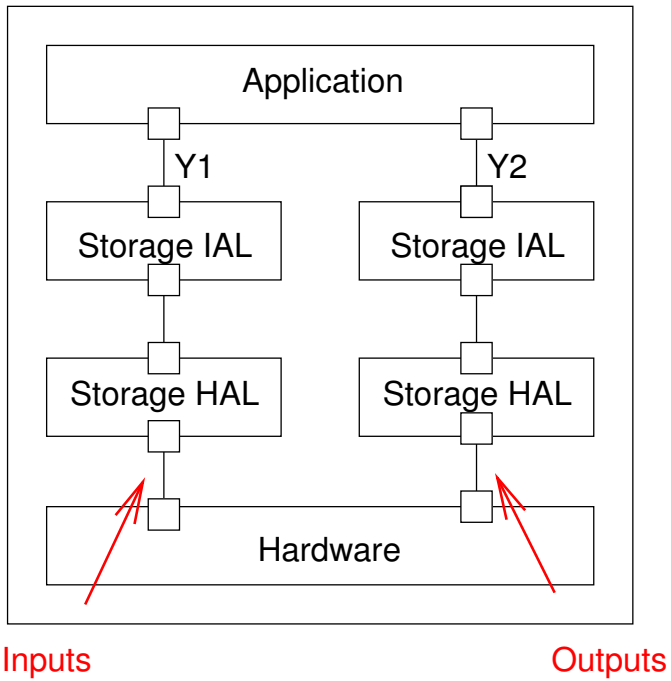




Note that there is only one interface with the environment.

Transformation frame diagram and architectural pattern





Phase 7: Software architecture for all programmable components of the global system arch.

input:	global system architecture from Phase 5	composite structure gram
	problem diagrams from Phase 3	Jackson with notation
	interface specifications from Phase 5	interfaces classes
	relationships between subproblems specified in Phase 5	grammars
	possibly reusable components from other projects (Phase 9)	active or passive c with interface classe
	machine behavior specifications from Phase 4	sequence diagrams annotated states
output:	layered software architecture for each subproblem	composite structure grams
	merged layered software architecture (with subcomponents)	composite structure grams
	purpose of each software component	natural language
	specification of interfaces between software components	interface classes
validation:	if no instantiation of architectural patterns: consistent with problem diagram	
	signals of Phase 4 sequence diagrams are interfaces of the application layer	
	direction of all signals consistent to each other and input	
	external interfaces must be consistent with the interfaces of the system architecture developed in Phase 5	

Executing Phase 7 I

For each programmable component and each subproblem, an architecture should be developed. If the component implements several subproblems, we develop a seperate architecture for each subproblem:

- If a subproblem fits to a known problem frame, then a simple instantiation of the corresponding pattern suffices.
- If the subproblem fits to a variant of some problem frame, the corresponding architectual pattern can be adjusted.
- If a subproblem is unrelated to any problem frame, then a corresponding architecture has to be developed from scratch. The following rules can be applied to develop a layered architecture:
  - The interfaces of the architecture correspond exactly to the interfaces of the machine domains as defined in the different problem diagrams.

Executing Phase 7 II

- If the machine has interfaces with causal domains, the corresponding architecture should contain components for handling sensors and actuators. This reflects the way in which software can communicate with and influence the physical world.
- If the frame diagram contains a biddable domain (i.e., an operator or user), then the corresponding architecture should contain a user interface component.
- If the machine has interfaces with lexical domains, these domains should be reflected as parts of the corresponding architecture, because lexical domains can only exist inside the machine.
- Components for data storage should only be included if the data is stored persistently. Otherwise they can be assumed to be part of some other component.

Executing Phase 7 III

Merge the architectures to a global architecture:

- Decide if two components contained in different subproblem architectures should occur only once in the global architecture.
- Make use of the information gathered when decomposing the overall problem into subproblems. Therefore, distinguish the following cases:
  1. The components are hardware (HAL) or interface abstraction layers (IAL), establishing the connection to some hardware device.
     Such components should be merged if and only if they are associated to the same hardware device.
  2. Two application components belong to subproblems being related sequentially or by alternative.
     Such components should be merged into one application component.

Executing Phase 7 IV

3. Two application components belong to parallel subproblems and share some output phenomena. Such components should be merged, because the output must be generated in a way satisfying both subproblems.
4. Two application components belong to parallel subproblems and share some input phenomena. If the components do not share any output phenomena, both alternatives (merging the components or keeping them separate) are possible. If the components are not merged, then the common input must be duplicated.
5. Two application components belong to parallel subproblems and do not share any interface phenomena. Such components should be kept separately.

- If a component is too complex, it should be split into subcomponents.

Executing Phase 7 V

- If timing constraints have been specified, include a timer component with corresponding time-out timer.

Executing Phase 7 VI

Specify interface classes:

- The external interfaces are the same as the interfaces in the system architecture between the components.
- Since we use interface classes to describe the hardware interfaces, the interfaces between IAL and HAL are similar to the external interfaces of the component. Thereby, the HAL contains no application-specific functionality. It only provides easy-to-use software interfaces to access the hardware (e.g. registers, interrupts, direct memory access).
- The interfaces to the application component can be derived from sequence diagrams that describe the machine behavior (Phase 4).

Executing Phase 7 VII

- If the interface of the application layer is the same as the interface of the HAL, the IAL can be removed from the architecture.
- As described in Phase 5, for each interface it must be decided, which component provides the interface and which component uses the interface. Usually, the component being in control of a phenomenon uses the corresponding interface. If an interface contains operations with return values, then the component providing these interfaces is in control of a phenomenon.

Remarks I

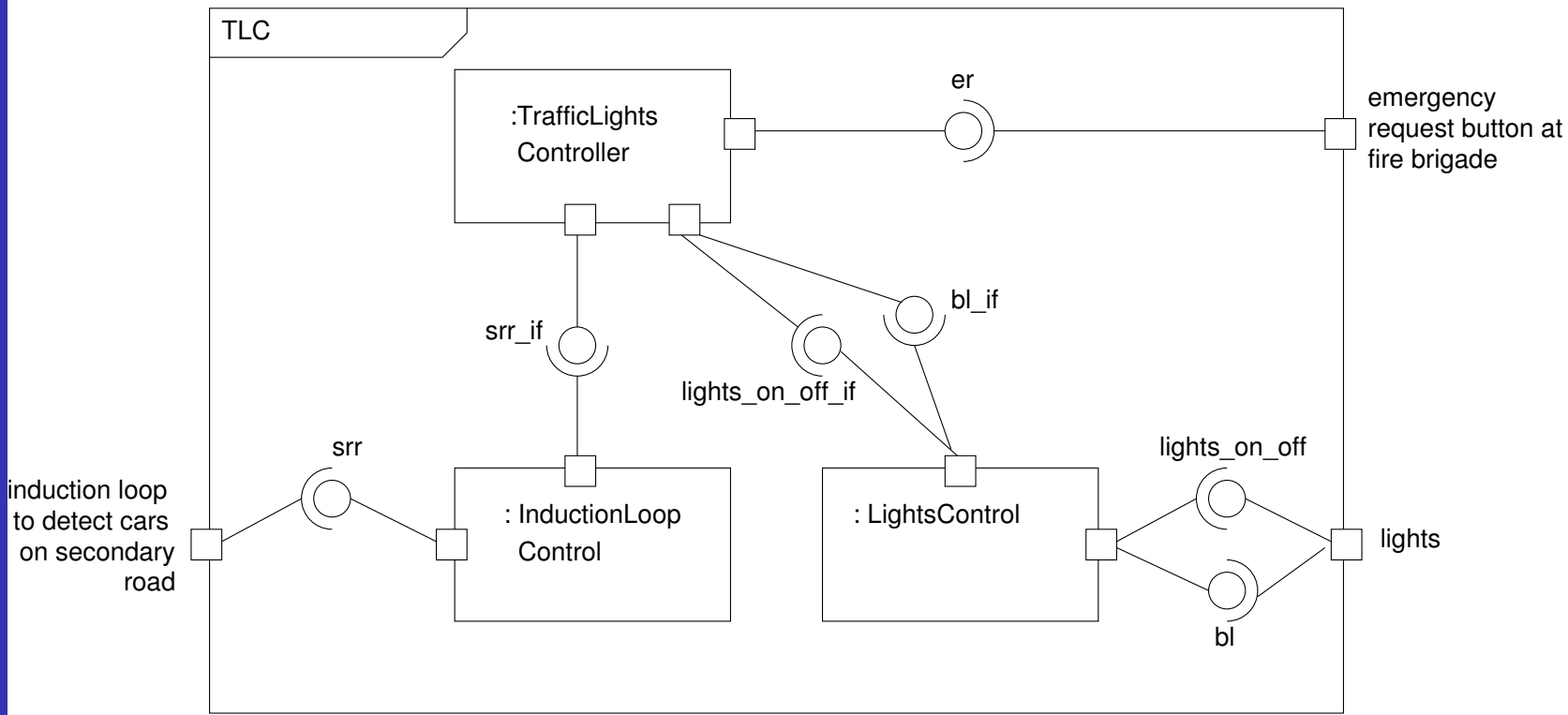
- If reusable components from other projects are used, they must be integrated into the software architecture.
- The already developed architectures of the other subproblems should be checked for reusable components.
- The global architecture must contain all components of all subproblem architectures. Its external interfaces must be the same as in the system architecture developed in Phase 5.
- The external interfaces of the software architecture are usually connected with a microcontroller component. The software components can access these interfaces using *ports* and *interrupts*.

Remarks II

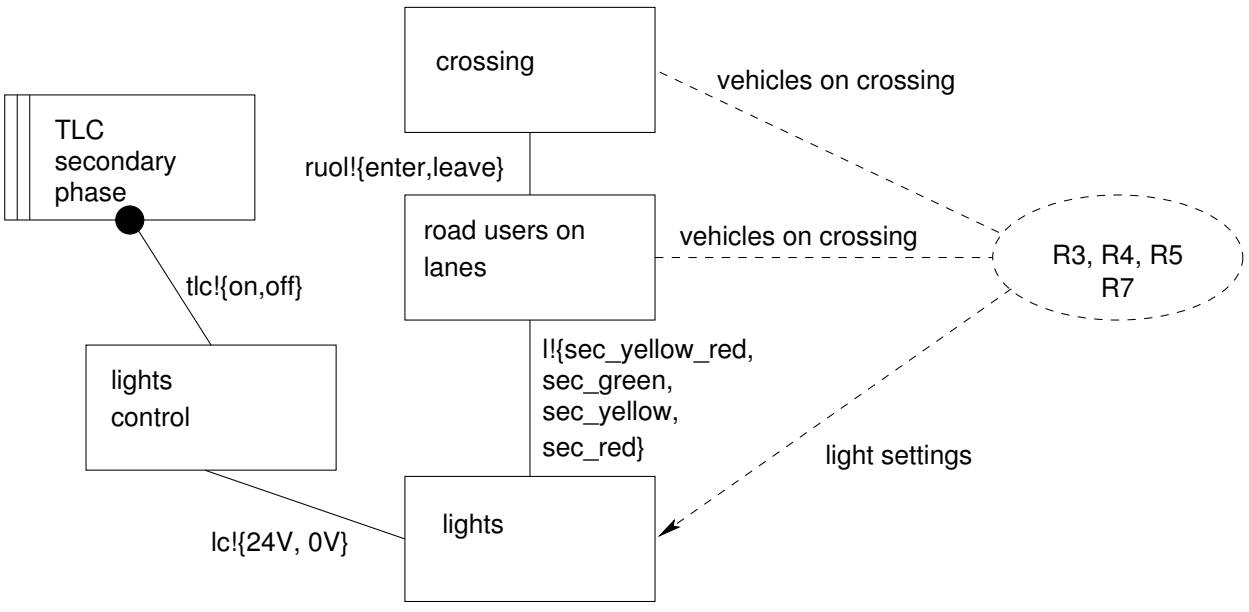
- *Ports* are provided by the microcontroller, and the software can use ports to read out input data or to send output signals.
- *Interrupts* are required interfaces of the microcontroller. The microcontroller sends the interrupts to the pre-configured software component when a change of the state at the interface is detected. Note: An interrupt cannot have parameters. The parameters must be read out using *ports*.
- Interrupts are assigned to fixed input pins of the microcontroller. Each microcontroller has a fixed number of pins that can send interrupt signals.

Example 1: traffic light control

TrafficLightsControl System Architecture

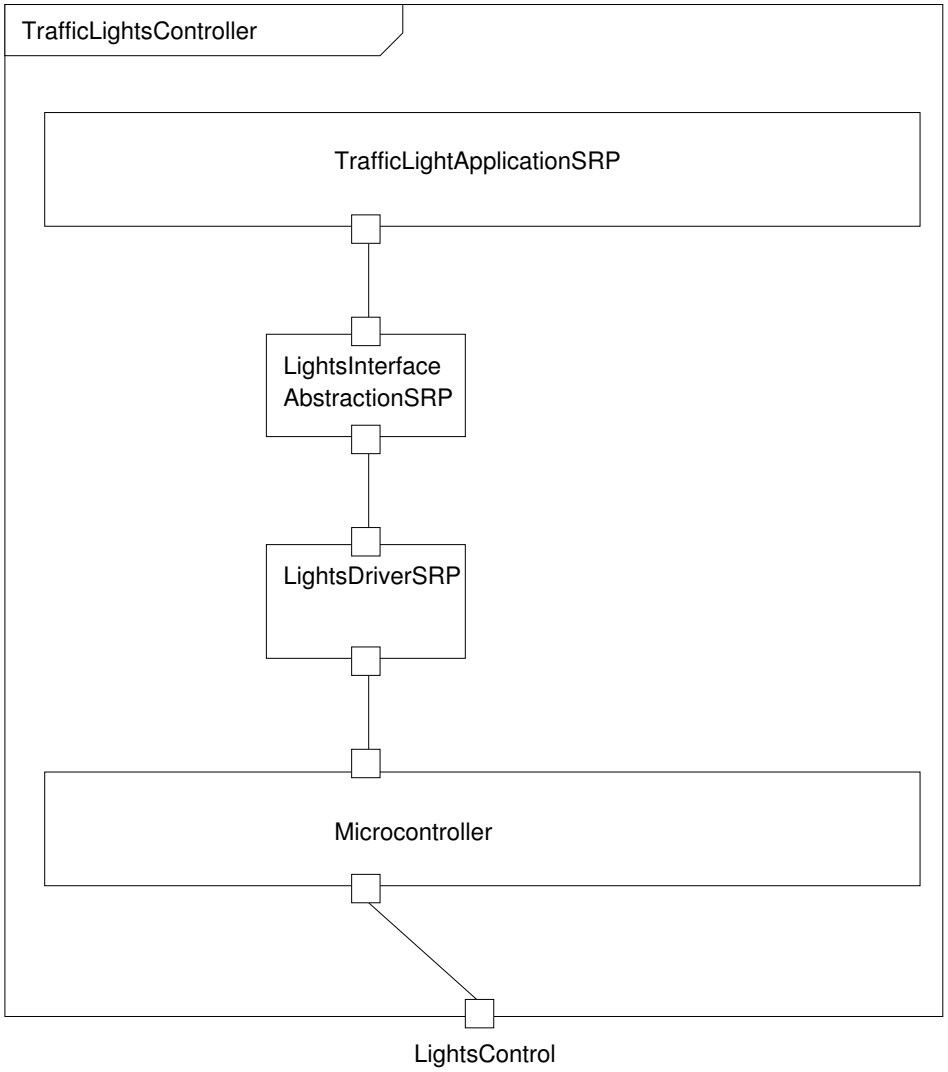


TrafficLightsControl SecondaryRoadPassing problem diagram



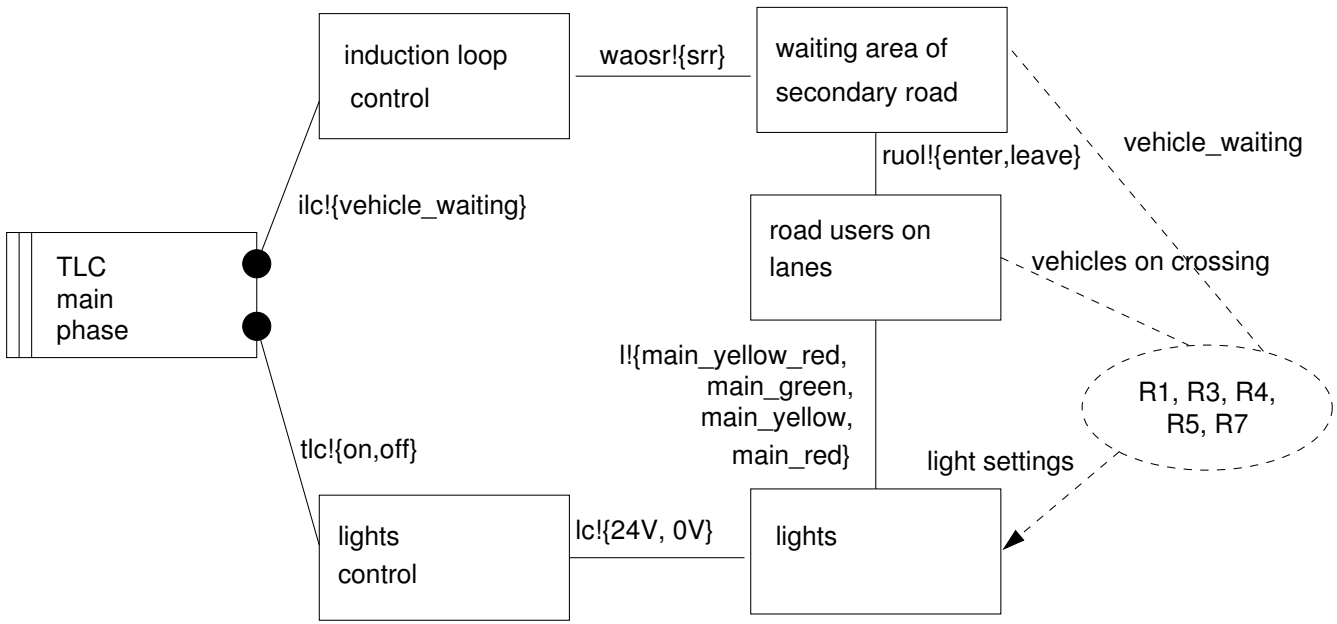
Variant of the required behavior.

TrafficLightsControl SecondaryRoadPassing architecture



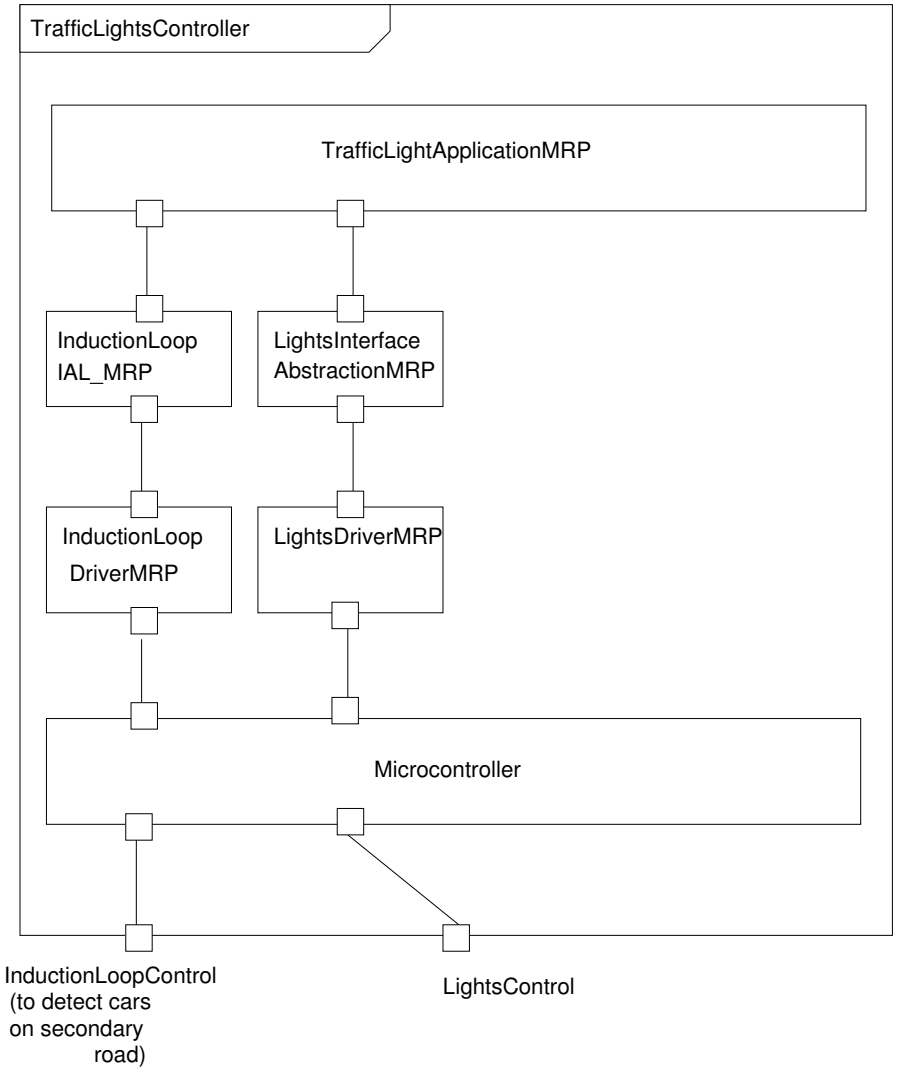
Variant of the required behavior architectural pattern.
In the problem diagram no sensor is contained. For this reason, the components *Sensor IAL* and *Sensor HAL* are removed from the software architecture.
The Microcontroller is a reused component.

TrafficLightsController MainRoadPassing problem diagram

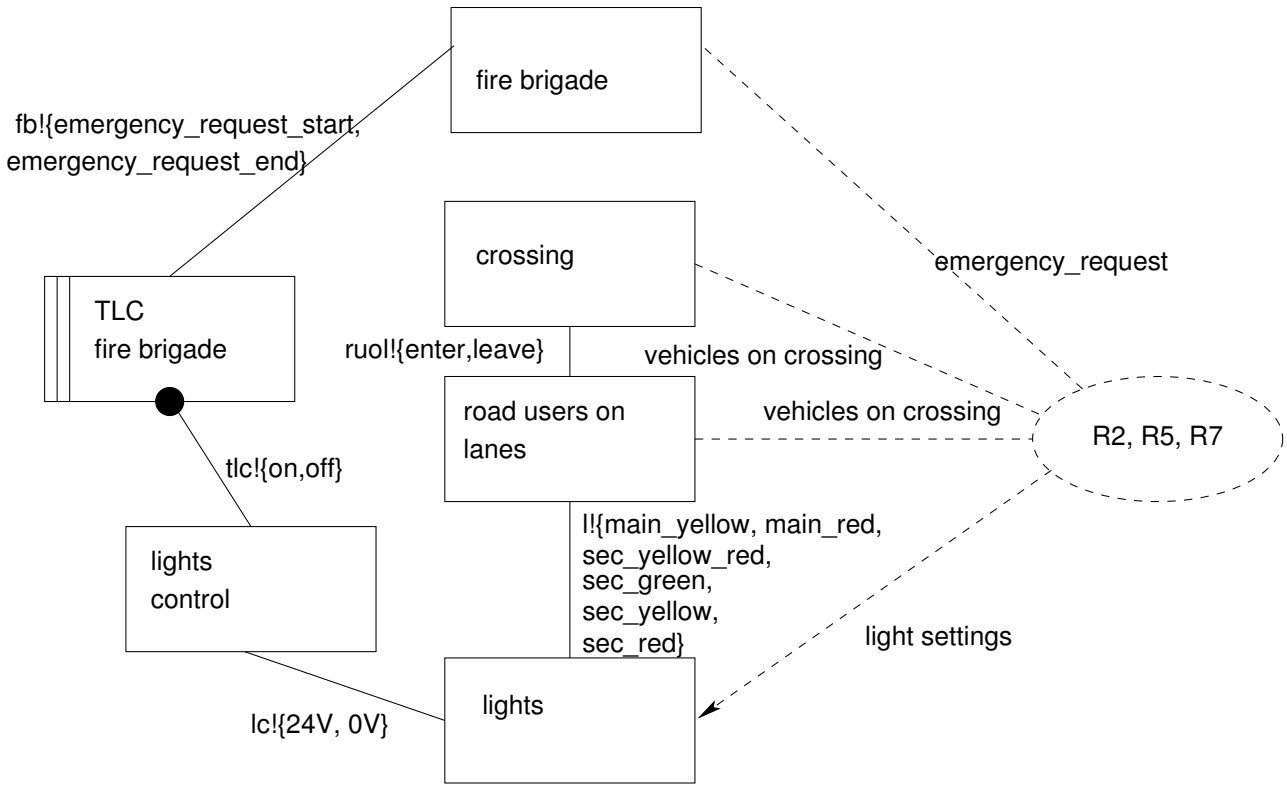


Variant of the required behavior.

TrafficLightsController MainRoadPassing architecture

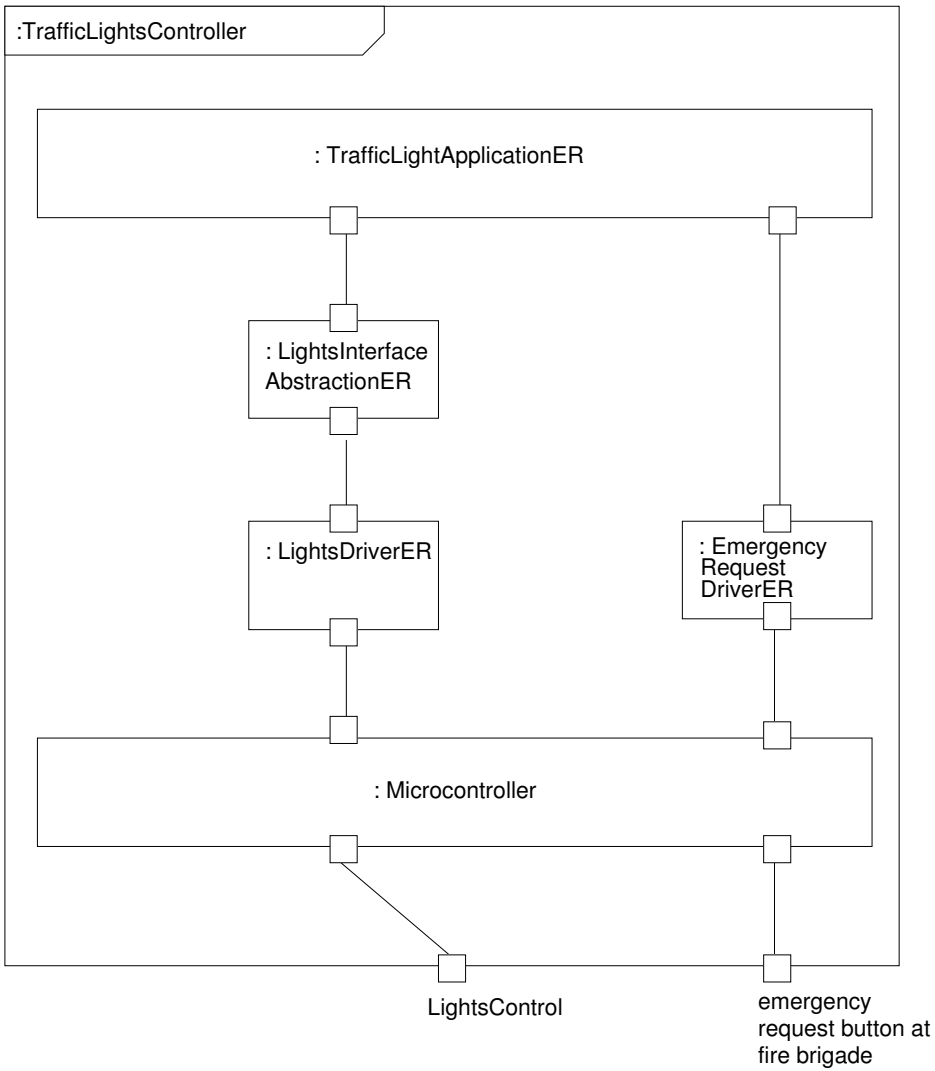


EmergencyRequestSecondaryRoadPassing Problem Diagram



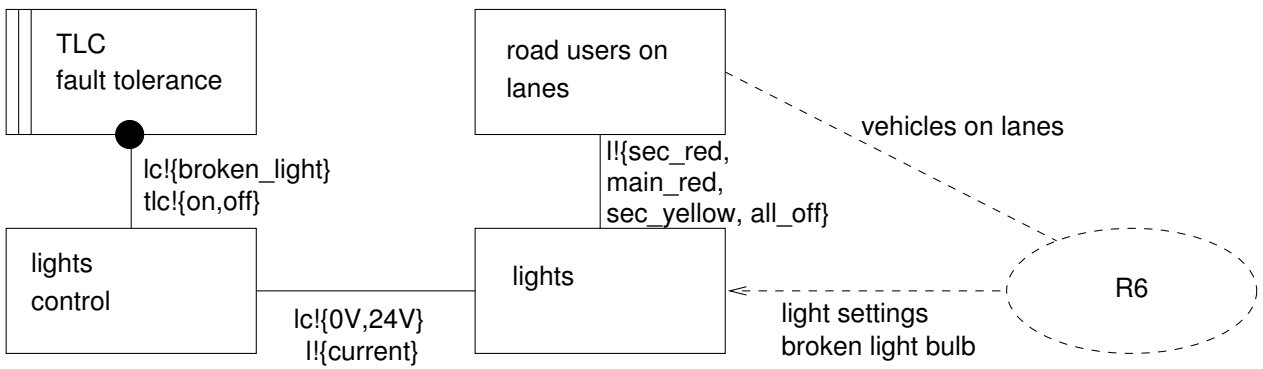
Variant of the commanded behavior.

EmergencyRequestSecondaryRoadPassing architecture



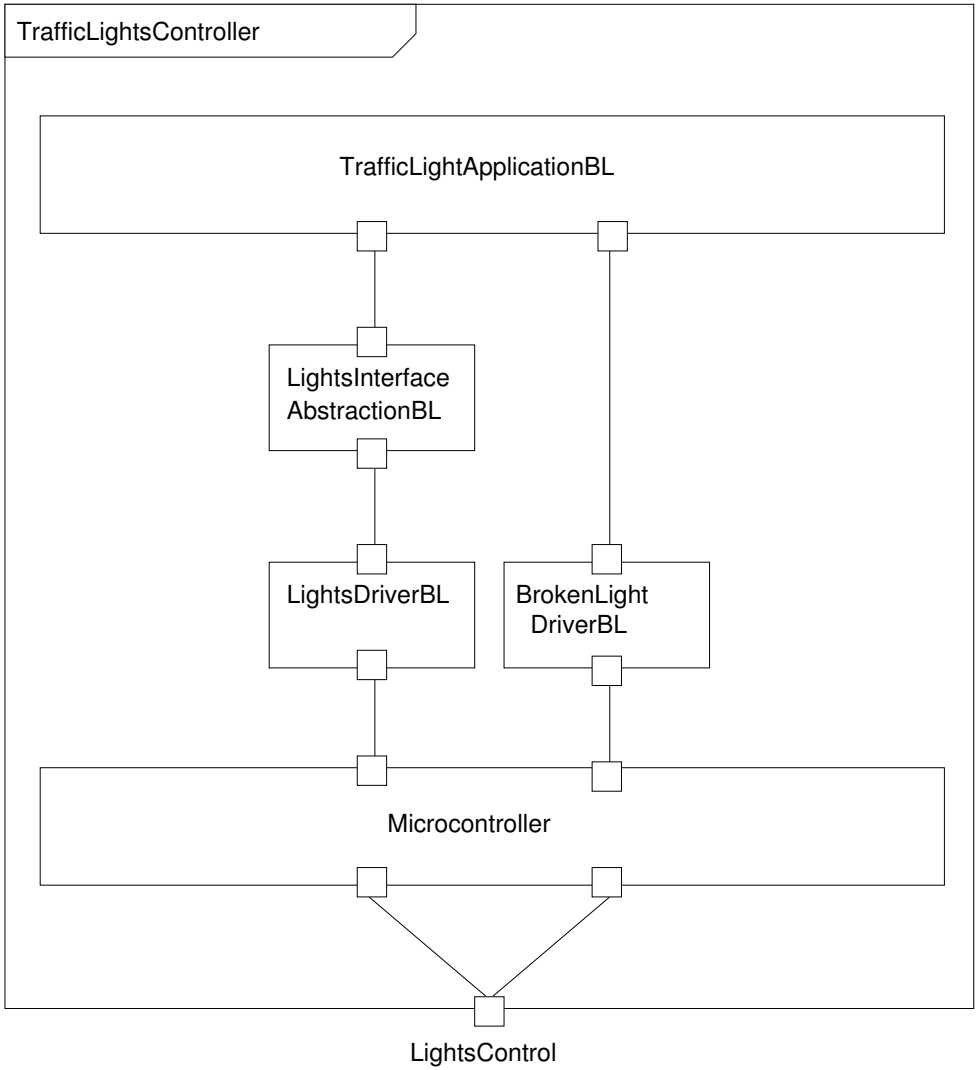
A sensor is not necessary and therefore removed. The user interface is just a button and no feedback is given to the user.

TrafficLightsController BrokenLightSafeState problem diagram

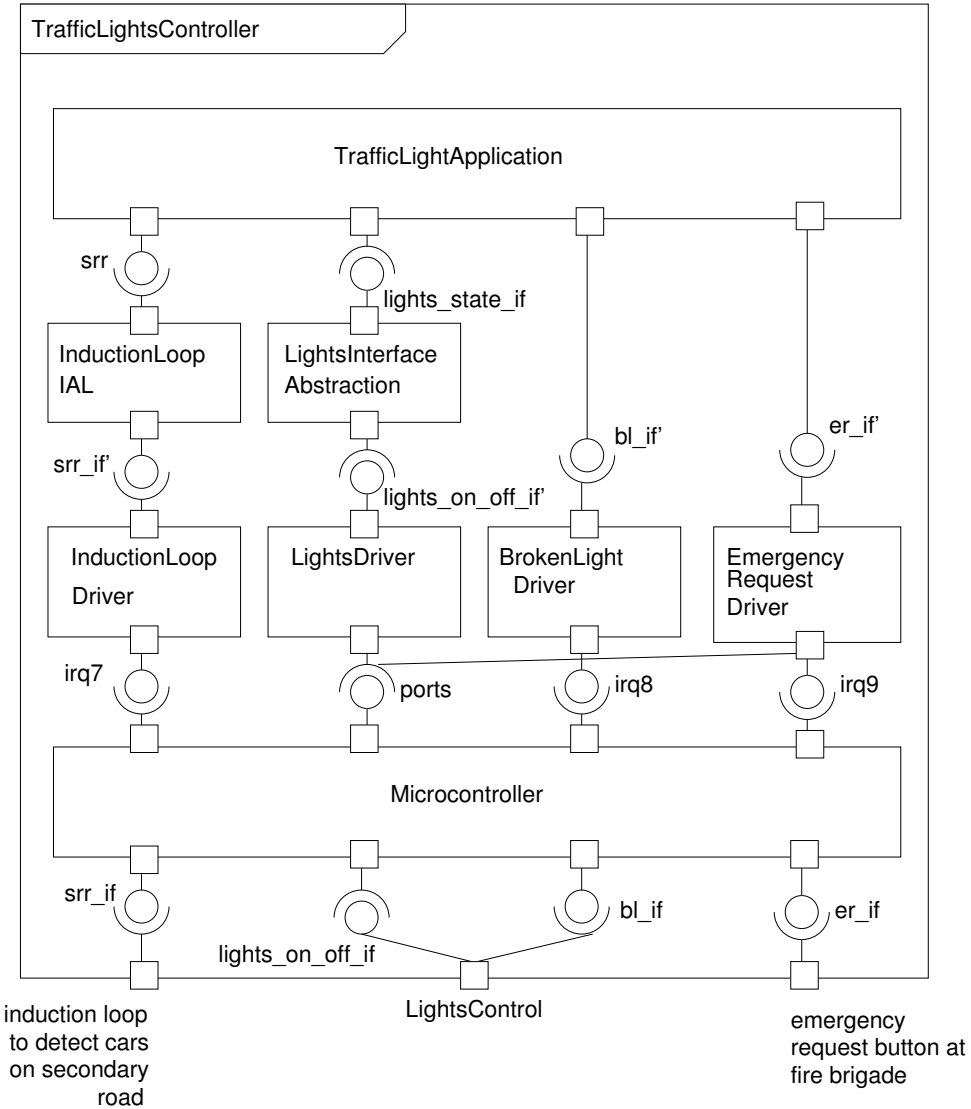


Variant of the required behavior.

TrafficLightsController BrokenLightSafeState Architecture



Global Architecture I



Global Architecture II

The components of the global architecture are merged using the following components of the subproblem architectures.

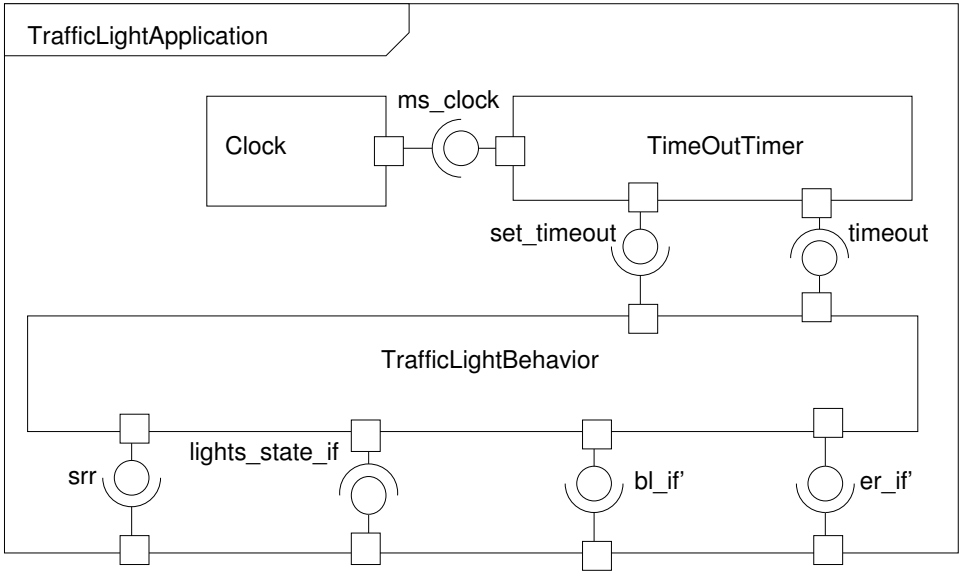
TrafficLightApplication	TrafficLightApplicationSRP, TrafficLightApplicationMRP, TrafficLightApplicationER, TrafficLightApplicationBL
IndictionLoopIAL	IndictionLoopIAL_MRP
LightsInterfaceAbstraction	LightsInterfaceAbstractionSRP, LightsInterfaceAbstractionMRP, LightsInterfaceAbstractionER, LightsInterfaceAbstractionBL
IndictionLoopDriver	IndictionLoopDriverMRP
LightsDriver	LightsDriverSRP, LightsDriverMRP, LightsDriverER, LightsDriverBL
EmergencyRequestDriver	EmergencyRequestDriverER
BrokenLightDriver	BrokenLightDriverBL
Microcontroller	Existing component

Global Architecture III

The components *LightsInterfaceAbstractionSRP*, *LightsInterfaceAbstractionMRP*, *LightsInterfaceAbstractionER*, and *LightsInterfaceAbstractionBL* are merged since they are associated to the same hardware device (Case 1).
The components *LightsDriverSRP*, *LightsDriverMRP*, *LightsDriverER*, and *LightsDriverBL* are merged since they are associated to the same hardware device (Case 1).
The components *TrafficLightApplicationSRP* and *TrafficLightApplicationMRP* implement sequential subproblems and are merged into one application component (Case 2). The merged component and the components *TrafficLightApplicationER* and *TrafficLightApplicationBL* belong to parallel subproblems and share all output phenomena. They are also merged, because the output must be generated in a way satisfying all subproblems (Case 3).

Global Architecture – TrafficLightApplication

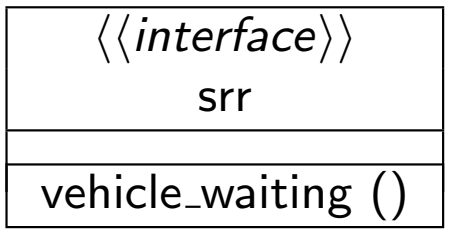
Since the component is complex, it is split into subcomponents. A *TimeOutTimer* and a *Clock* are introduced to separate the timers from the logic (in *TrafficLightBehavior*).

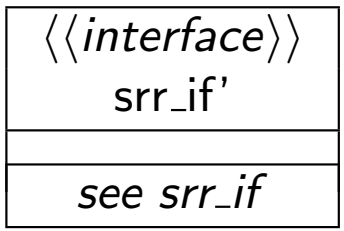


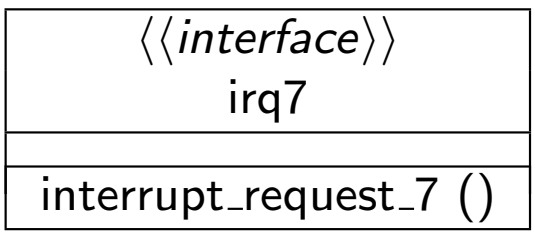
Global Architecture – Purpose of the components

Clock	Generates a pulse each millisecond.
TimeOutTimer	Sends a timeout message after a predefined time is elapsed.
TrafficLightBehavior	Control of Traffic Lights.
IndictionLoopIAL	Detects if a vehicle is waiting (based on a secondary road request). (More complex in real machines).
LightsInterfaceAbstractionIAL	Transforms lights commands for each road into commands for each light bulb.
IndictionLoopDriver	HAL for induction loop access.
LightsDriver	HAL for lights access.
EmergencyRequestDriver	HAL for emergency request button.
BrokenLightDriver	HAL for lights (broken light detection).
Microcontroller	Hardware running the application.

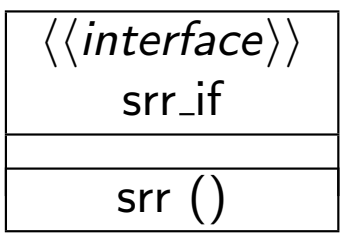
TrafficLightsControl interfaces I



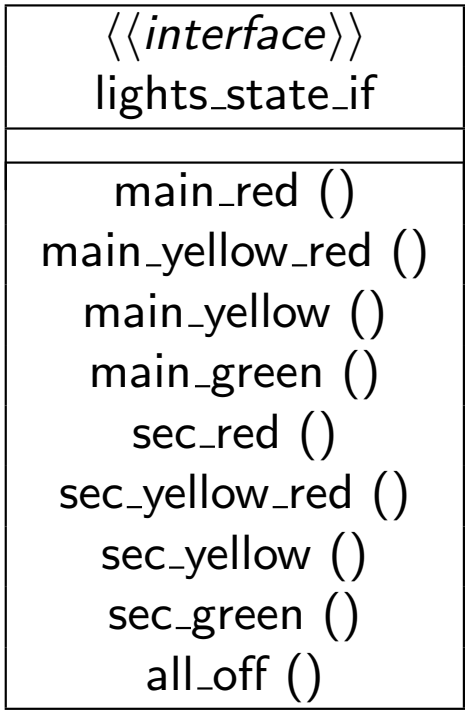


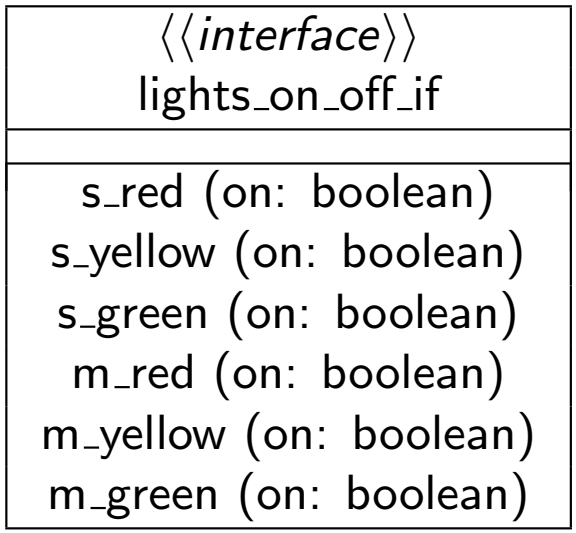


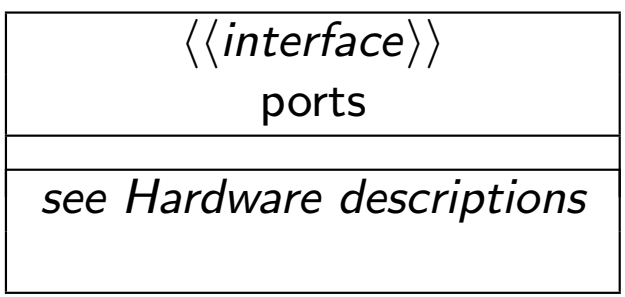
The microcontroller schematic and databook show that the *Induction-LoopControl* is connected to the pin of the microcontroller that generates the interrupt request with number 7.

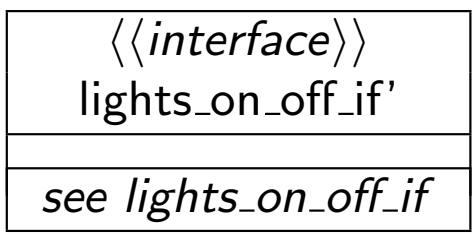


TrafficLightsControl interfaces II









TrafficLightsControl interfaces III

⟨⟨*interface*⟩⟩ bl_if
broken_light ()

⟨⟨*interface*⟩⟩ bl_if'
see bl_if

⟨⟨*interface*⟩⟩ irq8
interrupt_request_8 ()

The microcontroller schematic and databook show that the broken lights detection of *LightsControl* is connected to the pin of the microcontroller that generates the interrupt request with number 8.

TrafficLightsControl interfaces IV

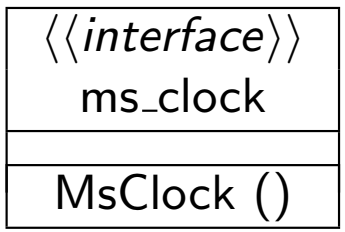
⟨⟨*interface*⟩⟩ er_if
emergency_request_start() emergency_request_end()

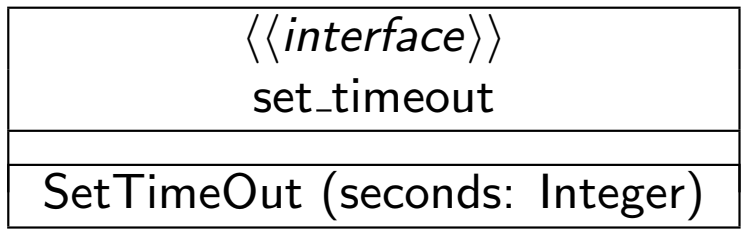
⟨⟨*interface*⟩⟩ er_if'
see er_if

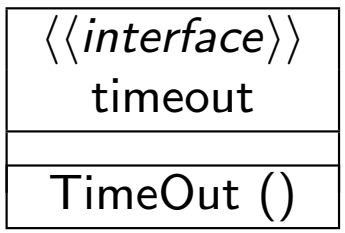
⟨⟨*interface*⟩⟩ irq9
interrupt_request_9 ()

The microcontroller schematic and databook show that the *button at the fire brigade* is connected to the pin of the microcontroller that generates the interrupt request with number 9.

TrafficLightsControl interfaces V





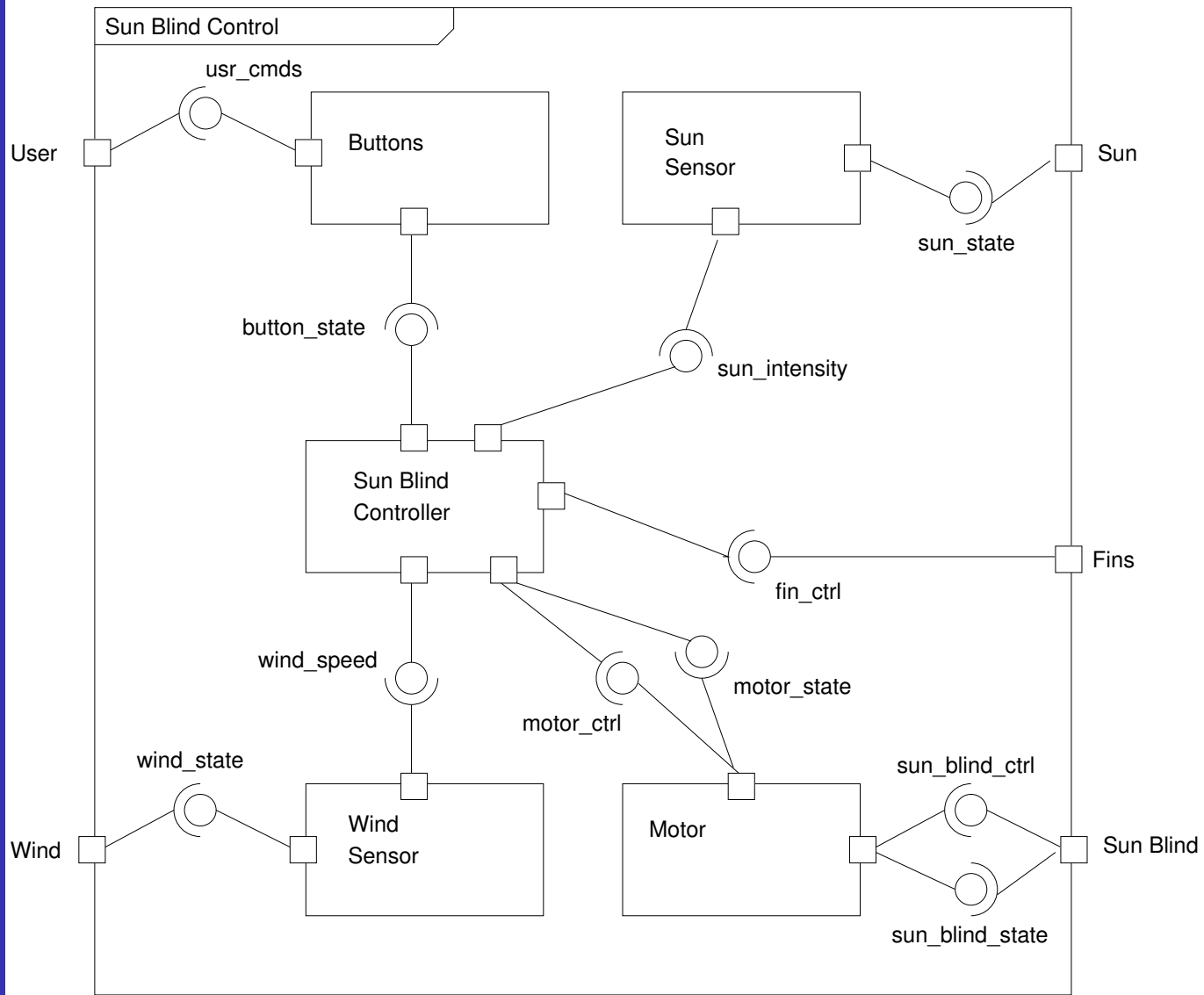


Validation

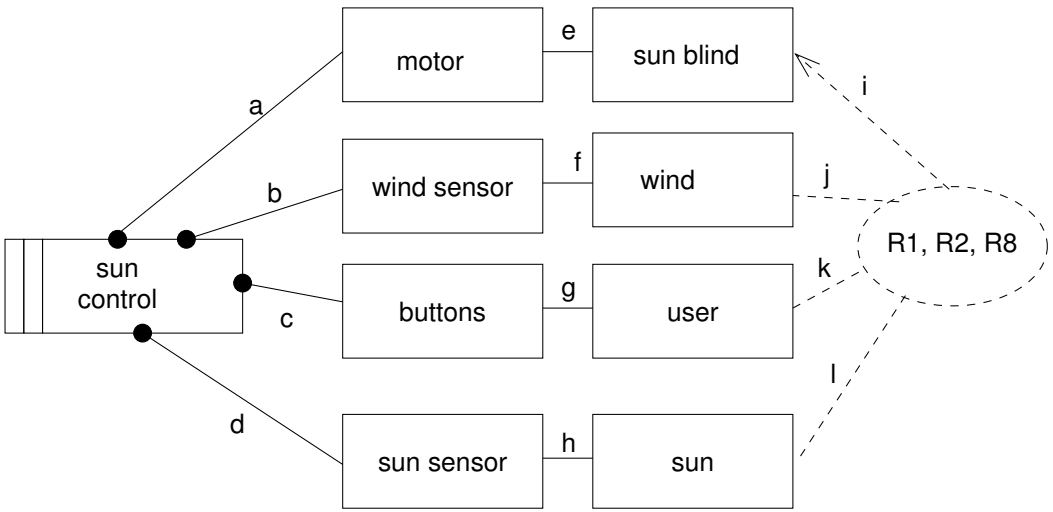
- The subproblem architectures have the same external interfaces as the problem diagrams.
- The signals of sequence diagrams at the external interfaces are the same as the signals in the interfaces of the application layer.
- The direction of all signals is consistent to each other and consistent to the input.
- The architecture has the same external interfaces as the traffic lights control component of the system architecture developed in Phase 5.
- The overall architecture contains all components of all subproblem architectures.

Example 2: sun blind control

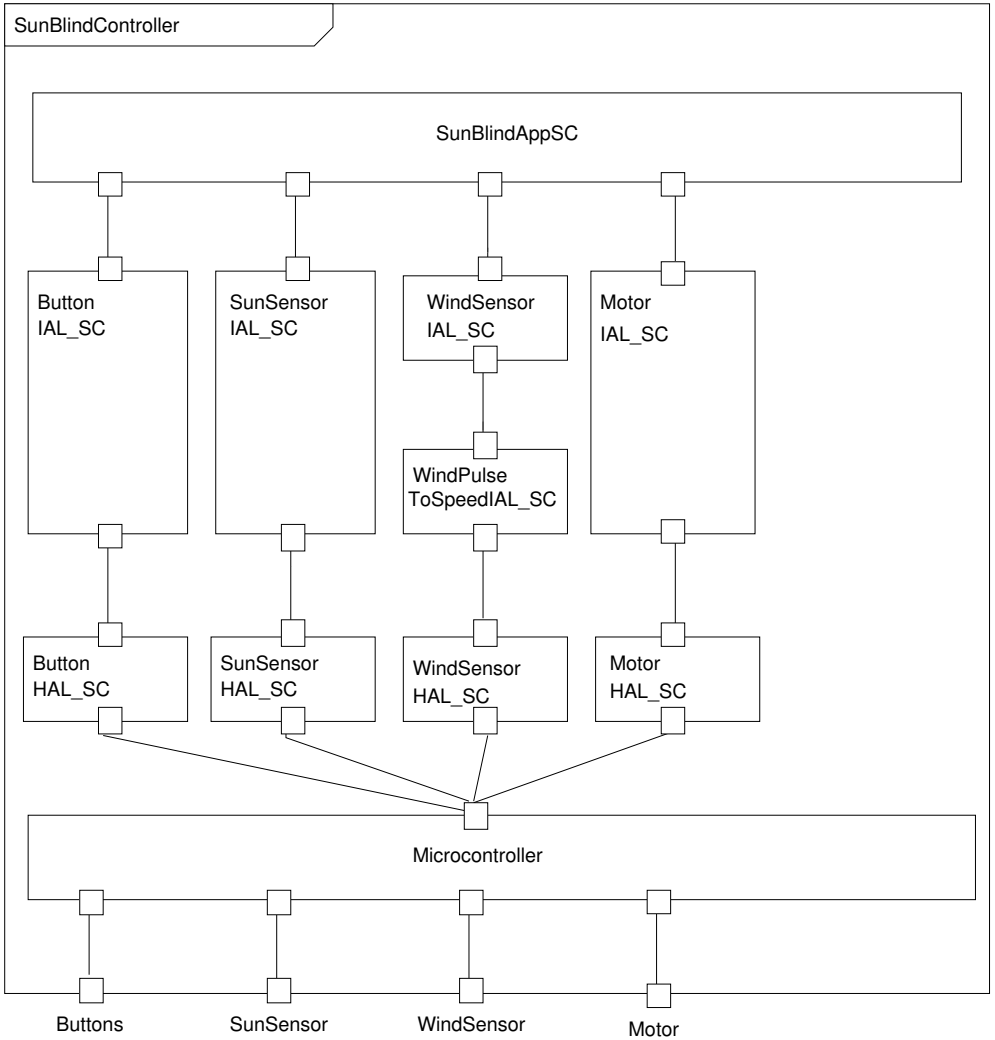
SunBlindControl system architecture



SunControl Problem Diagram I



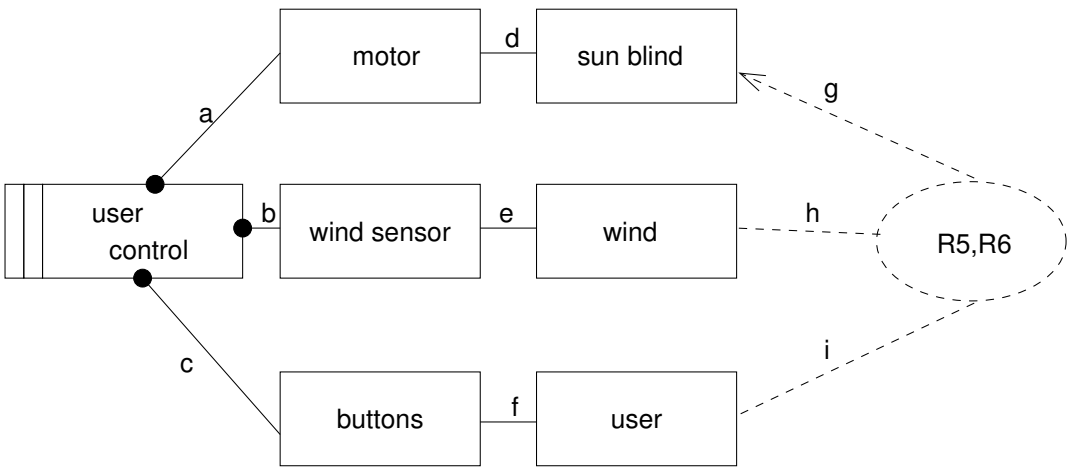
SunControl architecture



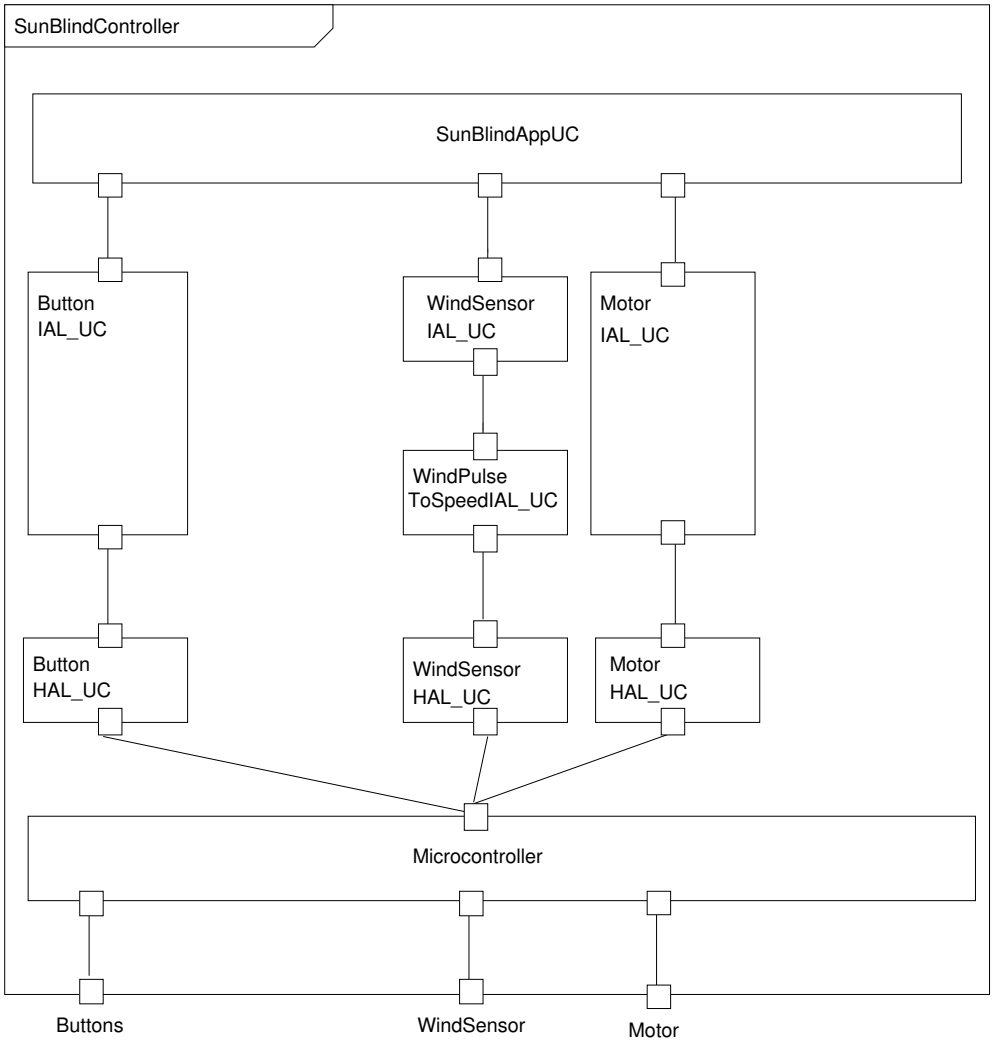
The IAL for the WindSensor is splitted since it has to perform two differnt task (transform pulses to a speed, calculates if there is heavy wind).

The Microcontroller is a reused component.

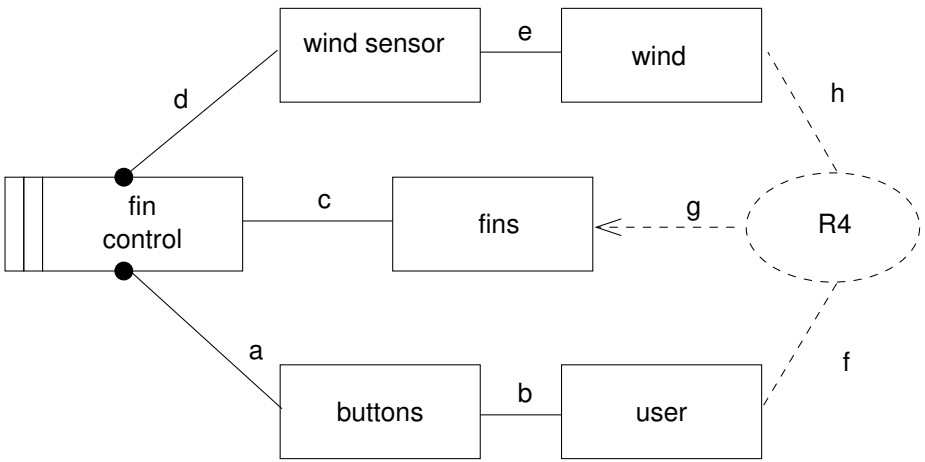
UserControl problem diagram



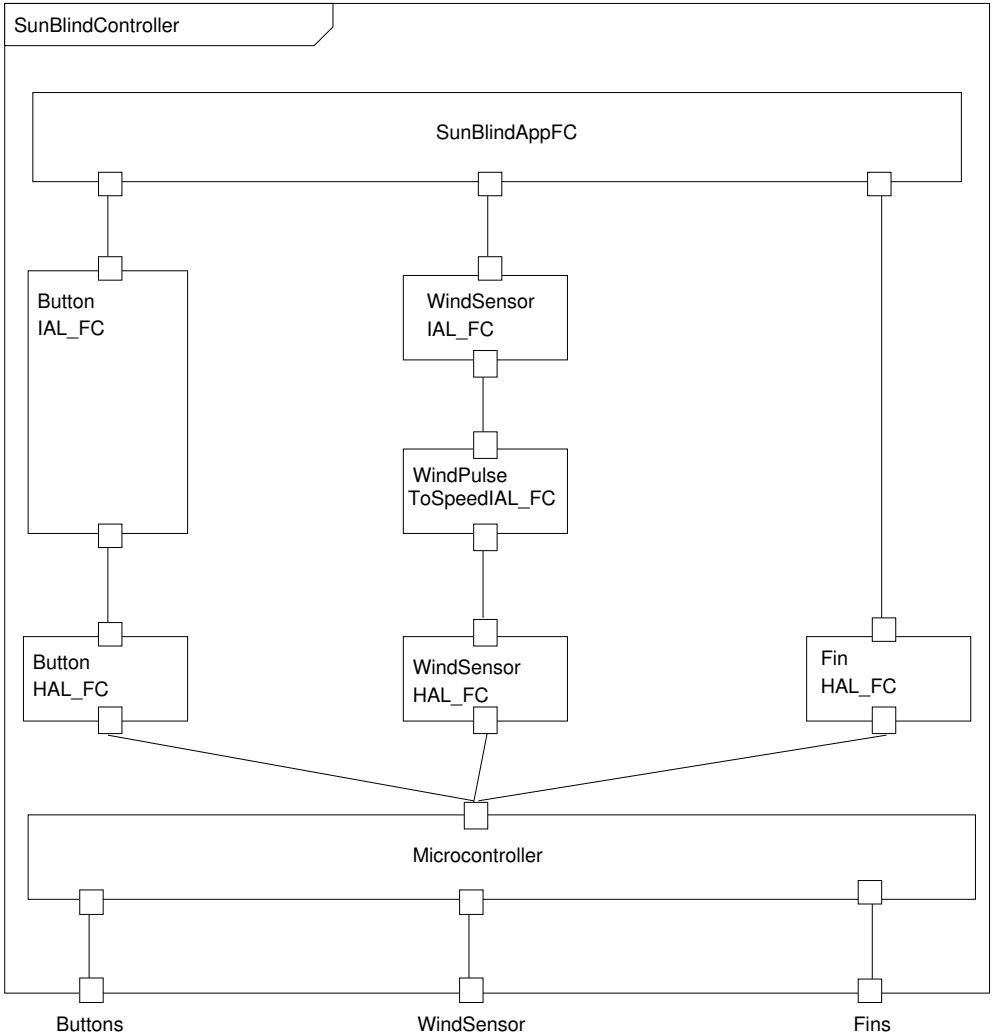
UserControl architecture



FinsControl problem diagram

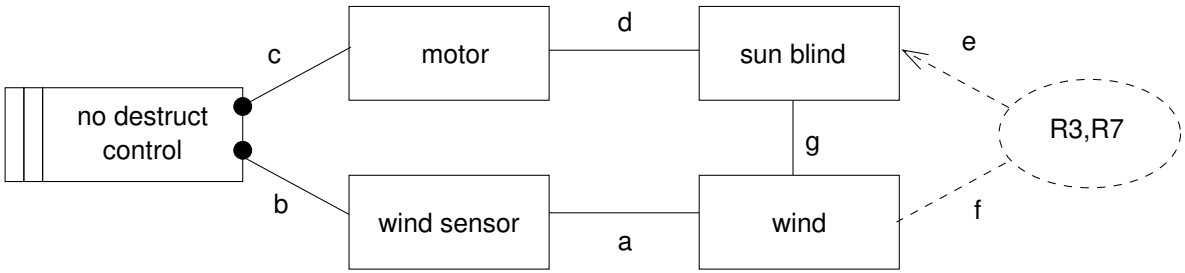


FinsControl architecture

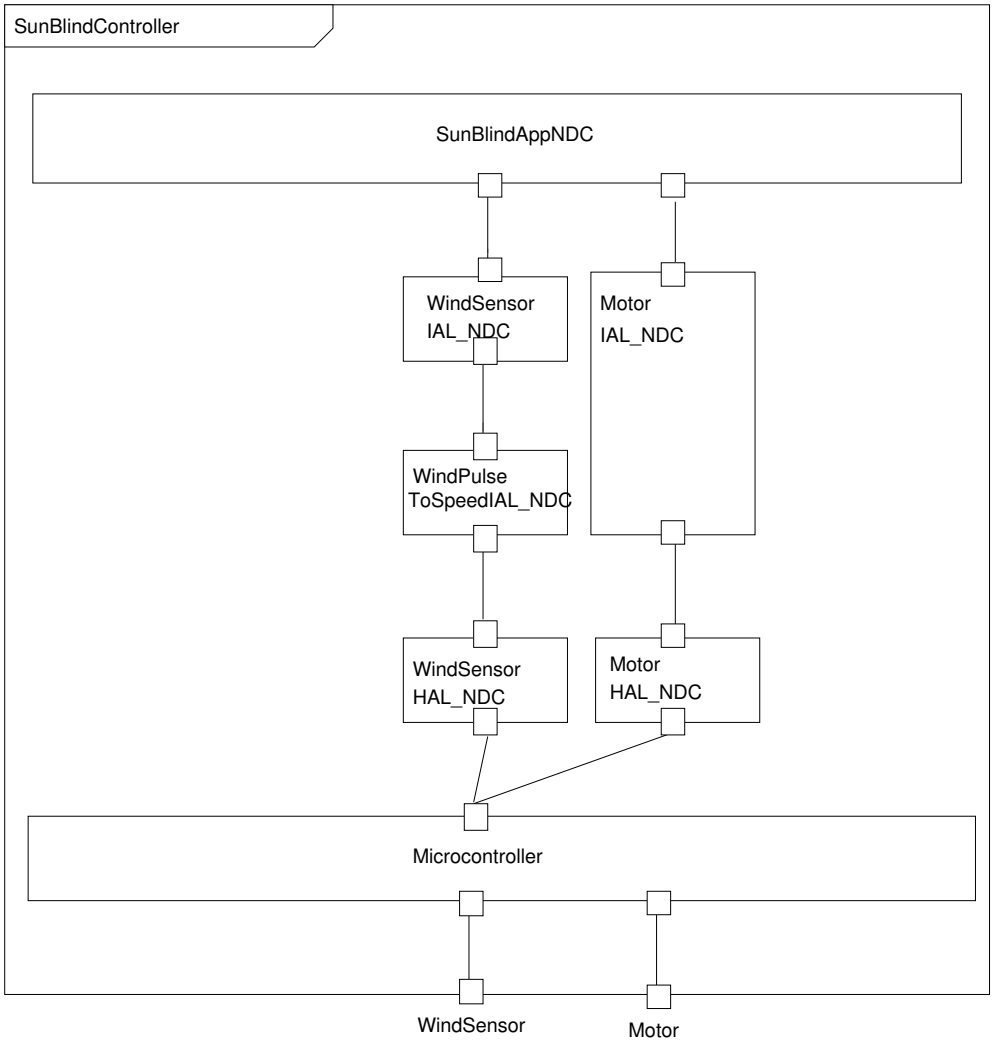


There is no IAL for the fins since the requirements for the fins are the same as the specification and no hardware component is necessary to control the fins.

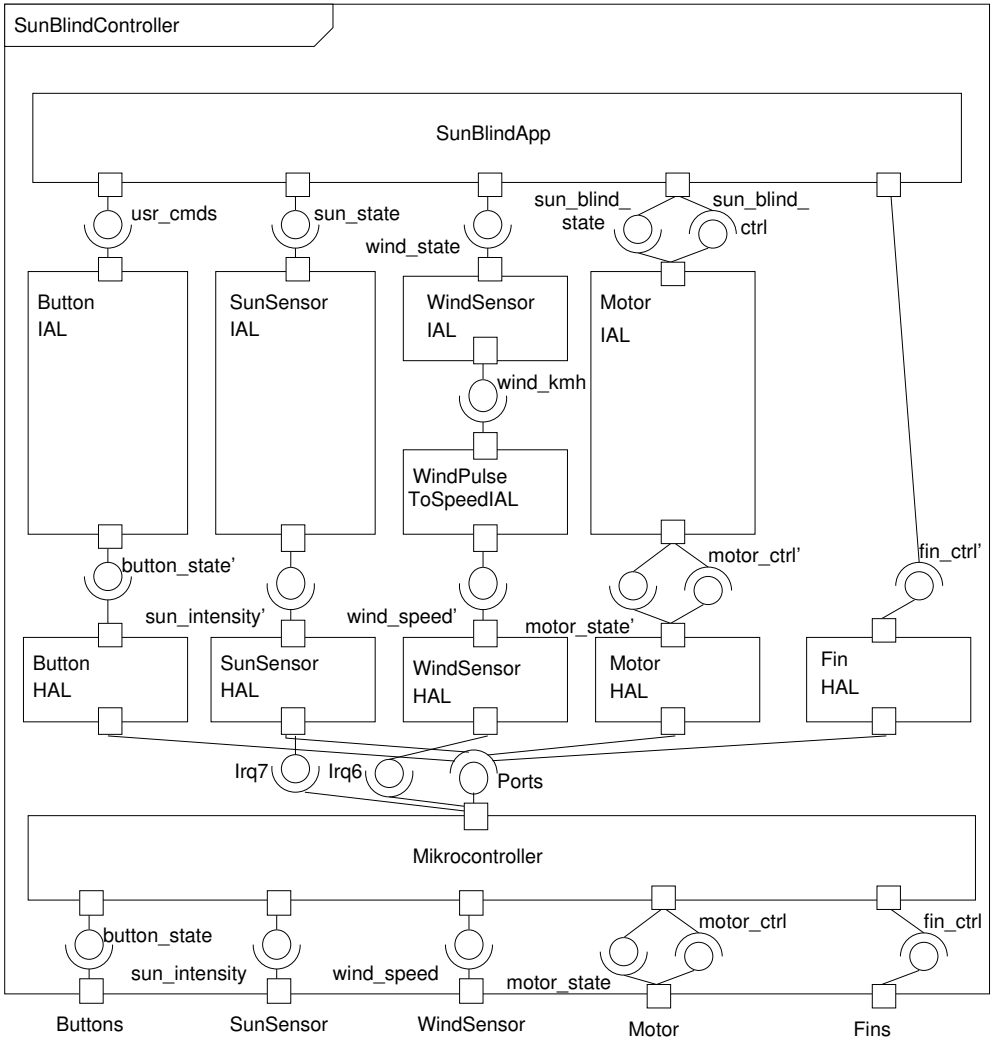
NoDestructControl problem diagram



NoDestructControl architecture

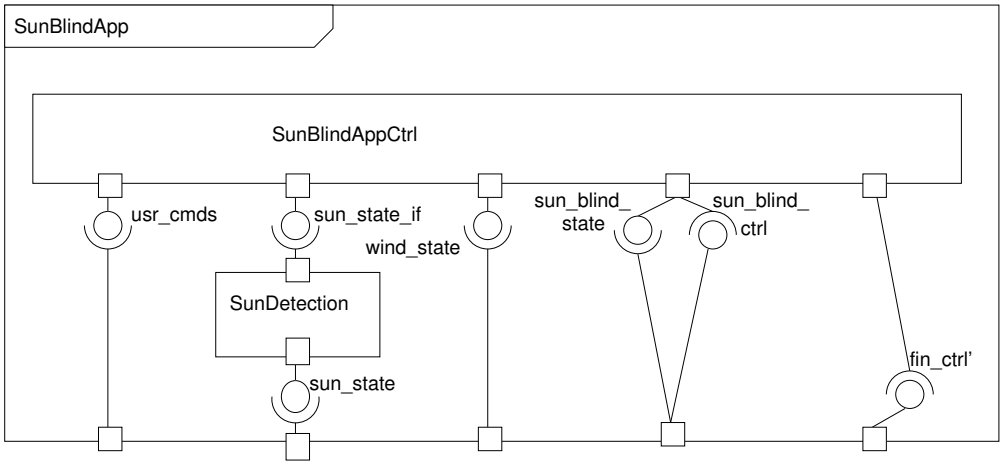


Global architecture



Components with similar names are merged.

Global architecture – SunBlindApp



In this example the timer is an internal class.
The component *SunDetection* was seperated since the state machine of the whole component *SunBlindApp* is too complex.

Global architecture – Purpose of the Components I

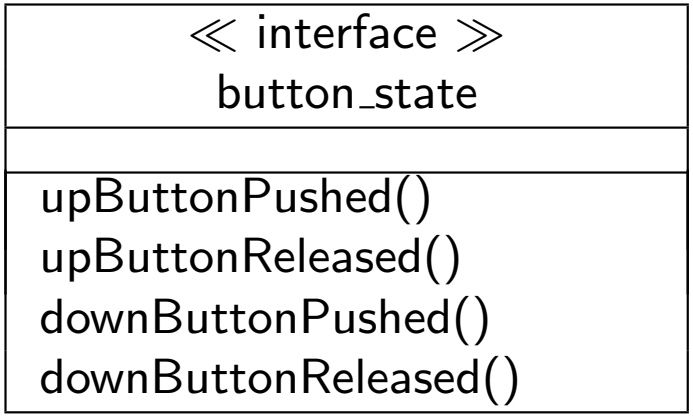
SunBlindApp	Control of SunBlind according to state of Blind, Buttons, Sun and Wind. Control of the Fins according to Buttons and Wind.
SunDetection	Calculates if the sun is shining or not shining for a certain period of time.
SunBlindAppCtrl	Control of SunBlind according to Buttons, Sun and Wind.
ButtonIAL	Transforms button state to intended user commands.
SunSensorIAL	Calculates if sun is shining or not based on intensity.
WindSensorIAL	Calculates if there is heavy wind or not based on speed.
WindPulseToSpeedIAL	Calculates wind speed from sequence of pulses.

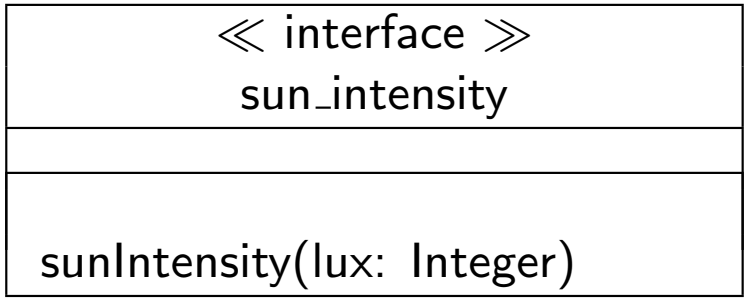
Global architecture – Purpose of the Components II

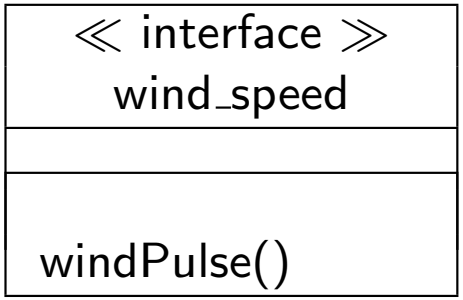
MotorIAL	Transforms sun blind commands into motor commands and motor state into sunblind state.
ButtonHAL	HAL for button access.
SunSensorHAL	HAL for sun sensor access.
WindSensorHAL	HAL for wind sensor access.
MotorHAL	HAL for motor access.
FinsHAL	HAL for fin access.
Microcontroller	Hardware running the application.

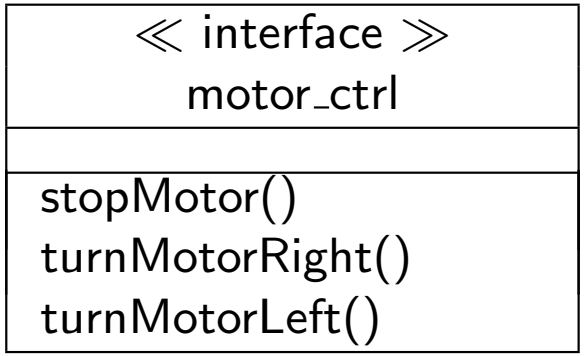
SunControl interfaces I

Hardware interfaces:

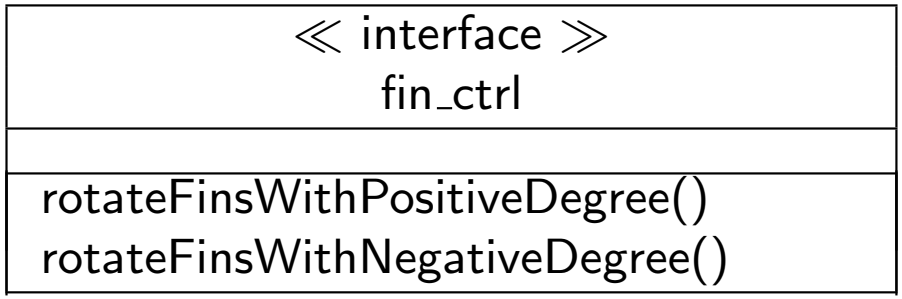




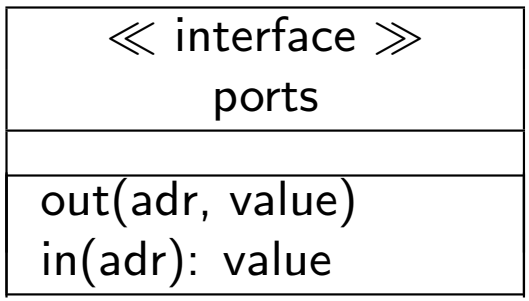


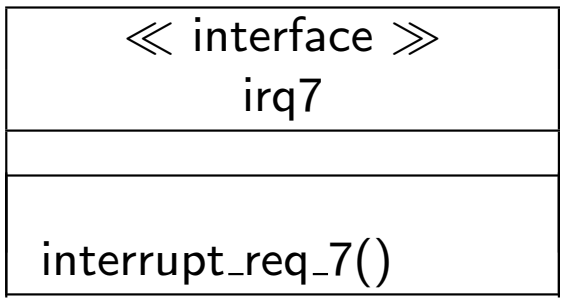


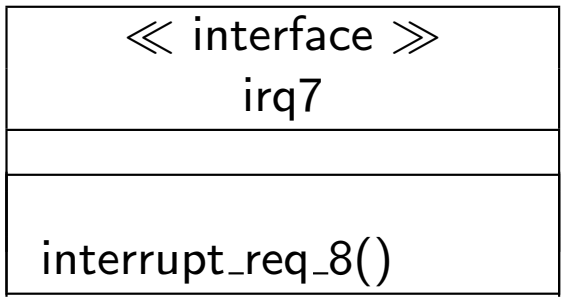
SunControl interfaces II



Microcontroller interfaces:

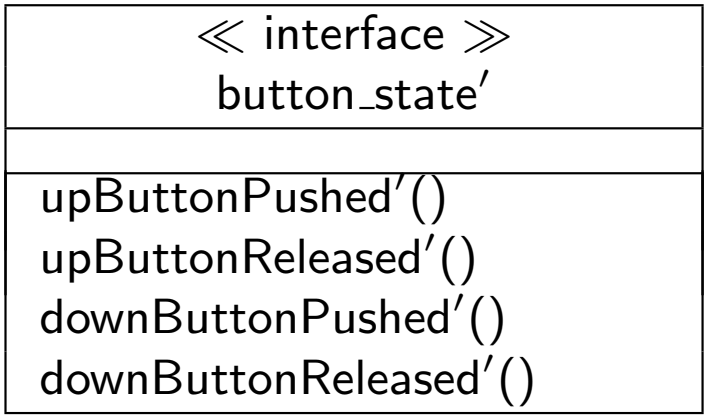


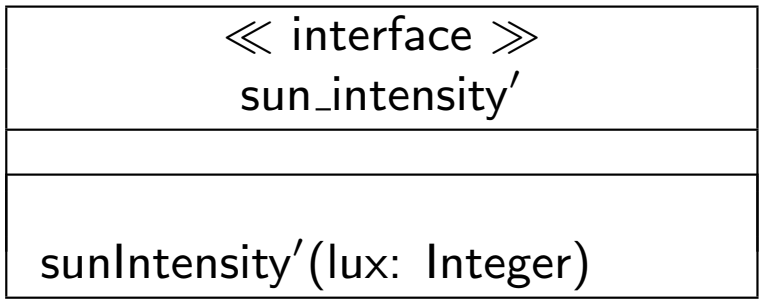


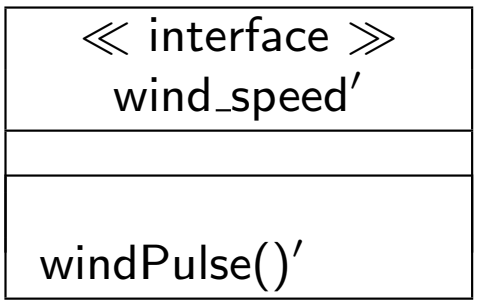


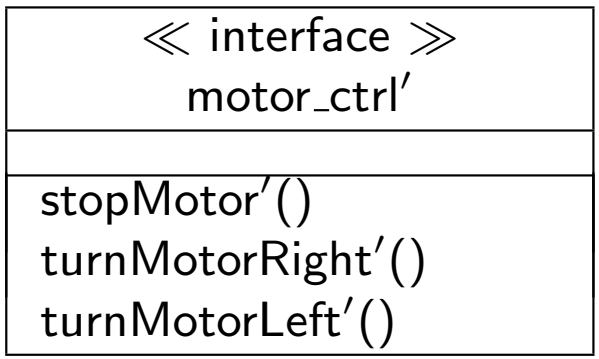
SunControl interfaces III

HAL interfaces:









SunControl interfaces IV

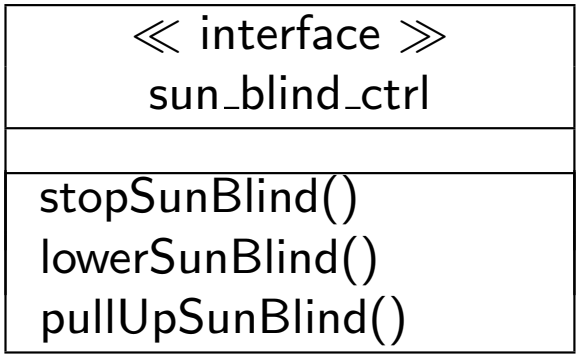
≪ interface ≫ fin_ctrl′
rotateFinsWithPositiveDegree′() rotateFinsWithNegativeDegree′()

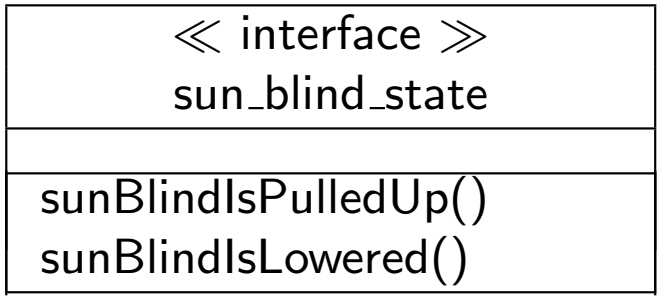
IAL Interfaces:

≪ interface ≫ usr_cmds
manuallyOpenSunBlind() manuallyCloseSunBlind() adjustFinsPositiveDegree() adjustFinsNegativeDegree()

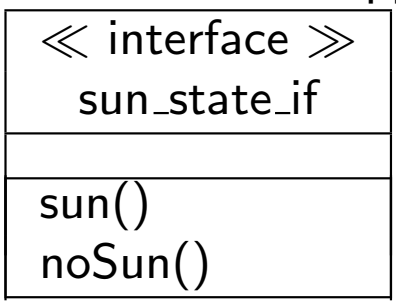
≪ interface ≫ sun_state
sunShine() noSunShine()

SunControl interfaces V



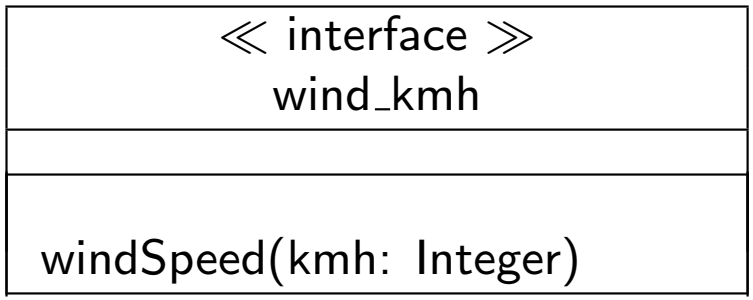


Interface inside application:



SunControl interfaces VI

Interface inside IAL:



Validation

- The subproblem architectures have the same external interfaces as the problem diagrams.
- The phenomena of sequence diagrams at the external interfaces are the same as the signals in the interfaces of the application layer.
- The direction of all signals is consistent to each other and consistent to the input.
- The architecture has the same external interfaces as the sun blind controller component of the system architecture developed in Phase 5.
- The overall architecture contains all components of all subproblem architectures.

Phase 8: Specify the behavior of all components of all software architectures, using sequence diagrams

...

5. Design global system architecture
6. Derive specifications for all components of the global system architecture
7. Design a software architecture for all components of the global system architecture that should be implemented in software
8. Specify the behavior of all components of all software architectures, using sequence diagrams
9. Specify the software components of all software architectures as state machines
10. Implement software components and test environment

...

Phase 8: Specify the behavior of all components of all software architectures, using sequence diagrams

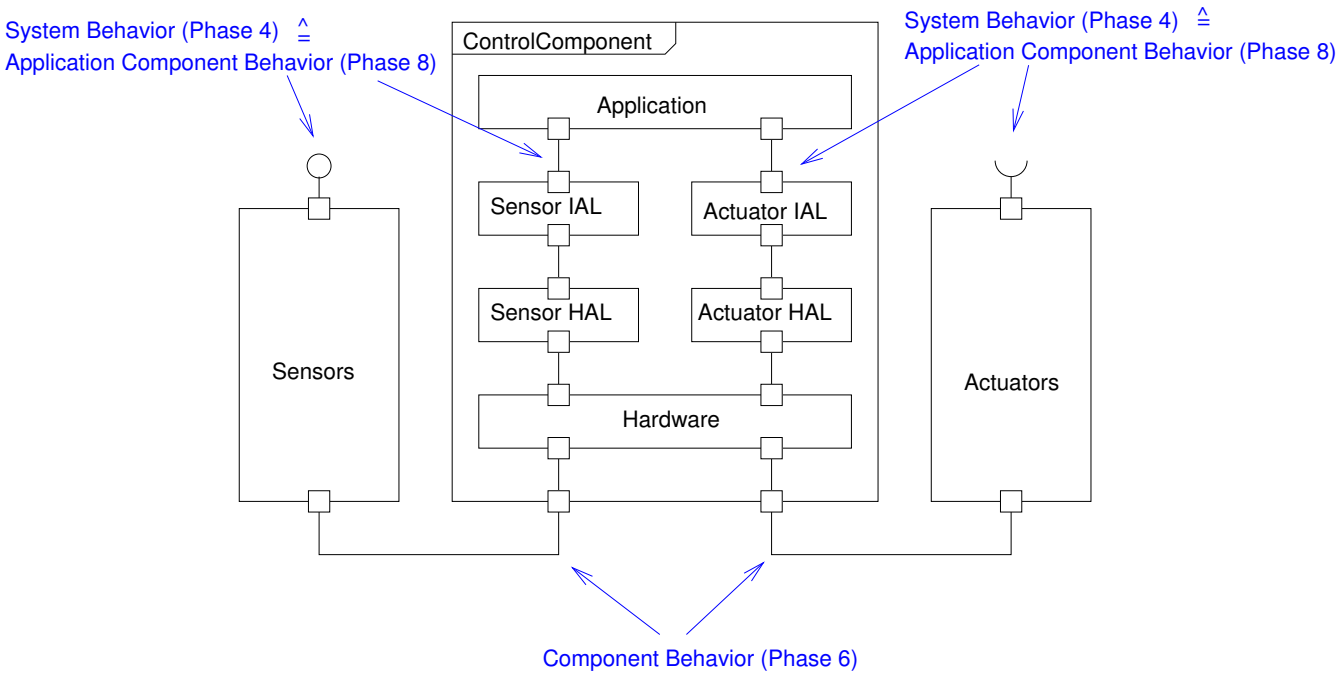
For each subproblem:

input:	software architectures from Phase 7	composite structure diagrams
	interface specifications from Phase 7	interface classes
	system behavior from Phase 4	sequence diagrams with annotated states
	interface behavior of all programmable components from Phase 6	sequence diagrams with annotated states
output:	interface behavior of all software components (test specification)	sequence diagrams with annotated states
validation:	all sequence diagrams together must describe the same interface behavior as in Phase 6	
	all signals in the interfaces classes of Phase 7 must be used in at least one sequence diagram	
	direction of signals must be consistent with the required and provided interfaces of Phase 7	
	signals must connect components as connected in the software architecture of Phase 7	
	it must be possible to map any new states to the states of Phase 6	

Notations and concepts

- Four-variable model (repetition)
- Transformation of timing constraints

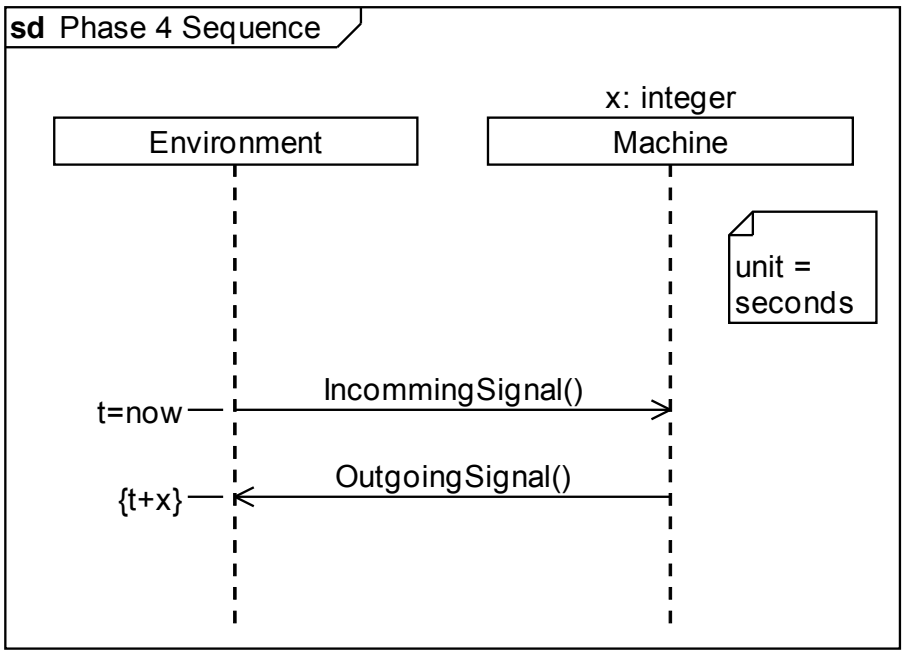
Four-variable model



- Application layer software should have the the same interfaces as the system, i.e., monitored and controlled variables.
- States of the environment will be mapped to internal states.

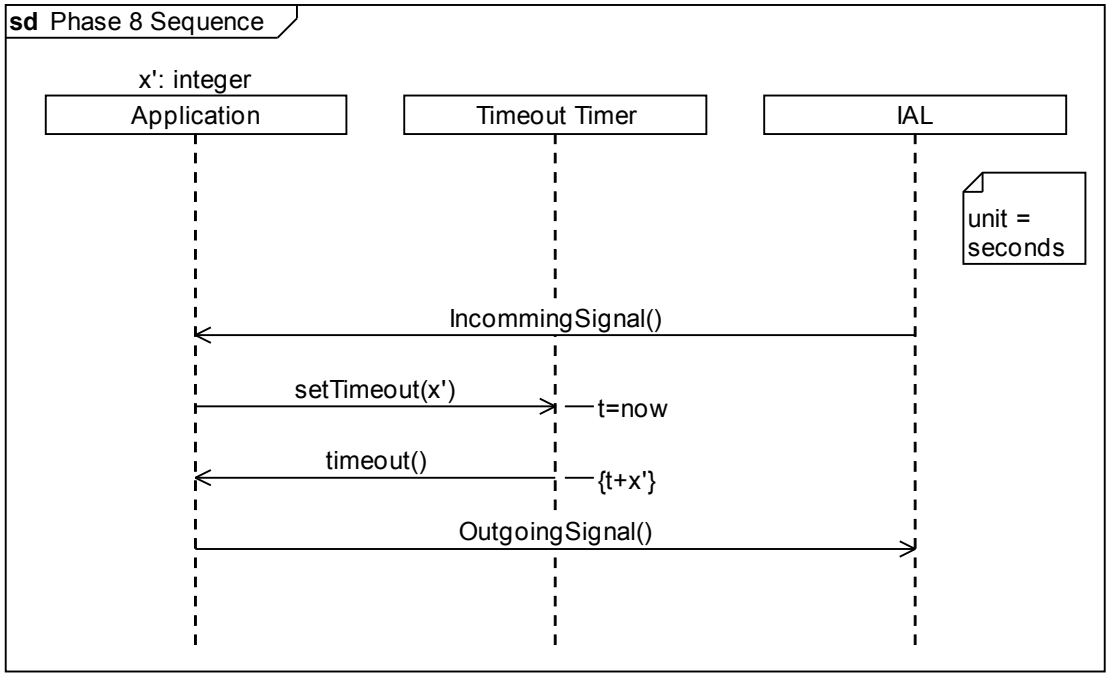
Transformation of timing constraints I

In Phase 4, timing constraints are specified as shown in the following figure.



Transformation of timing constraints II

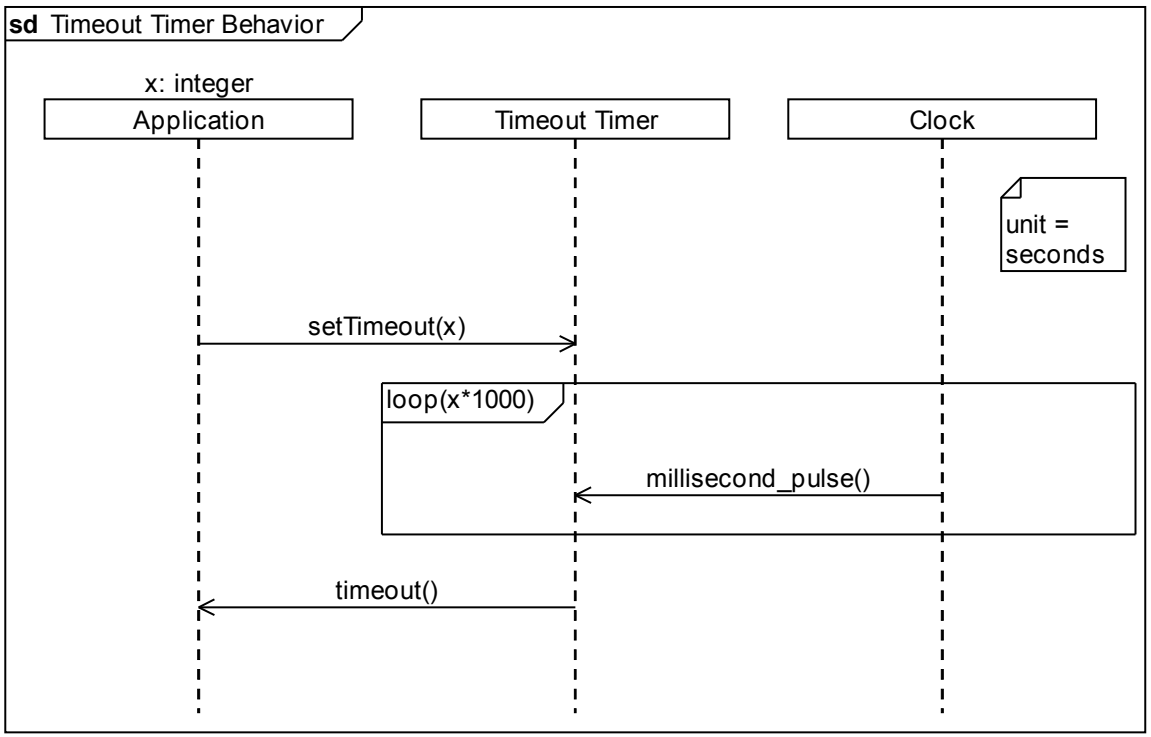
These timing constraints can be transformed in this phase into implementable events to reuse the specification from Phase 4, e.g., as shown in the following figure:



x' is derived from x according to the expected execution time of the application component and the hardware devices.

Transformation of timing constraints III

The *Timeout Timer* can be specified as follows:



From the timing constraints, concrete requirements for the execution time of the application component and the precision of the clock can be derived.

Executing Phase 8 I

To create the sequence diagrams, for each software component and each subproblem the following steps have to be performed:

- Draw a lifeline for the software component to be specified. Either introduce a lifeline for the connected components, or connect the arcs representing a signal with the left and right border of the sequence diagram.
- Describe the interface behavior of the component using the signals from the software architecture.
- The specification of the application components should be reused from Phase 4.

Executing Phase 8 II

- The specification for the interface abstraction layer can be derived from the domain knowledge used to derive the specification and the specification of the other components in the system architecture, expressed as sequence diagrams.
- The specification for the hardware abstraction layer should show the mapping from the IAL to the hardware and vice versa. Since this specification is usually described in the data book and in the schematic of the hardware, only a reference must be given.
- If possible, refer to other sequence diagrams and do not draw diagrams for the same sequence several times.

Executing Phase 8 III

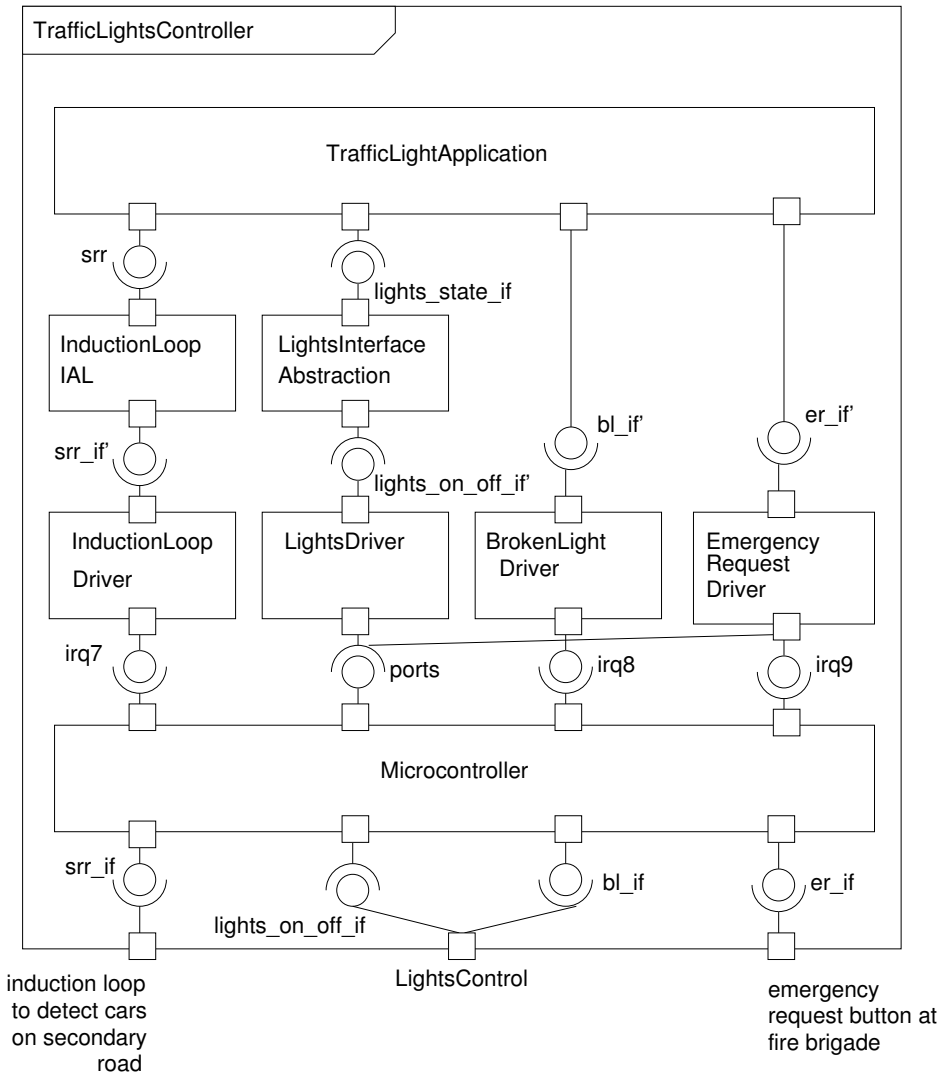
- Add states where they are relevant to describe the behavior. Map the defined states to the states in the environment.
- Add missing sequence diagrams to describe the behavior for all relevant states.
- Add timing constraints if necessary.
- Specify the initialization sequences. They describe the state of the software components after initializing the components (e.g., after power-on).

Remarks

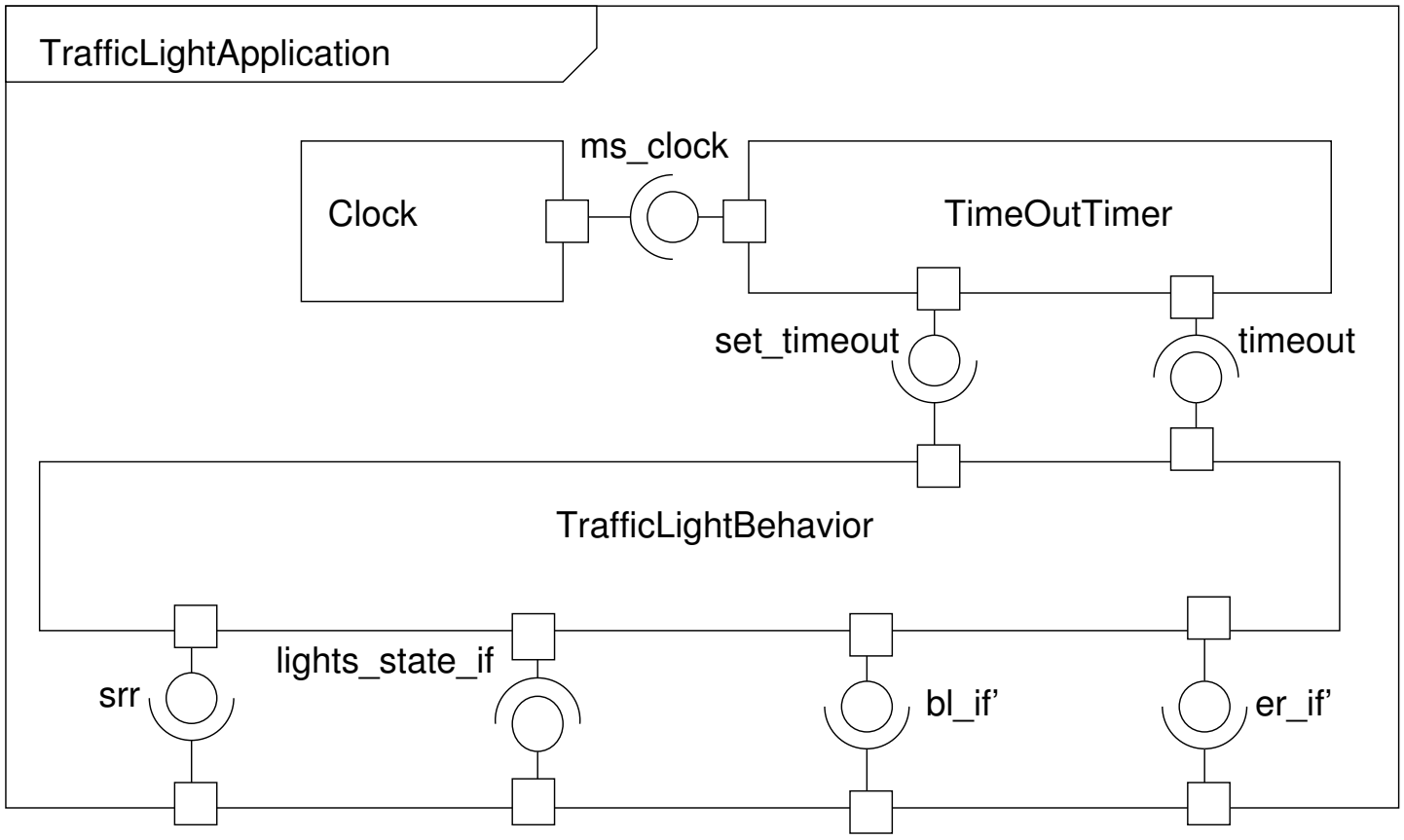
- In this phase, each component is described separately.
- Do not forget the extended four-variable model (for reuse of specifications).
- A specification expressed somewhere else should be referenced and does not have to be translated into sequence diagrams.
- The sequence diagrams developed in this phase are a concrete basis for the implementation of test cases for all software components.

Example 1: traffic light control

Global software architecture I



Global software architecture II



Behavior of the components I

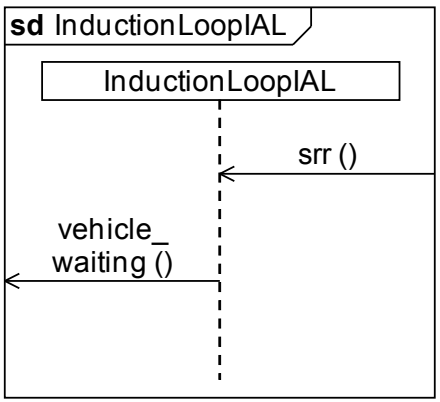
TrafficLightBehavior – all subproblems: same as in Phase 4 (also for the initialization sequence), see four-variable model and transformation of timing constrains

Clock – all subproblems: reused component without states (initialization sequence not necessary).

TimeOutTimer – all subproblems: see slide *Transformation of timing constraints*.

Behavior of the components II

InductionLoopIAL – subproblem MainRoadPassing:

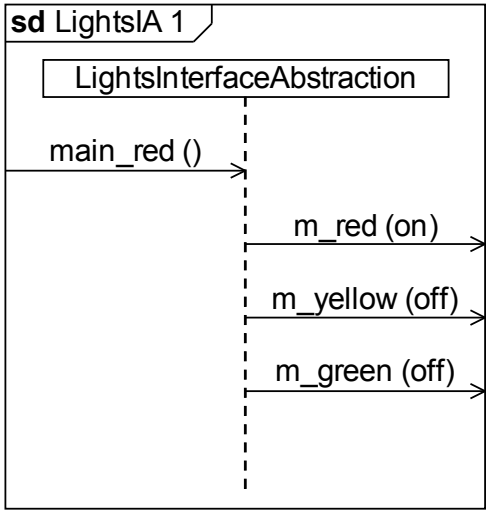


An initialization sequence is not necessary since no states are specified.

InductionLoopDriver – subproblem MainRoadPassing: see *InductionLoopIAL* and description of the Microcontroller.

Behavior of the components III

LightsInterfaceAbstraction – all subproblems:
The complete specification for this component can be derived from the domain knowledge about the lights compontent in Phase 4.



(example)

An initialization sequence is not necessary since no states are specified.

LightsDriver – all subproblems: see *LightsInterfaceAbstraction* and description of the Microcontroller.

Behavior of the components IV

BrokenLightDriver – subproblem BrokenLightSafeState: see description of the Microcontroller.

EmergencyRequestDriver – subproblem EmergencyRequestSecondaryRoadPassing: see description of the Microcontroller.

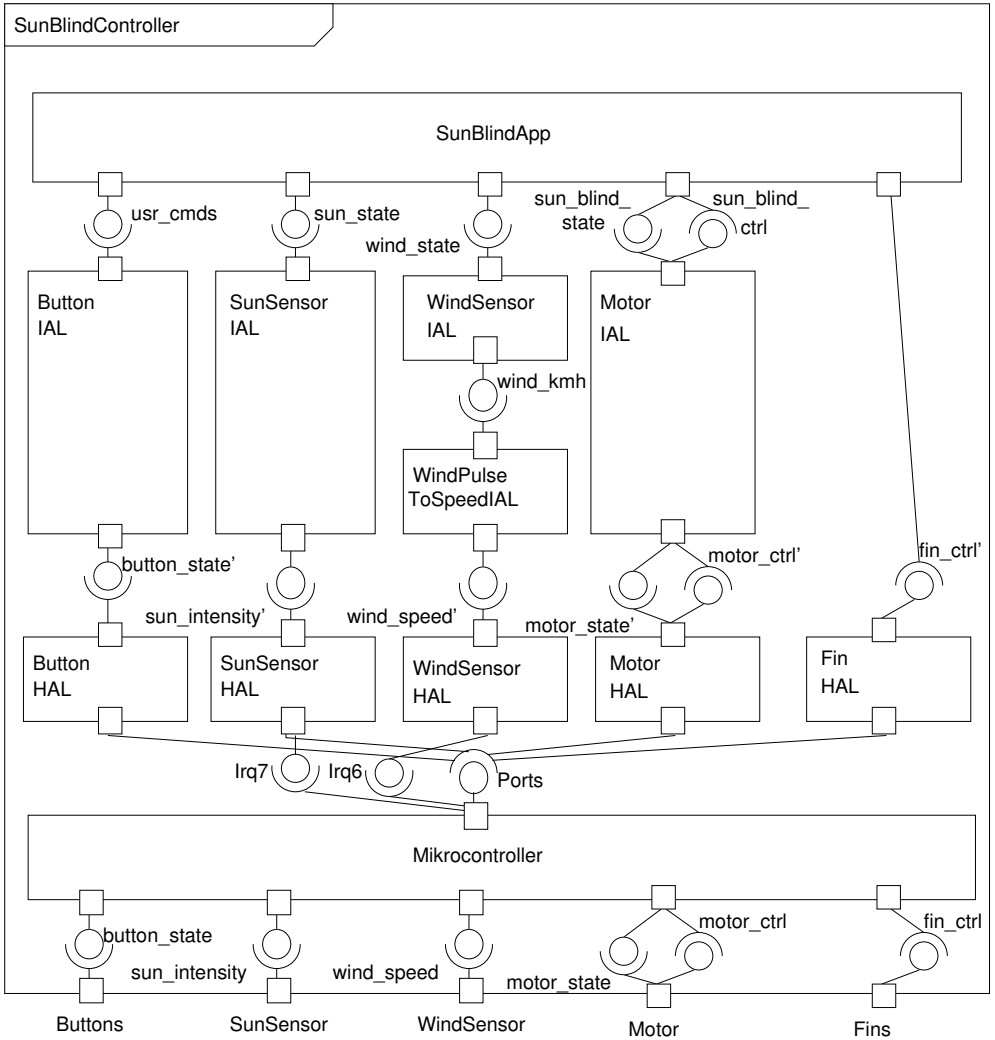
Microcontroller – all subproblems: The behavior is described in the Microcontroller data book and therefore not specified here.

Validation

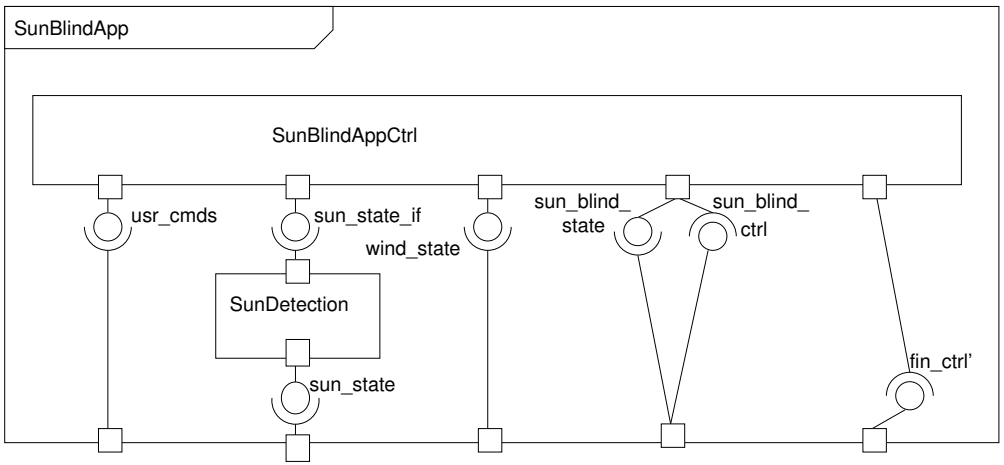
- All sequence diagrams together describe the same behavior as in Phase 6.
- All signals in the interfaces classes of Phase 7 occur in the specification of at least one component (in the complete specification).
- The direction of the signals are consistent with the required or provided interfaces of Phase 7.
- The signals connect the same components as connected in the software architecture of Phase 7.
- The states of the environment are mapped 1:1 to the application states.

Example 2: sun blind control

Global software architecture

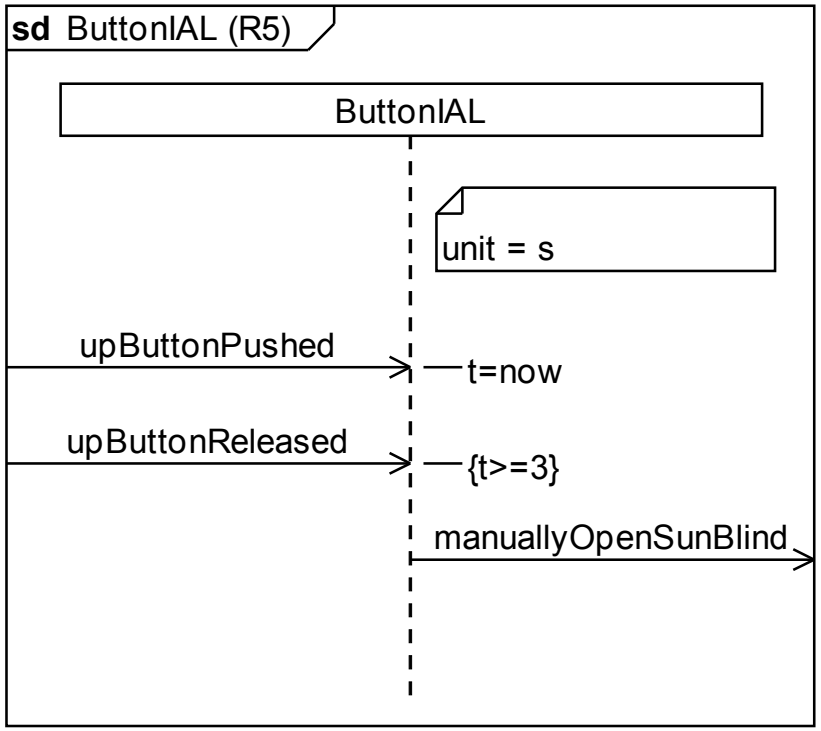


Software architecture of the application

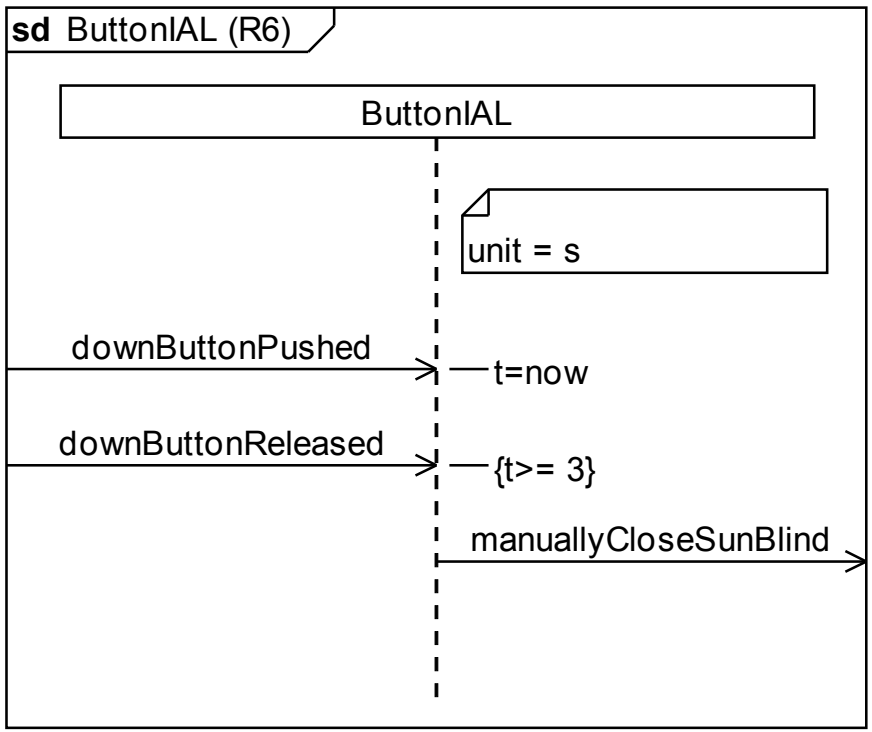


Behavior of the Components I

ButtonIAL – subproblems SunControl and UserControl

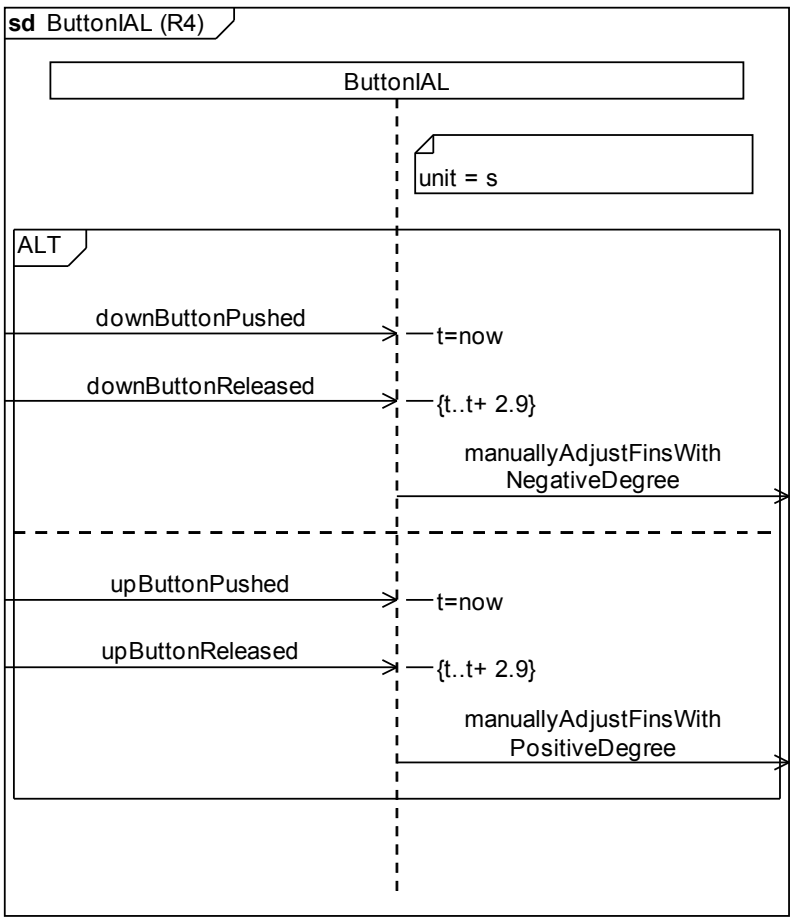


Behavior of the Components II



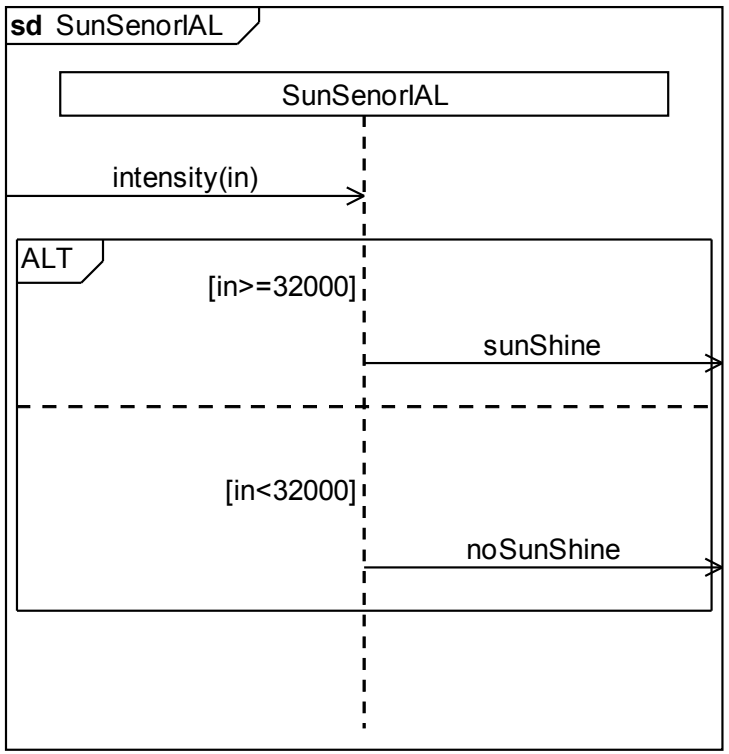
Behavior of the Components III

ButtonIAL – subproblems SunControl and FinsControl



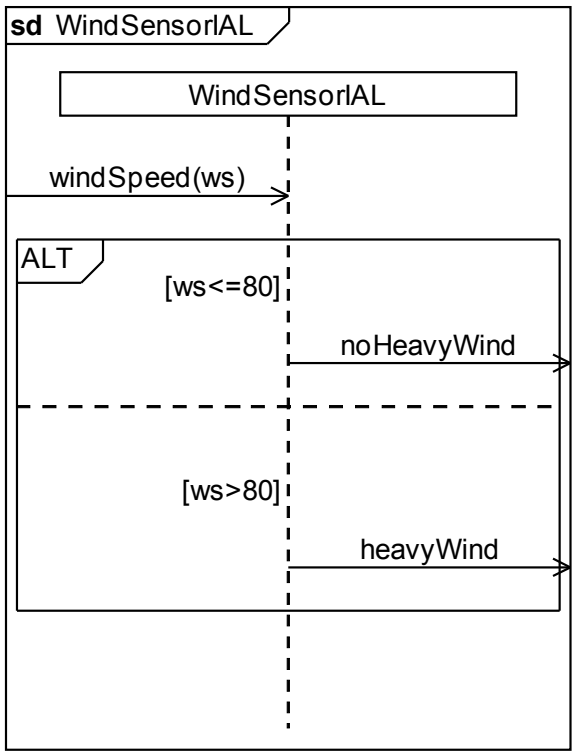
Behavior of the Components IV

SunSensorIAL – subproblem SunControl



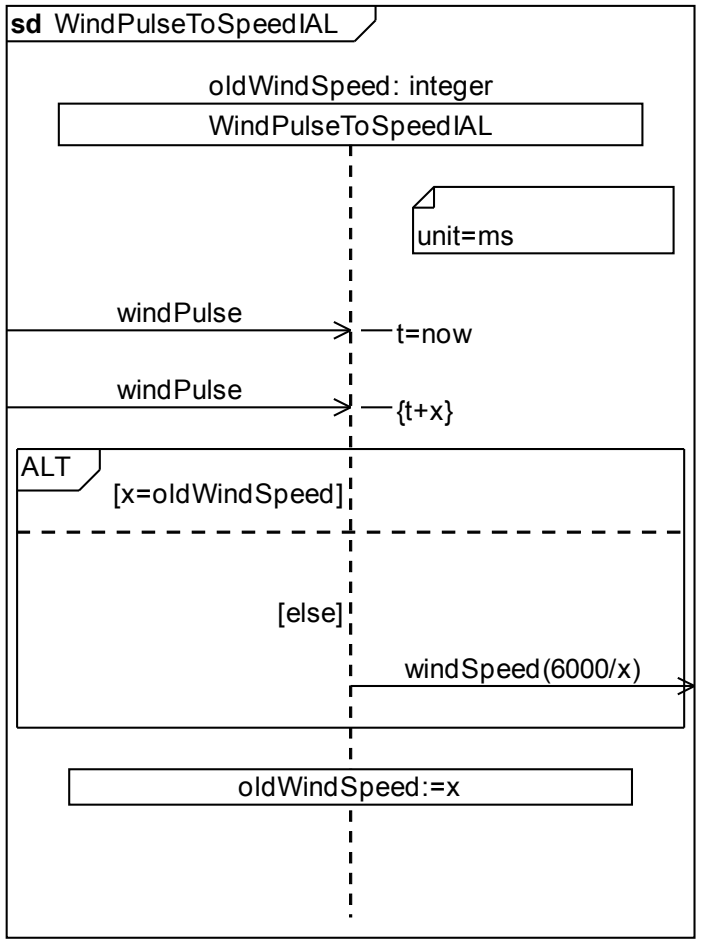
Behavior of the Components V

WindSensorIAL – all subproblems



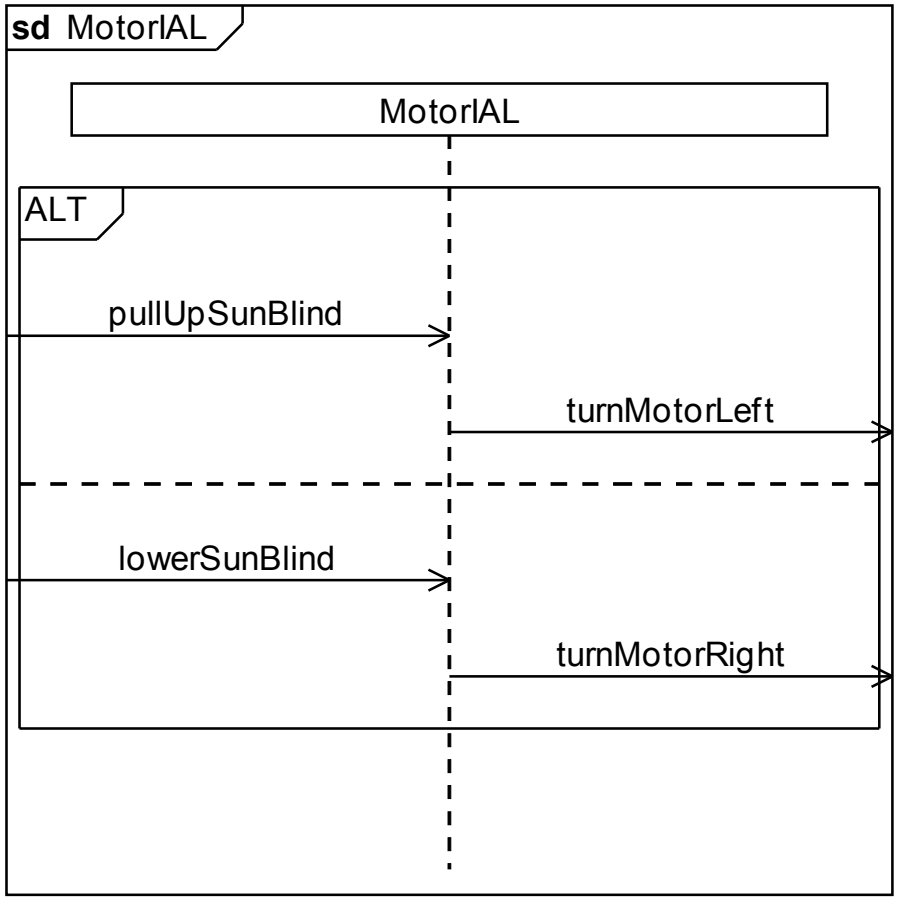
Behavior of the Components VI

WindPulseToSpeedIAL – all subproblems



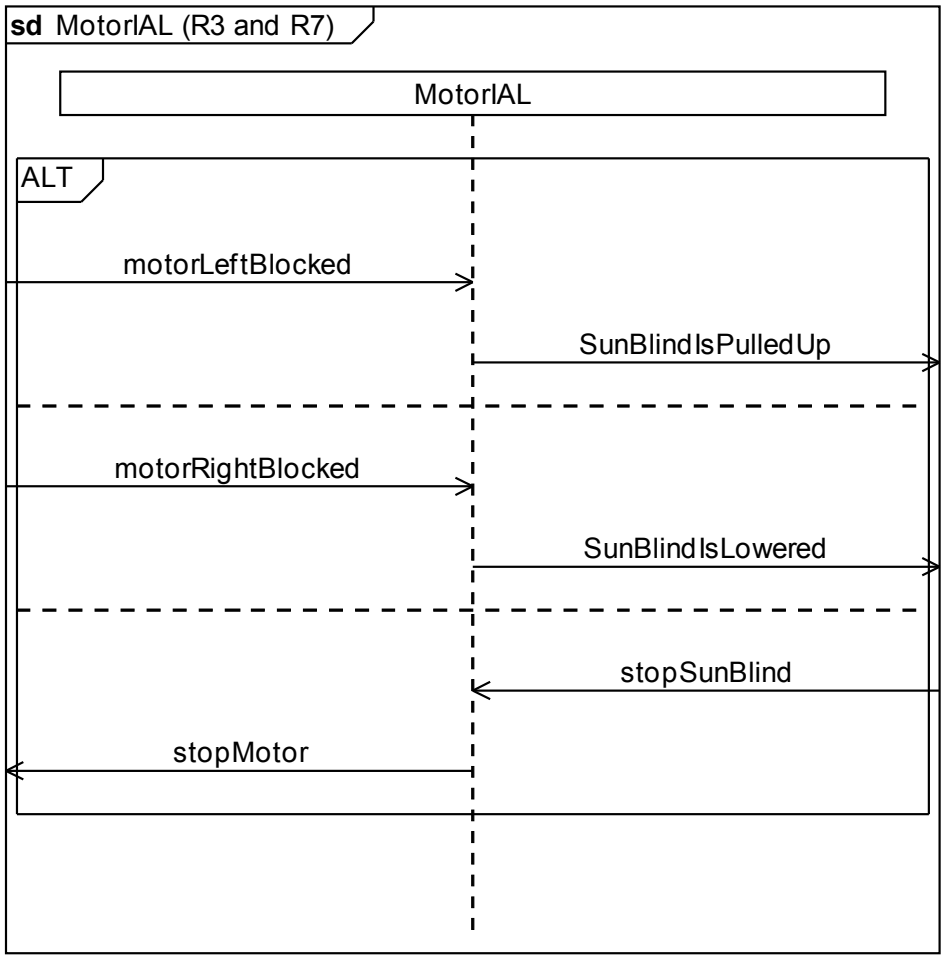
Behavior of the Components VII

MotorIAL – subproblems UserControl and SunControl



Behavior of the Components VIII

MotorIAL – subproblem NodestructControl

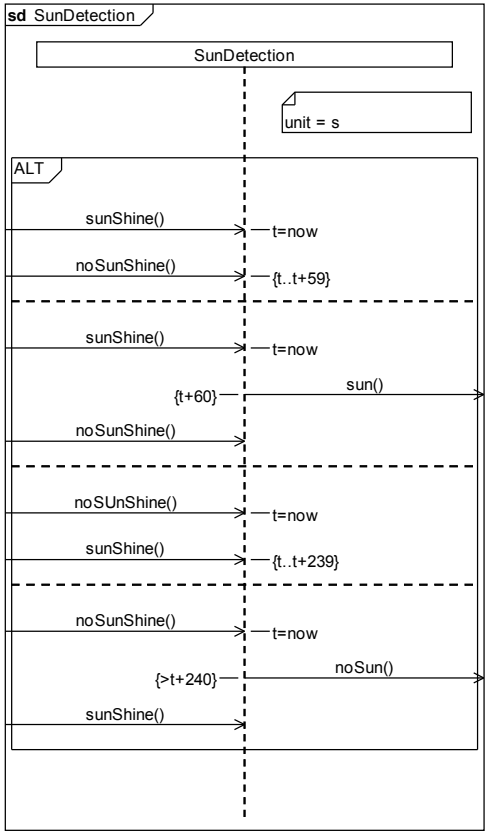


Behavior of the Components IX

ButtonHAL see *ButtonIAL* and description of Microcontroller
SunSensorHAL see *SunSensorIAL* and description of Microcontroller
WindSensorHAL see *WindSensorIAL* and description of Microcontroller
MotorHAL see *MotorIAL* and description of Microcontroller.
Microcontroller see description of Microcontroller.

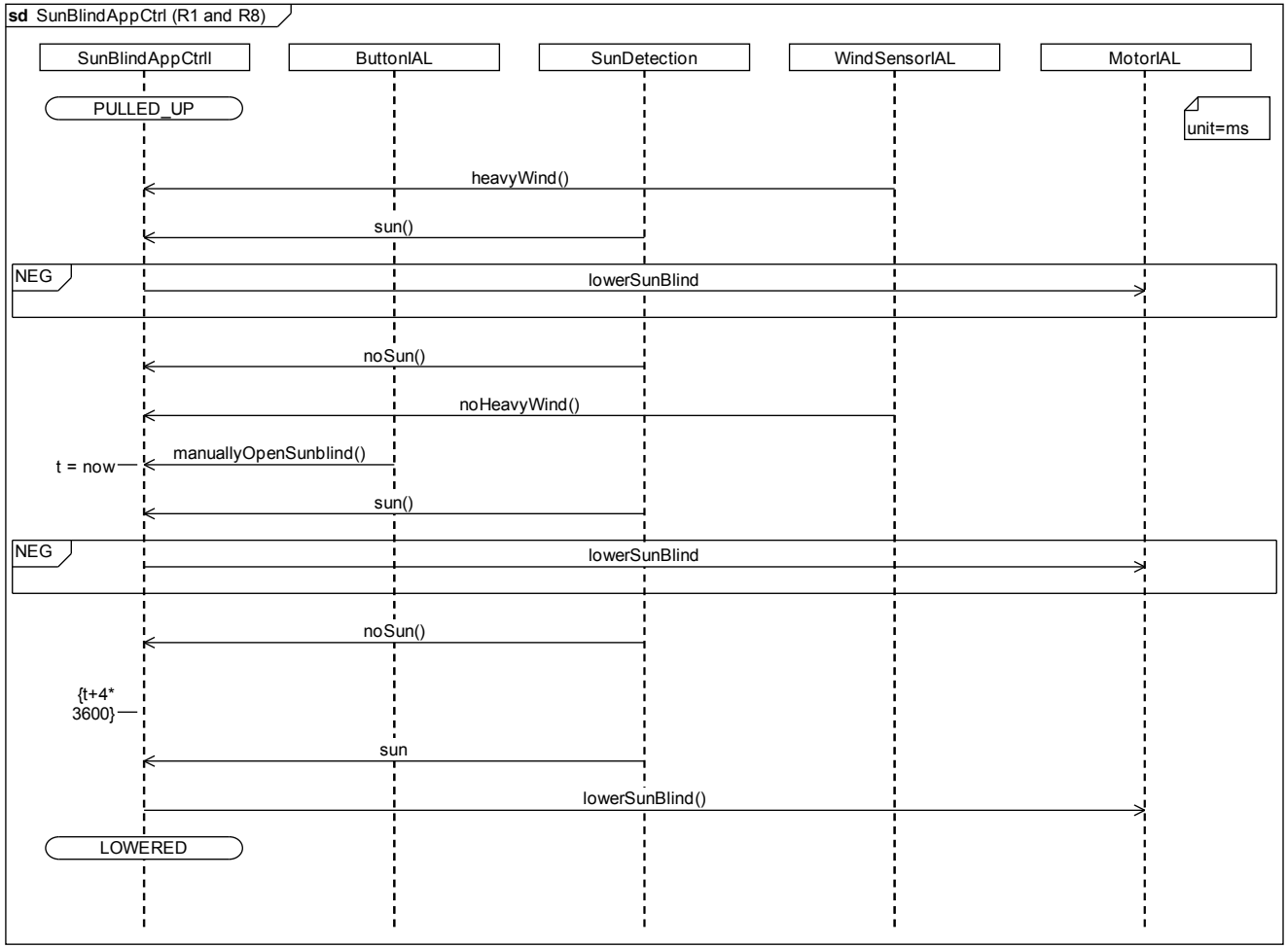
Behavior of the Components X

SunDetection – Subproblem SunControl

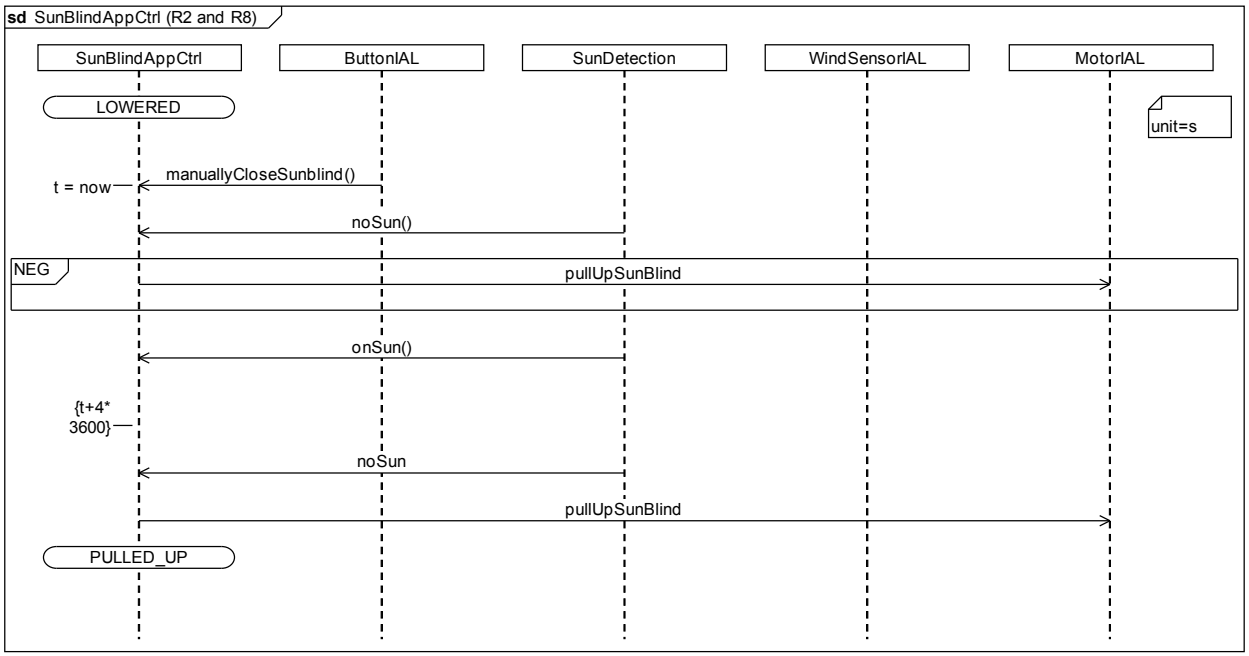


Behavior of the Components XI

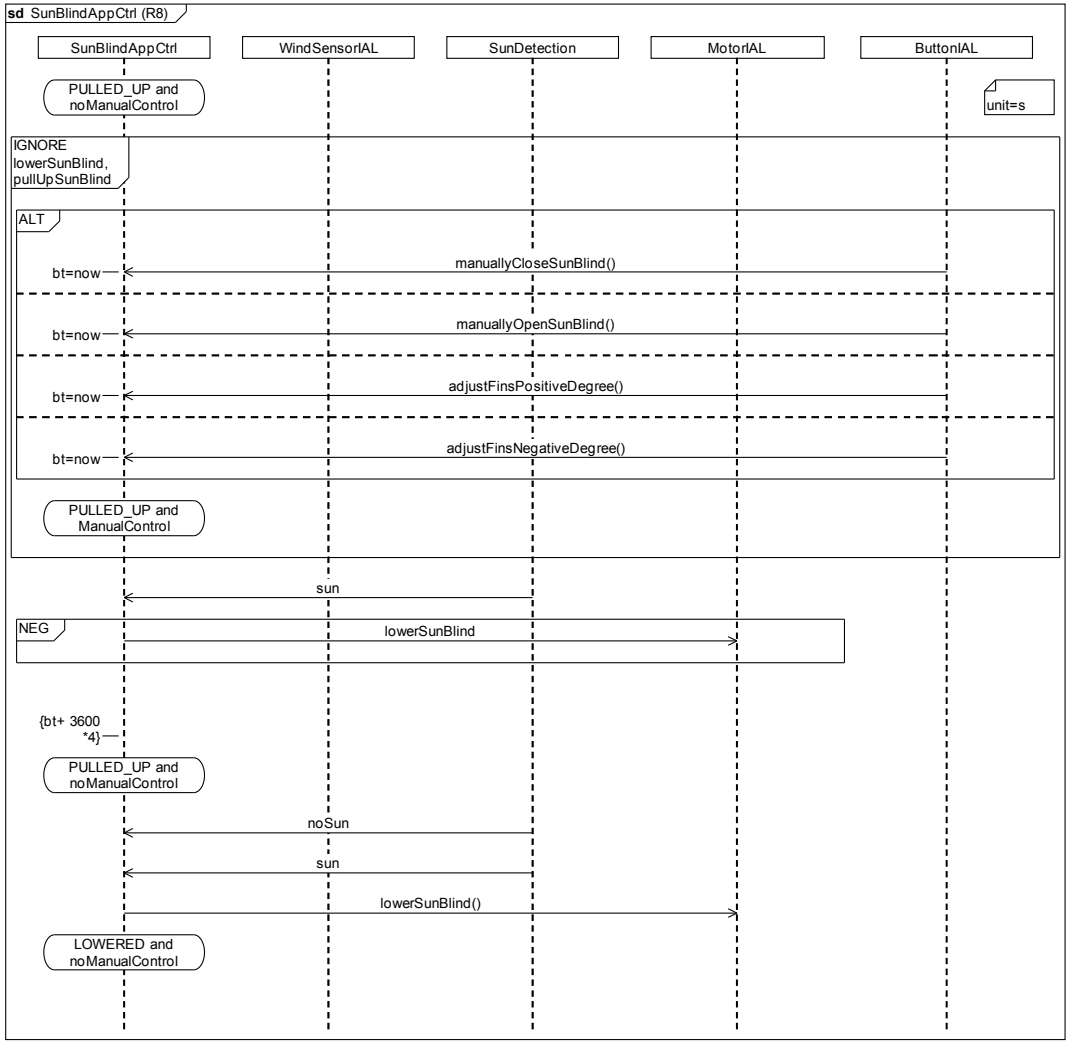
SunBlindAppCtrl – Subproblem SunControl



Behavior of the Components XII

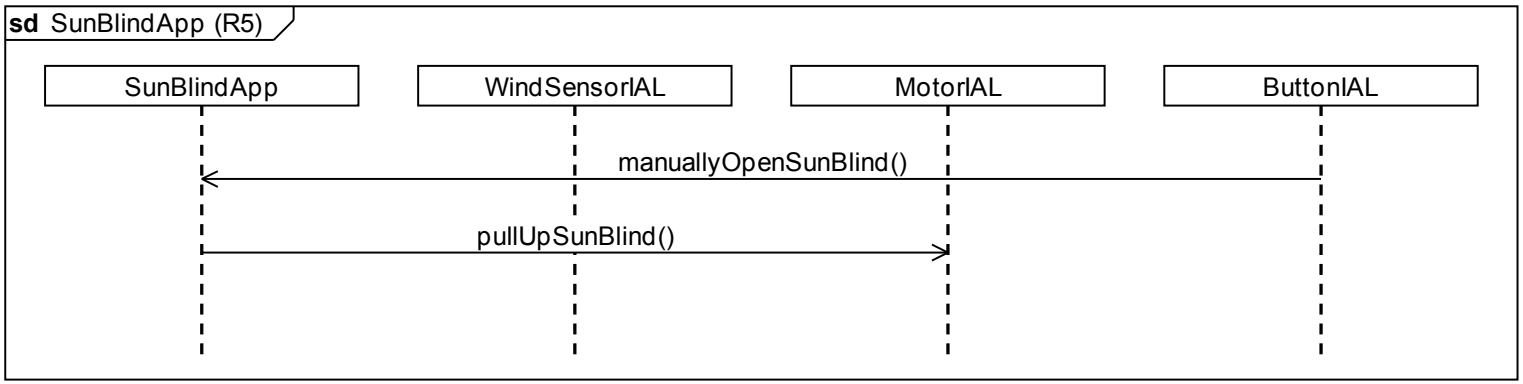


Behavior of the Components XIII

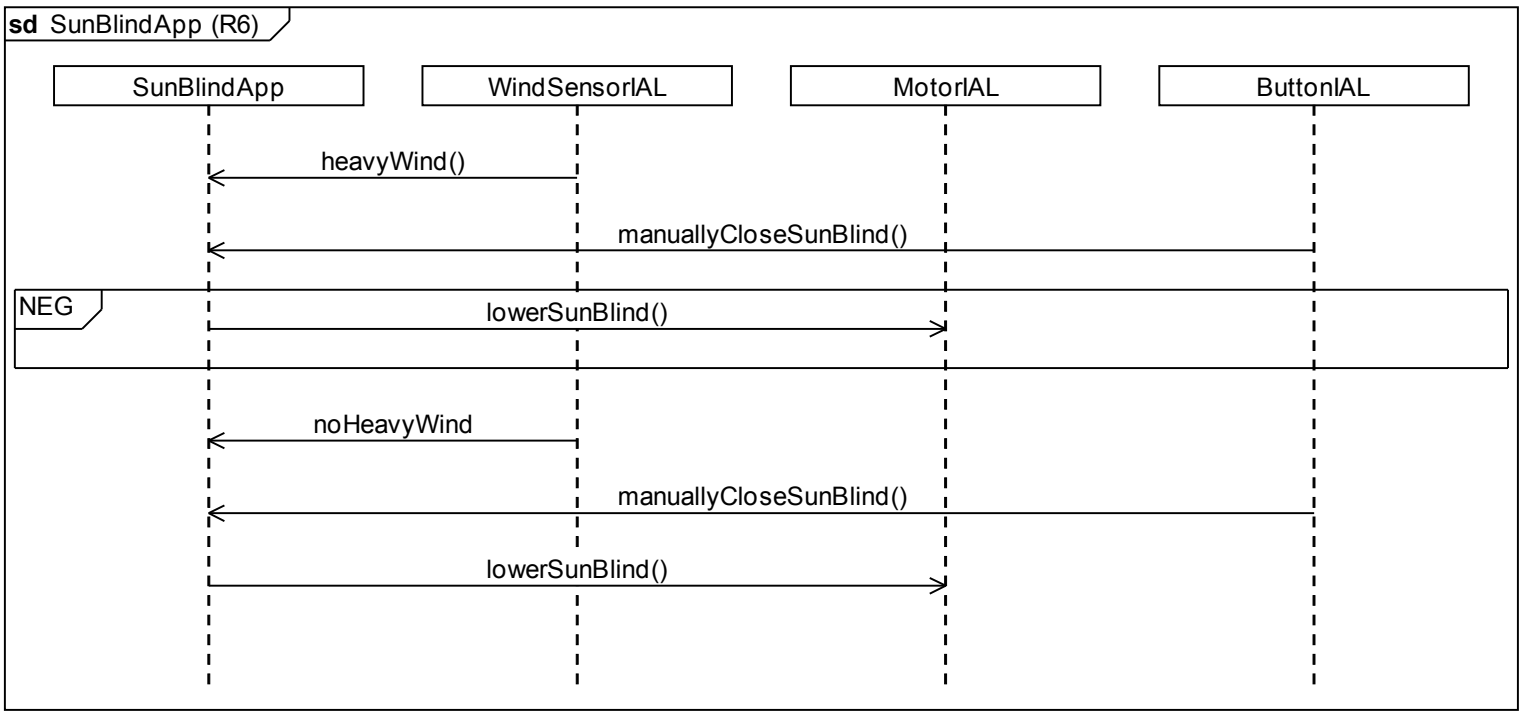


Behavior of the Components XIV

SunBlindAppCtrl – Subproblem UserControl

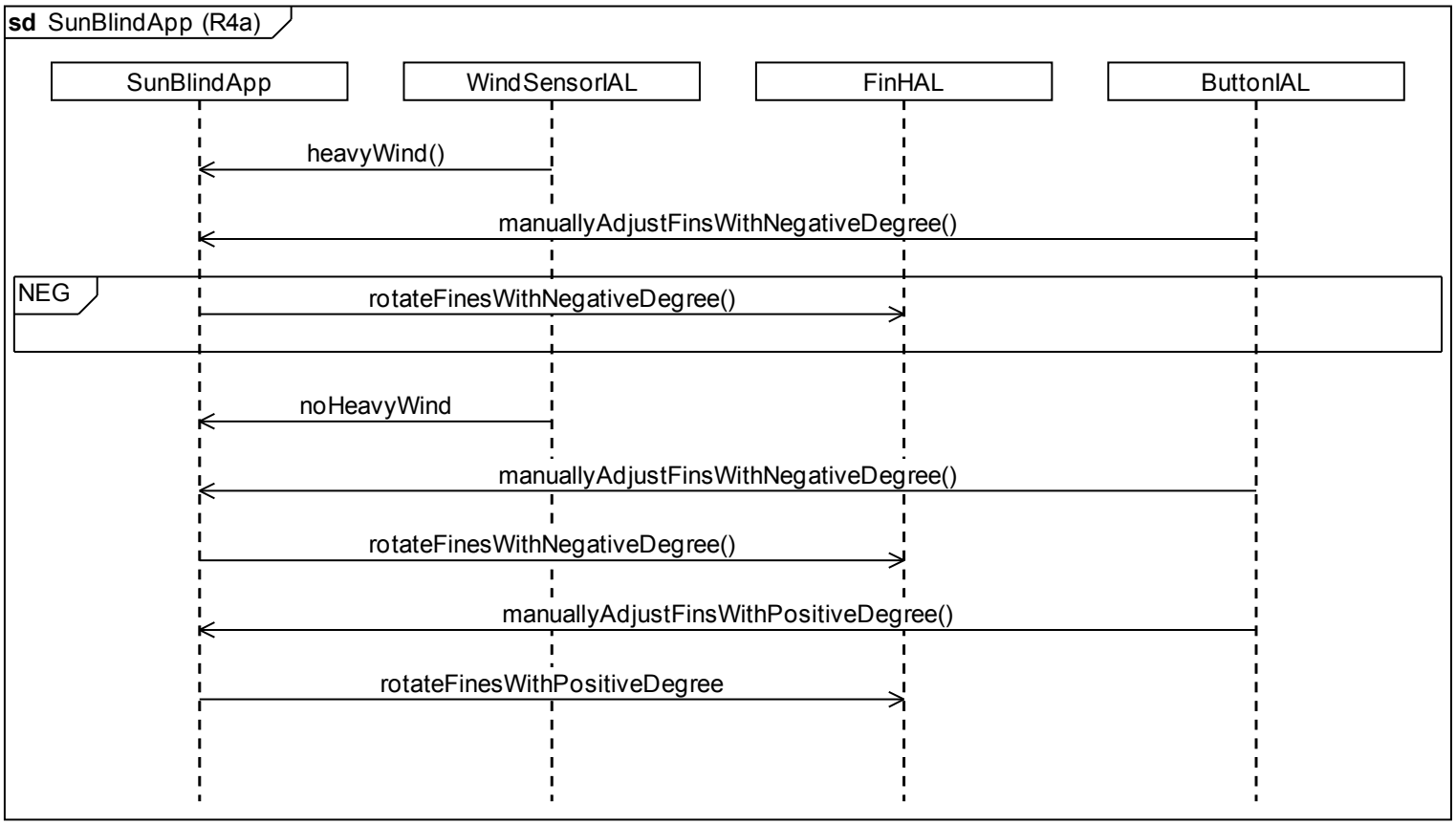


Behavior of the Components XV



Behavior of the Components XVI

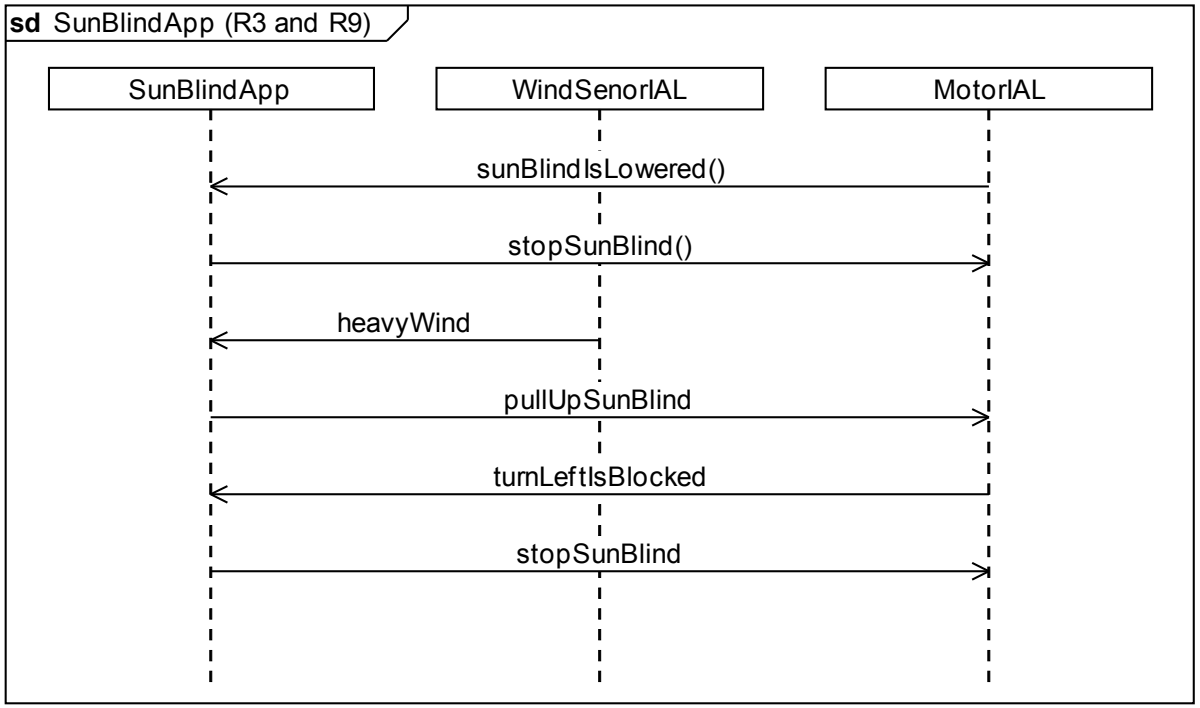
SunBlindApp – subproblem FinsControl



Sunl

Behavior of the Components XVII

– subproblem NodestructControl



Validation

- All sequence diagrams together describe the same behavior as in Phase 6.
- All signals in the interfaces classes of Phase 7 are captured in at least one sequence diagram.
- The direction of the signals are consistent with the required or provided interfaces of Phase 7.
- The signals connect the same components as connected in the software architecture of Phase 7.
- The state invariants can be mapped as follows:
  - *noManualControl* $\Leftrightarrow$ *no interaction within the last* 4 *hours*
  - *manualControl* $\Leftrightarrow$ *pressed within the last* 4 *hours*
  - *PULLEED_UP* $\Leftrightarrow$ *UP*
  - *LOWERED* $\Leftrightarrow$ *DOWN*

Phase 9: Specify the software components of all software architectures as state machines

...

7. Design a software architecture for all components of the global system architecture that should be implemented in software
8. Specify the behavior of all components of all software architectures, using sequence diagrams
9. Specify the software components of all software architectures as state machines
10. Implement software components and test environment
11. Integrate and test software components
12. Integrate and test hardware and software

Phase 9: Specify the software components of all software architectures as state machines

input:	interface behavior from Phase 8	sequence diagrams with annotated states
	relationships between subproblems specified in Phase 5	grammars
output:	component overview description with references to interface classes	class diagram with ports, sockets and lollipops
	data types and operations defined using pre- and postconditions	class diagrams formulas or natural language
	state machines	state machine diagrams
	invariants	formulas or natural language
validation:	consistent with interface behavior from Phase 8	
	completeness of state machines (implies error-cases for user-interaction)	
	a class must be active if it contains an active class or a timer	

Notations and concepts

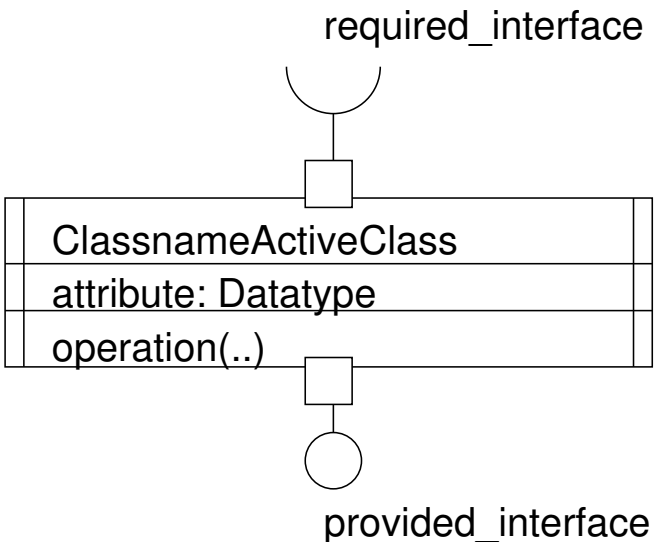
- Notation for active and passive classes
- Pre- and postconditions
- Class invariants
- UML 2.0 state machine diagrams
- Active and passive sensors

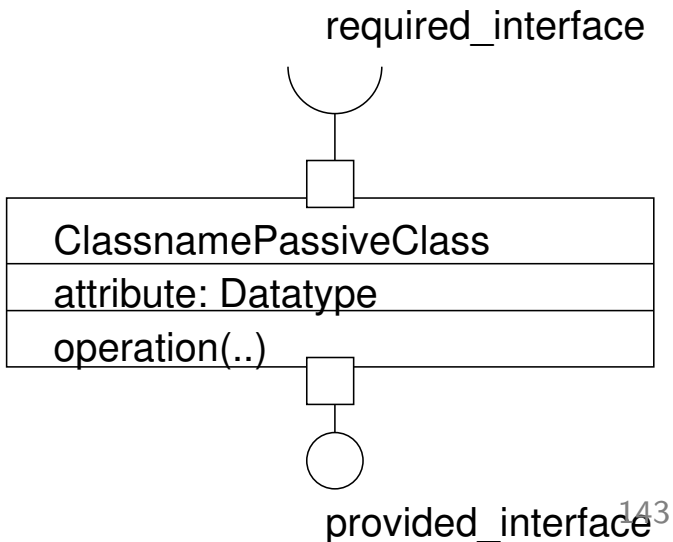
Notation for active and passive classes

Components corespond to classes.

Active Class: May contain timers, work in parallel to its environment, may contain other passive or active classes (see composite structure diagram of Phase 7)

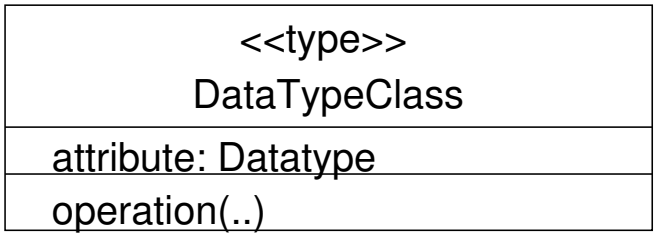
Passive Class: Cannot contain timers, functionality is executed in the time context of an active class, may contain other passive classes.





Notation for data types

Data Type Class: passive class without required or provided interfaces



Design by Contract – Pre- and Postconditions

- Needed to apply principle of design by contract.
- Must be specified for all operations.
- Preconditions: express when an operation may be called.
- Postconditions: express effect of operations by relating the state before calling the operation with the state that is reached after termination of the operation.

Design by Contract – Contracts in daily life

Bertrand Meyer
Object-Oriented Software Construction
Prentice Hall 1988 (first edition), 1997 (second edition)
online see:
http://archive.eiffel.com/doc/manuals/technology/contract/page.html

- Contractual partners are clients and sellers or service providers.
- Both expect advantages from the contract and are willing to make a commitment.

Design by Contract – Example

I want to travel from Berlin to Duisburg.

	Commitments	Advantages
Passenger	Pay ticket Be there at departure time Must keep precondition	getting to Duisburg Has advantages from post-condition
Traffic provider	Must take the passenger to Duisburg Must guarantee postcond.	Receives price for the ticket; does not have to take passengers who have not paid or did not arrive in time Can assume precondition

Pre- and Postconditions – Advantages of explicit contracts

Meyer:

> *A contract document protects both the client, by specifying how much should be done, and the supplier, by stating that the supplier is not liable for failing to carry out tasks outside of the specified scope.*

Application to software: A contract is a formal agreement between a machine / a class and its actors / clients. It specifies the rights and duties for both sides.

Pre- and Postconditions – Example: Stack (generic class)

class	Stack[T]
attribute	nb_elements: integer
method	empty(): Boolean
	full(): Boolean
	push(x: T)
	pop()
	top(): T
end class	Stack[T]

Pre- and Postconditions – Specification of the stack operations I

empty() **pre** true
post Result = true ⇔ nb_elements = 0

full() **pre** true
post Result = true ⇔ nb_elements = ...

push(x: T) **pre** not full
post not empty
and nb_elements = nb_elements@pre + 1
and "top = x"

"x@pre": old value of attribute x, i.e., value when operation is called;
"x": new value of attribute x, i.e., value when operation terminates

Pre- and Postconditions – Specification of the stack operations II

pop() **pre** not empty
post not full
and nb_elements = nb_elements@pre - 1
and "top element of the stack is deleted"

top(): T **pre** not empty
post noChange
and Result = "top element of the stack"

Pre- and Postconditions – Commitments and advantages

	Commitments	Advantages
Client	Call *push*(x) only if stack is not full Must keep precondition	Element x is put on stack, top() results in x, *nb_elements* is increased by 1. Has advantages from post-condition
Server	Makes sure that x is placed on the stack Must guarantee post-condition	Unnecessary to handle the case if stack is full. Can assume precondition

Pre- and Postconditions – Application of the design by contract principle I

- An operation may only be called when its precondition is satisfied. Otherwise, its effect is unspecified.
- The caller of an operation must make sure that the precondition of the operation holds.
  This can be done by checking the precondition explicitely before calling the operation. However, the precondition can also be guaranteed by the context. For example, immediately after a *pop* operation, a *push* operation is always possible.
- Pre- and postcondition must be contained in the code of each operation, either as an assertion that can be checked at runtime, or (at least) as a comment.
- In the implementation of the operation, the precondition is not checked!!

Pre- and Postconditions – Application of the design by contract principle II

- Operations that are called from the environment outside the machine should be robust, i.e., they should have precondition *true*, and the treatment of error cases should be performed in the operation.
- Internal operations may have stronger preconditions than *true*. Then, the treatment of error cases must be performed in the operations that call the operation in question.
- Example: An operation *push1* that checks if the stack is full is a different operation than the *push* operation given above. It implements a different functionality!

Pre- and Postconditions – Syntax

- Boolean expressions like in programming languages
  Example: `x <> 0 and x <= y`
- Formulas, using logical connectives
  Example: $x \neq 0 \wedge x \leq y$
- Natural language
  Example: "x must not be equal to zero, and x must be less than or equal to y"

Class invariants

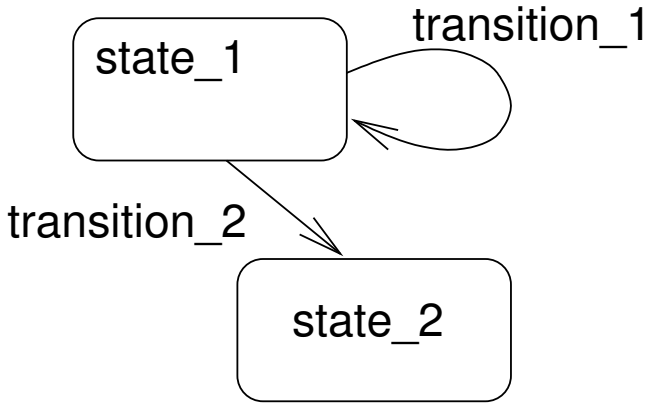
Condition that holds for all objects of a class, in the initial state, before and after each operation.

Example: class invariant for *Stack*: $0 \leq nb\_elements$

State Machines

Consist of states and transitions between these states.

Example: two states with transitions



Transitions, initial states

1

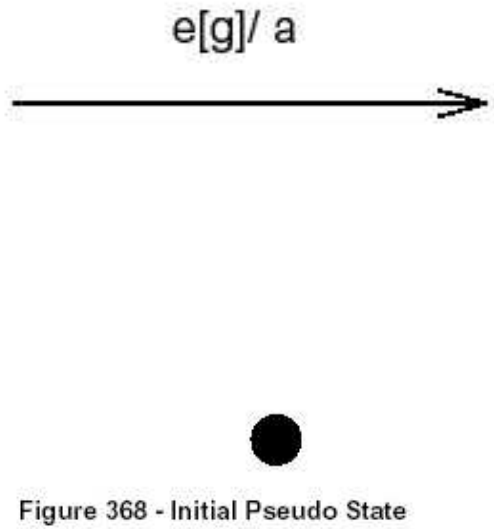


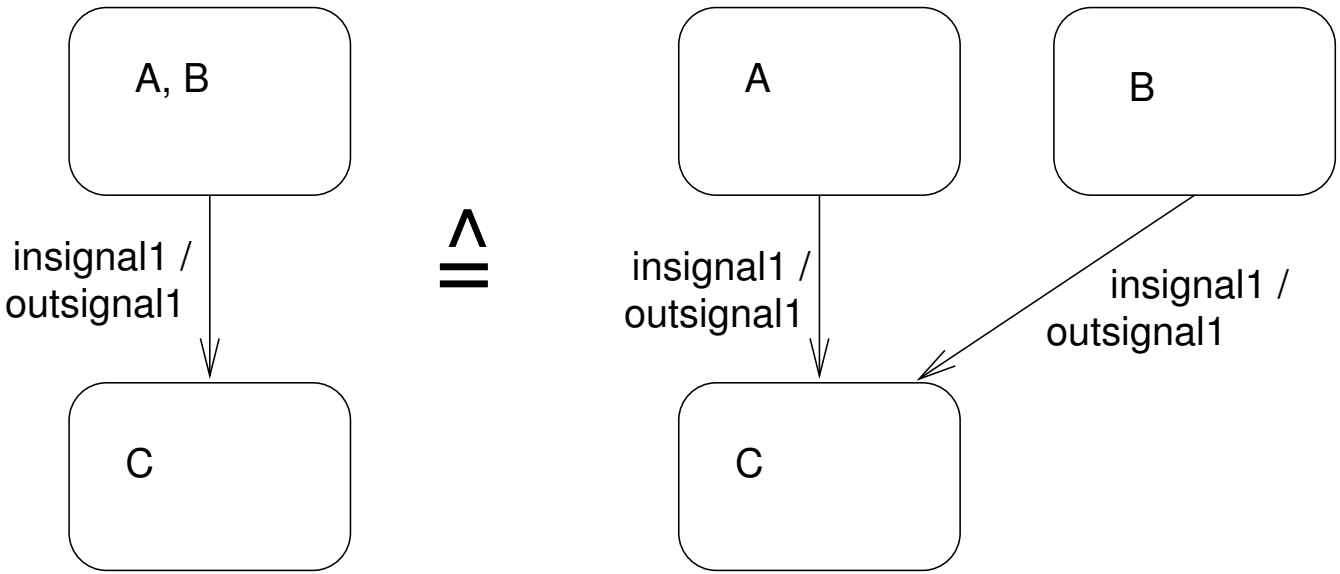
Figure 368 - Initial Pseudo State

- Transition:
  - e: input signal
  - g: guard (boolean expression)
  - a: output signals or actions, seperated with commas

  A transition is taken when g is true and e occurs. Then, a is sent/executed.
- Each state machine must have an initial (pseudo-) state that points to another state

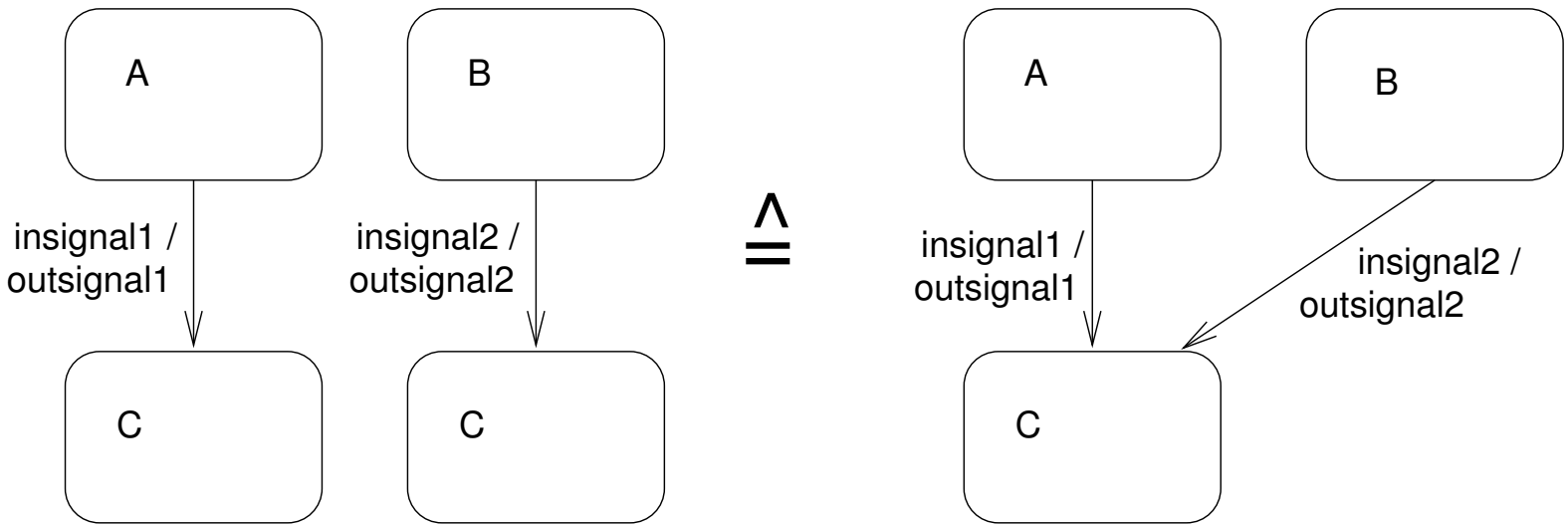
[1] Several figures taken from UML Superstructure Specification, v2.0 (709 Pages), http://www.omg.org/docs/formal/05-07-04.pdf

State Lists



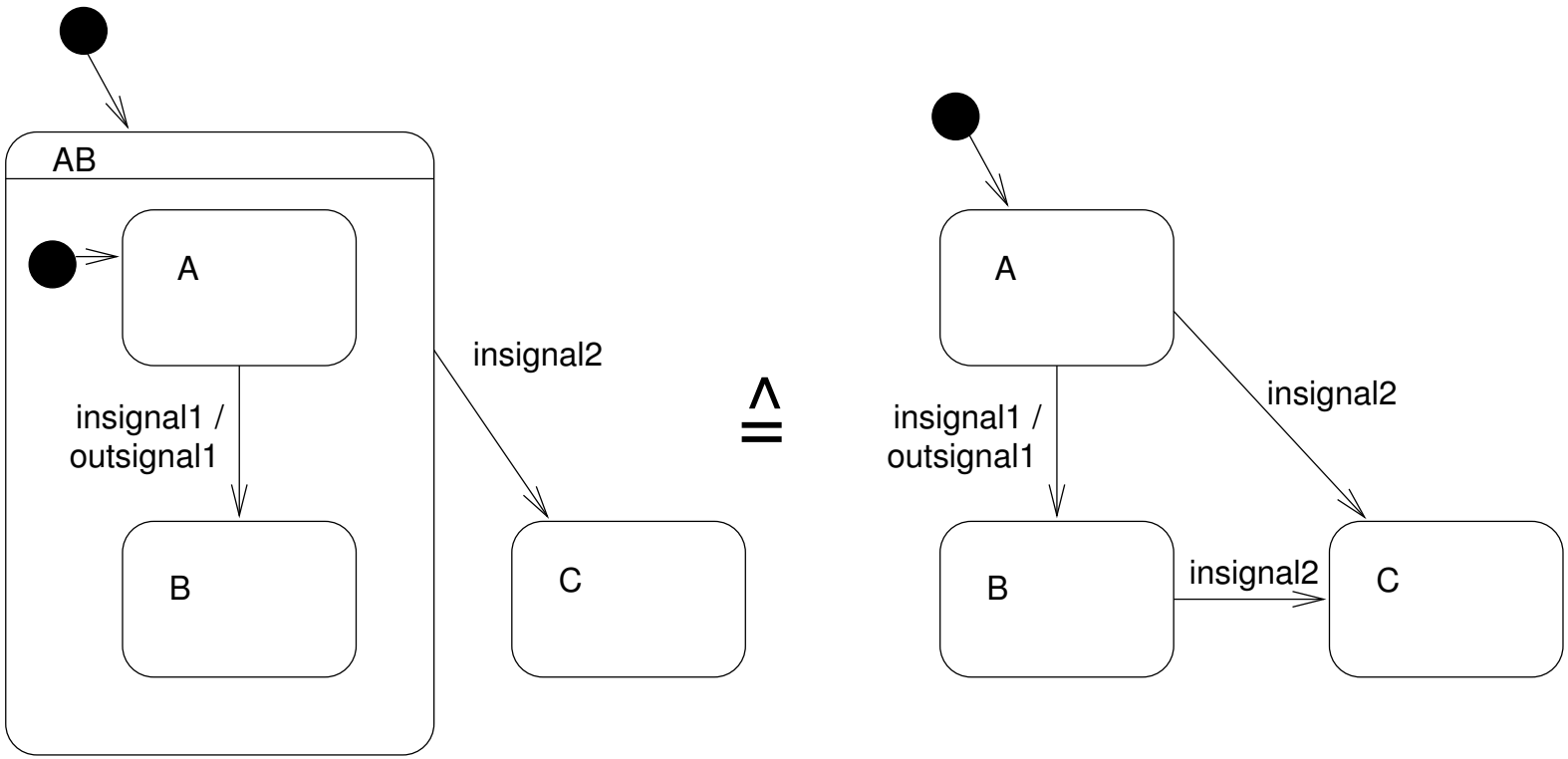
It is allowed to combine states.

State References



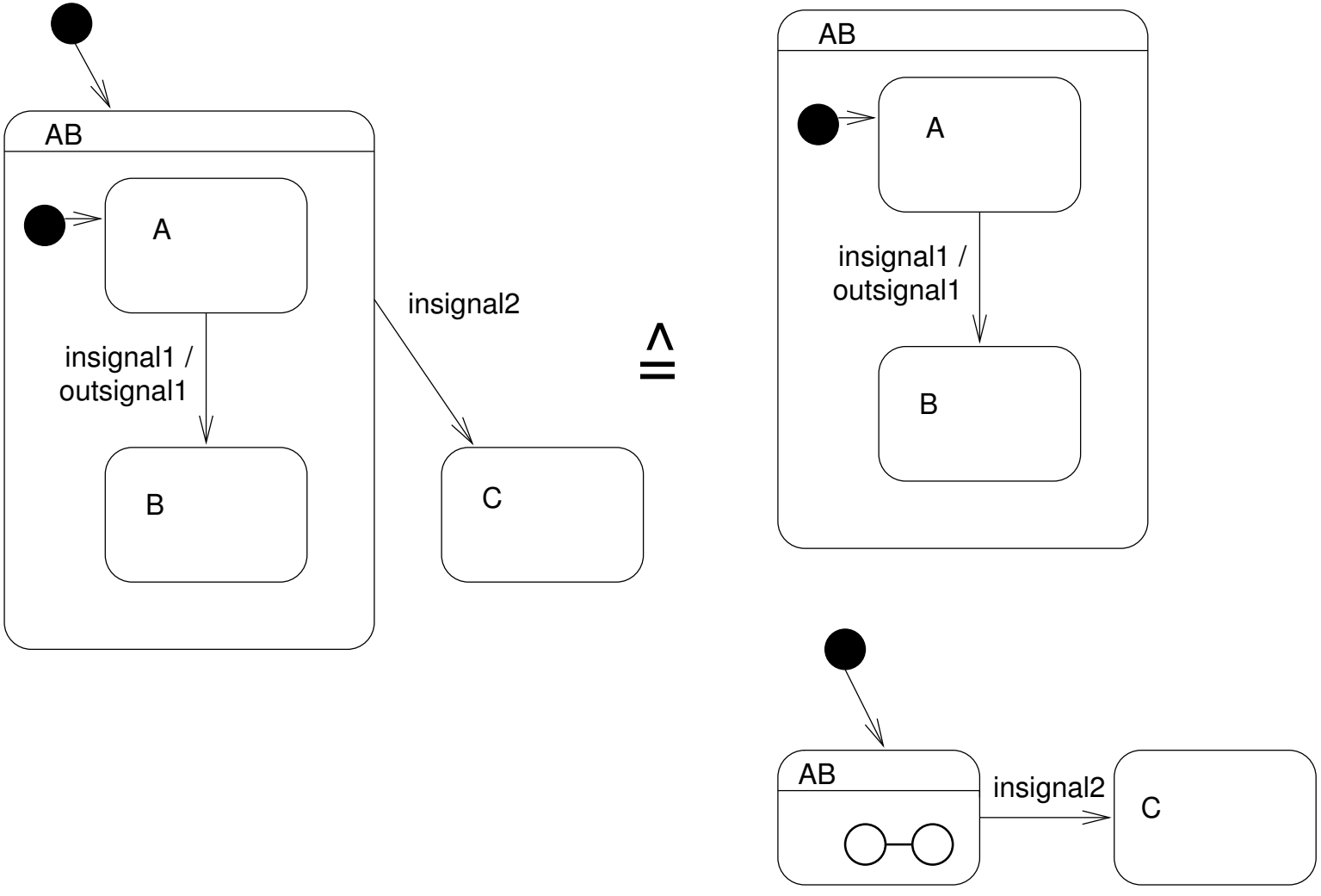
It is allowed to repeat states.

Composite States



"XOR" hierachical states. When the state machine is in state *AB*, it is *either* in state *A* *or* in state *B*.

Referenced Composite States



Regions

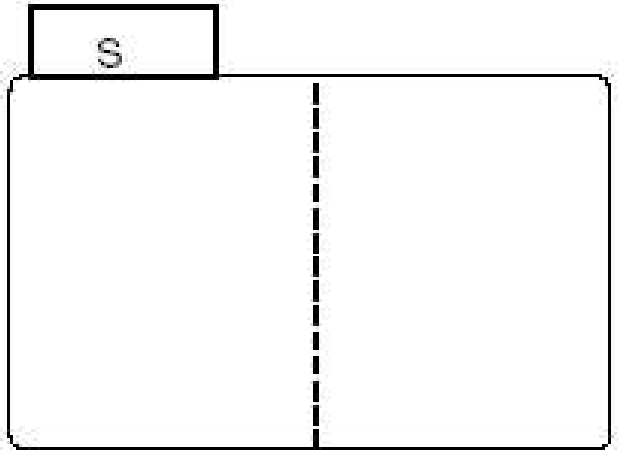


Figure 380 - Notation for composite state/state machine with regions

"AND" hierarchical (parallel) states. When the state machine is in state S, it is *in all* regions at the same time. To eliminate regions, one has to form the Cartesian product of the states of the (two or more) regions.
Alternative: Do not use regions, but define a separate active class (and corresponding state machine) for each parallel state machine.

Choice States

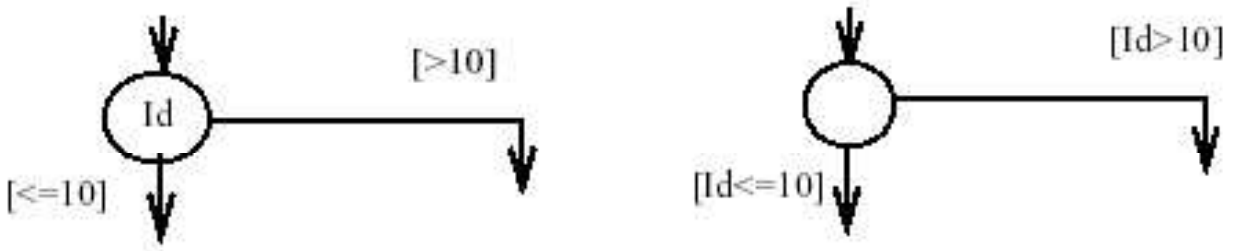
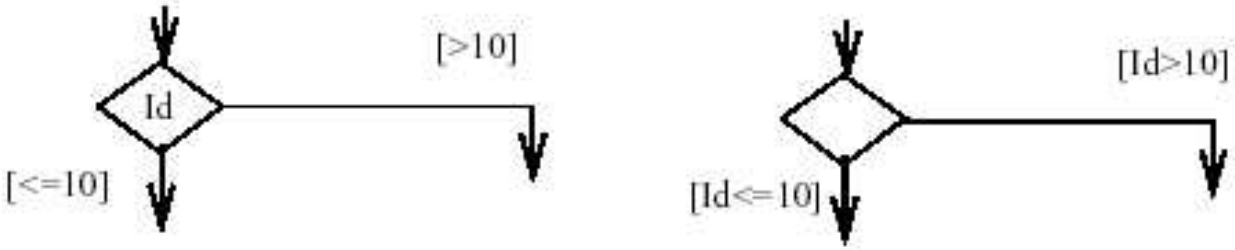


Figure 374 - Choice Pseudo State



No “real” states (the state machine cannot stay in such states, events are not processed); they are only used to make case distinctions.

Entry and Exit Points

Figure 371 - Entry point

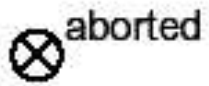


Figure 372 - Exit point

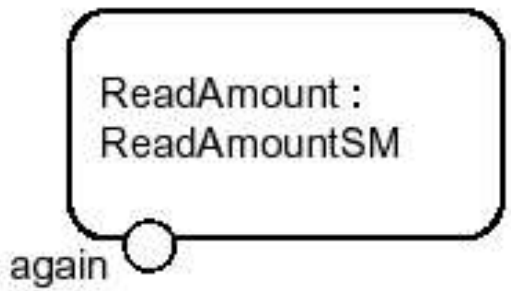


Figure 359 - Entry Point

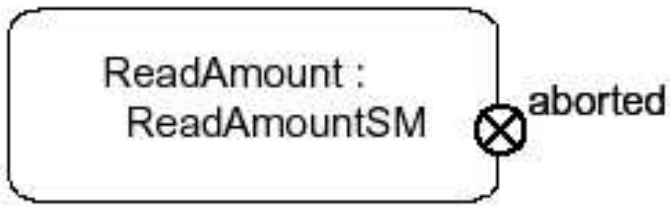
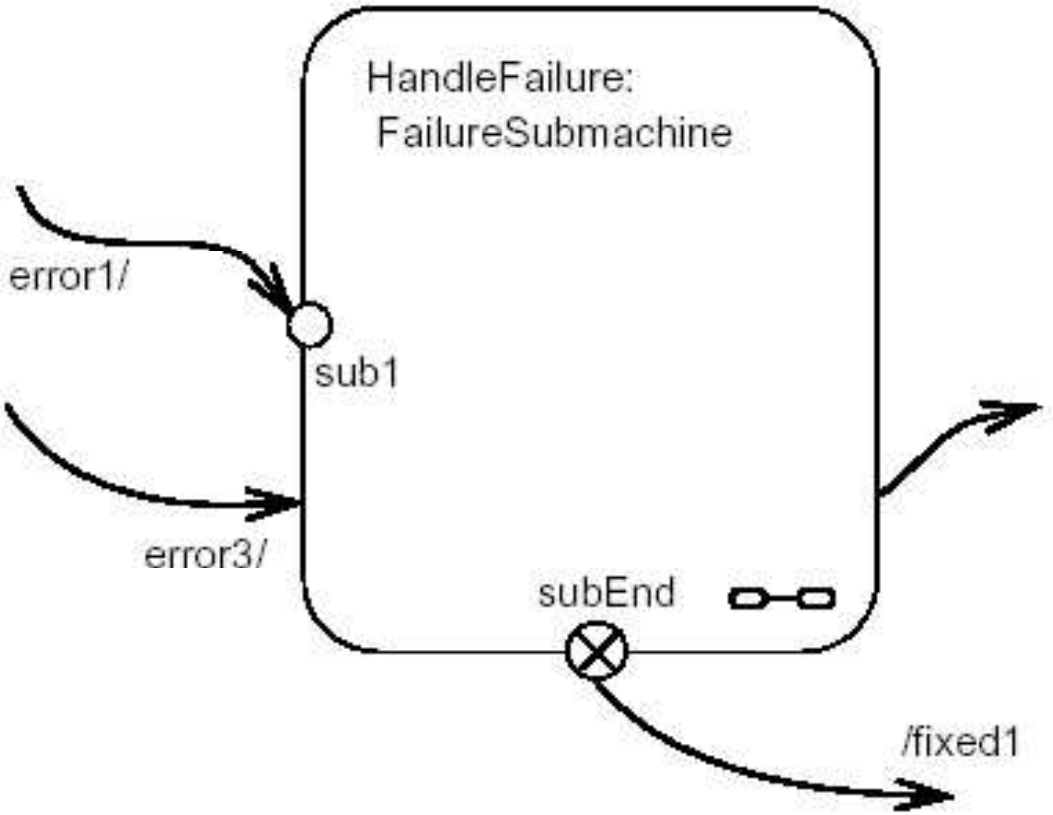


Figure 361 - Exit Point

Entry points can be used to go directly to a specified state inside a composite state.
Exit points can be used to leave the composite state from a specified state and go to another state outside (that is connected with the exit point).

Example: Reference to Sub-State Machines with Entry and Exit Points



Final State and Terminate Node

Figure 363 - Final State

A final state indicates, that the composite state will be left.

Figure 375 - Terminate node

A terminate node indicates that the object will be destroyed.

Sub-State Machines: Example ATM (1)

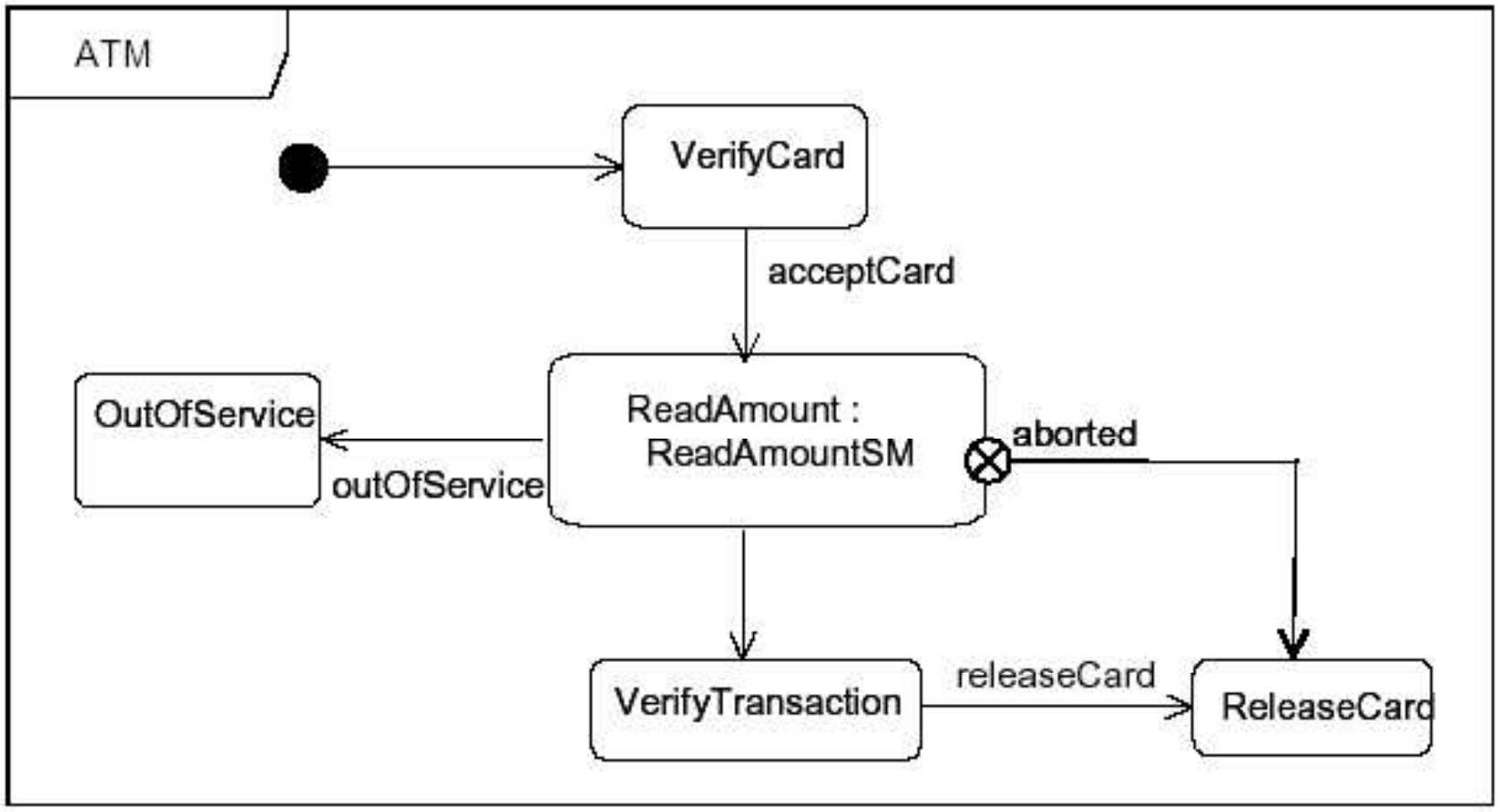


Figure 390 - SubmachineState with usage of exit point

Sub-State Machines: Example ATM (2)

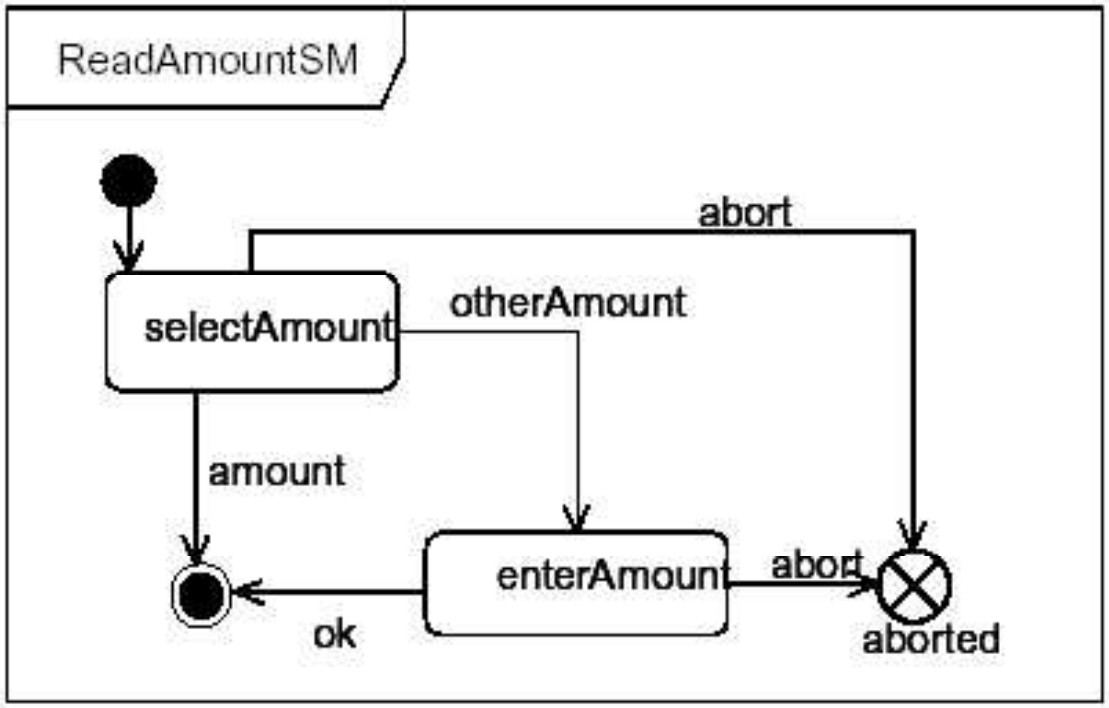


Figure 388 - State machine with exit point as part of the state graph

History

Figure 369 - Shallow History

When a composite state is re-entered, the sub-state is entered that was most recently left.

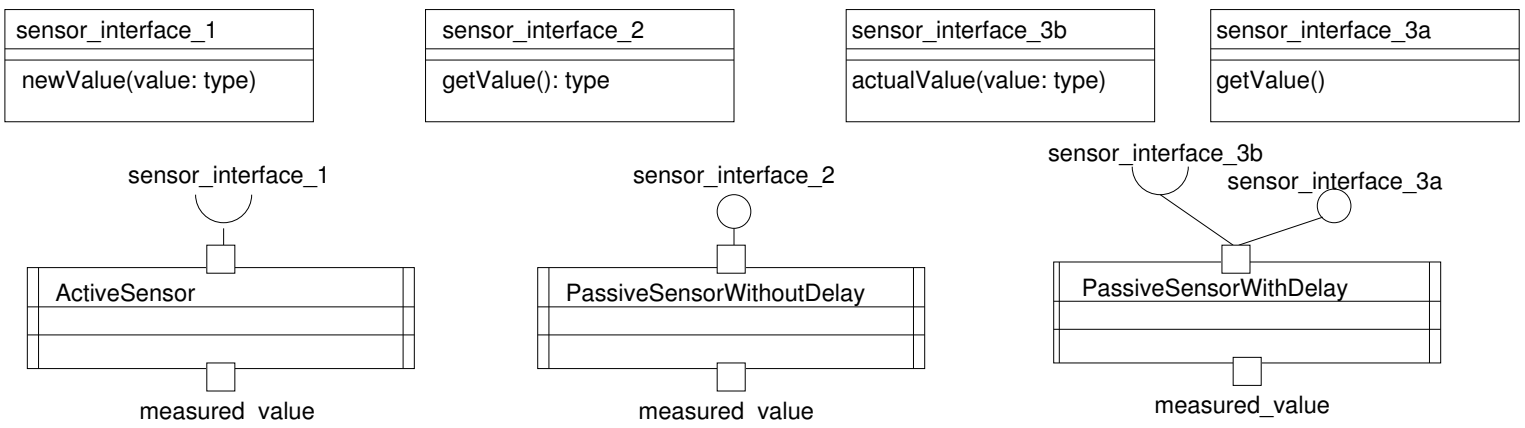
Figure 370 - Deep History

Recursive history connector. May be useful when substates are also hierarchically structured.

Difference between active and passive sensors I

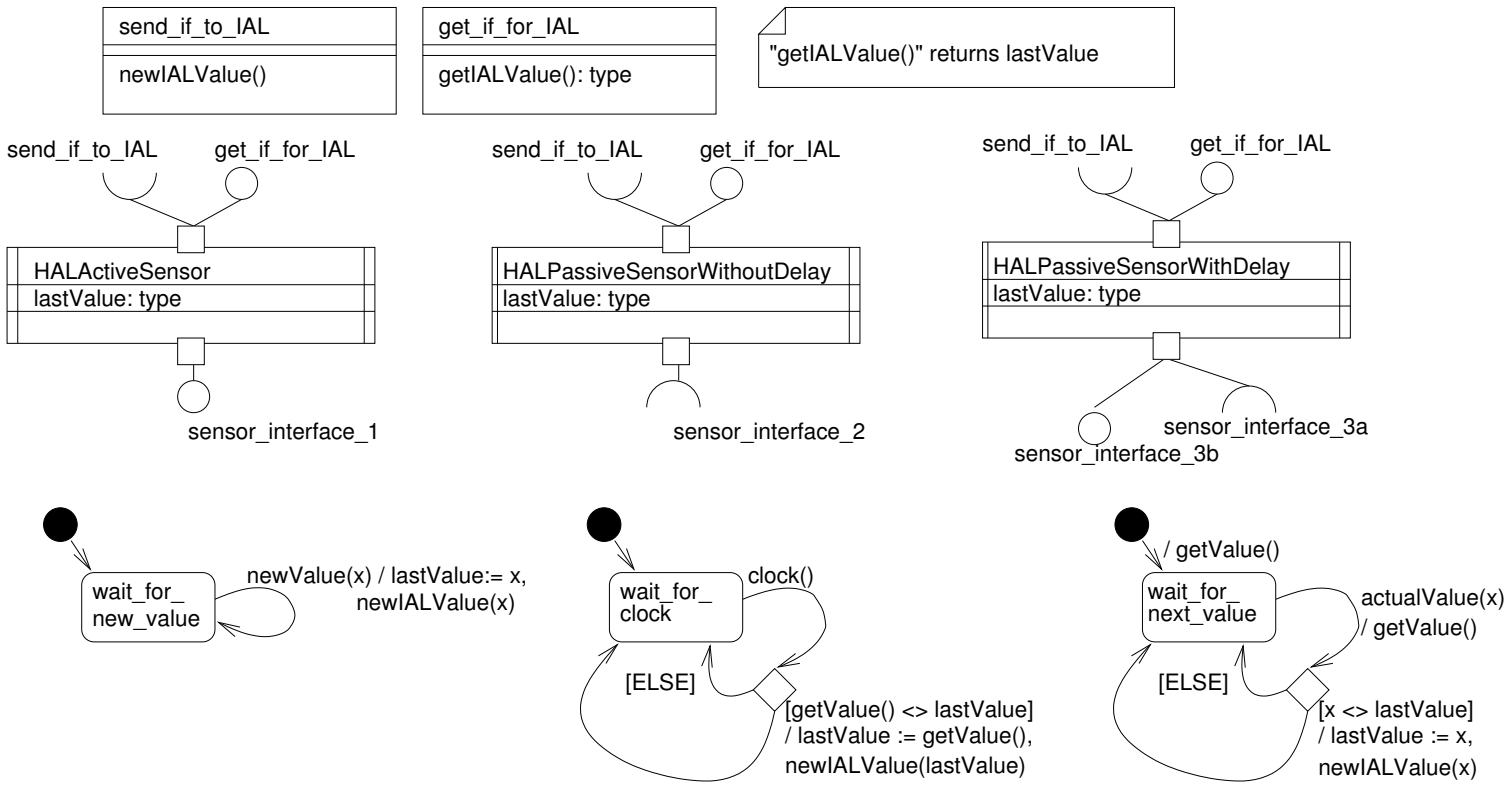
Active Sensor: Sensor actively sends its measured values (cyclic or when they changed).

Passive Sensor: Sensor values have to be requested. We distinguish between passive sensors that react with and without delay.



Both are drawn as *active classes* because they are hardware components, and hardware components usually work in parallel with their environment.

Difference between active and passive sensors II



The interface to the higher layer can be the same for all type of sensors if this kind of HAL is used.

Phase 9: Specify the software components of all software architectures as state machines

input:	interface behavior from Phase 8	sequence diagrams with annotated states
	relationships between subproblems specified in Phase 5	grammars
output:	component overview description with references to interface classes	class diagram with ports, sockets and lollipops
	data types and operations defined using pre- and postconditions	class diagrams formulas or natural language
	state machines	state machine diagrams
	invariants	formulas or natural language
validation:	consistent with interface behavior from Phase 8	
	completeness of state machines (implies error-cases for user-interaction)	
	a class must be active if it contains an active class or a timer	

Executing Phase 9 I

For each component, the following steps should be performed:

- Decompose the component if necessary. In this case, add descriptions of new interface classes.
- Draw an active (e.g., behaves like hardware, contains a clock) or passive (e.g., calculation-routine) class with its interfaces as a component overview description.

Executing Phase 9 II

For each subproblem and each component, the following steps should be performed:

- Design a state machines that implements the behavior of all sequence diagrams specified in Phase 8.
- Add necessary data as attributes to the component overview description.
- In case of complex data or complex operations on data types: add classes for data types.
- Specify pre- and postconditions for all operations of introduced classes.
- Complete the state machines, i.e. there must be a specified reaction to each possible input signal.
- Add class invariants to introduced classes if possible.

Executing Phase 9 III

To merge the different state machine associated with one component, the following steps should be performed:

- The state machine diagrams can be merged according to the case distinction we made in Phase 7:
  - Case 1 (The components are hardware (HAL) or interface abstraction layers (IAL), establishing the connection to some hardware device): often the state machines will already be equal, because they describe the same device. If not, the state machines must be merged manually. In many cases, we only need to add the additional signals to the appropriate states.
  - Case 2 (Two application components belong to subproblems being related sequentially or by alternative): the composition can be achieved by using composite states. The connecting arcs between the sub-automata depend on the problem.

Executing Phase 9 IV

- Case 3 (Two application components belong to parallel subproblems and share some output phenomena): here, the merge depends on the problem to be solved. The priorities from Phase 5 have to be taken into account.
- Case 4 (Two application components belong to parallel subproblems and share some input phenomena): the merge has to be performed manually.
- Case 5 (Two application components belong to parallel subproblems and do not share any interface phenomena): no merge should be performed, see Phase 7.

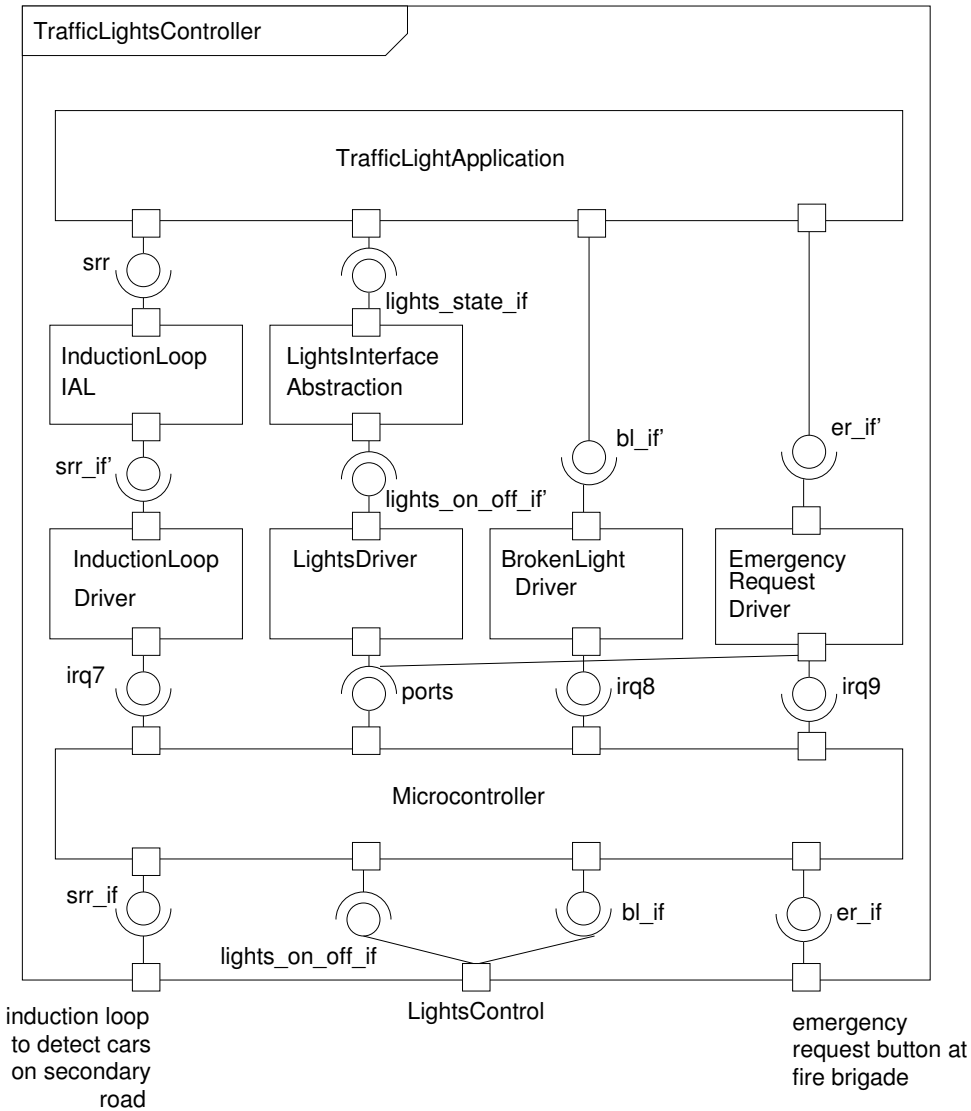
- When state machines are merged, for each state it must be checked if it can handle all events that can occur.

Remarks

- In contrast to Phase 8, the behavior must be described completely for each subproblem since the state diagrams form the basis for the implemention.
- To validate the results, it should be assured that each composed state machine is complete and covers all input events that can be sent by the components with an interface to the composed state machine.

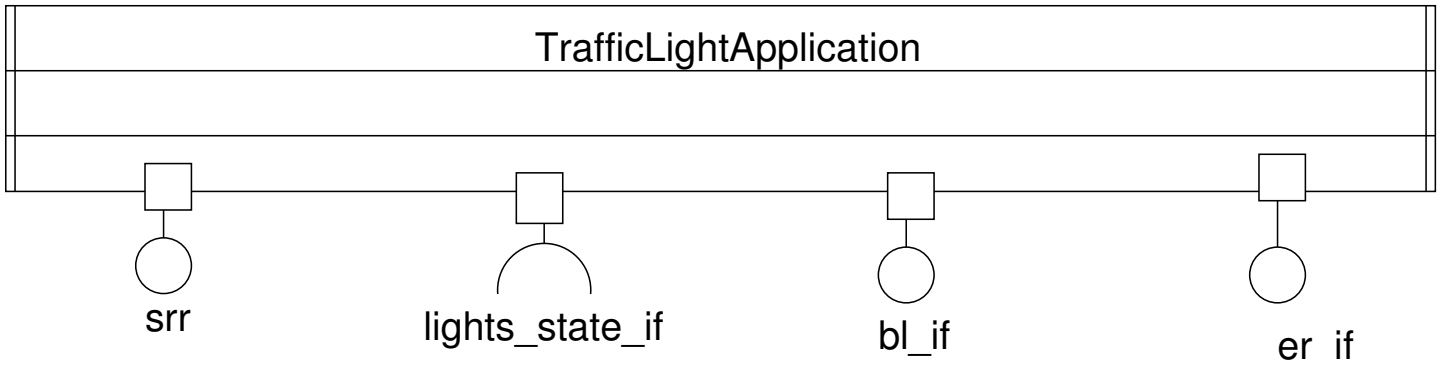
Example 1: traffic light control

Global software architecture



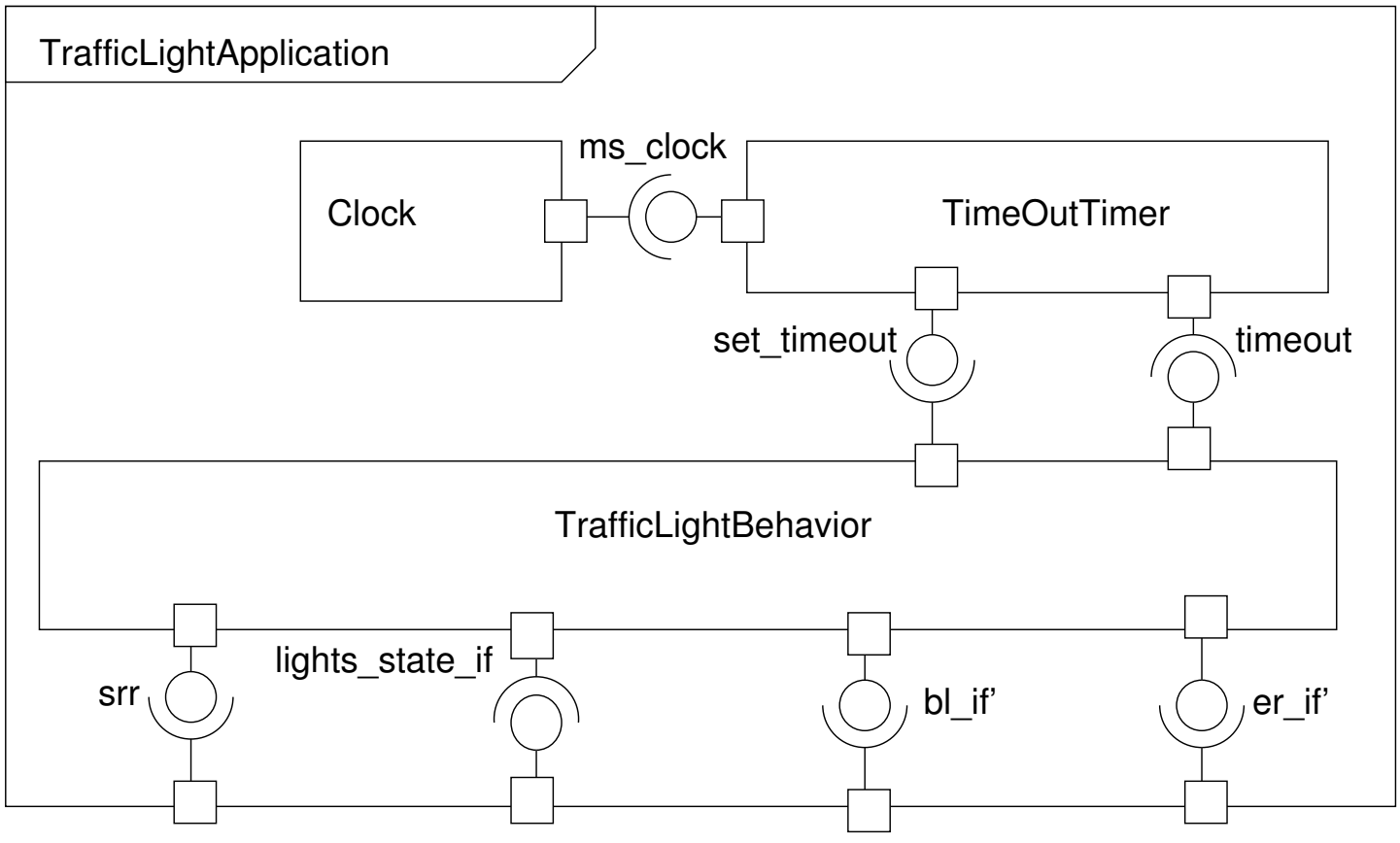
Component TrafficLightApplication I

The component *TrafficLightApplication* is an active component since it contains a clock. The following overview diagram is not strictly necessary since this component is decomposed as shown on the following slide.

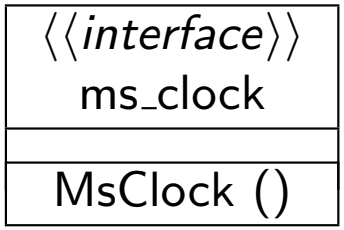


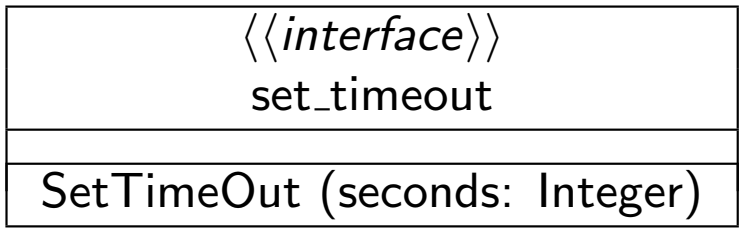
The component *TrafficLightApplication* is split into the subcomponents *TrafficLightBehavior*, *Clock*, and *TimeOutTimer*. These components are specified separately.

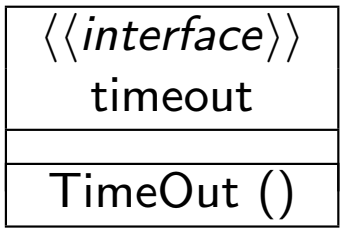
Component TrafficLightApplication II



Component TrafficLightApplication III

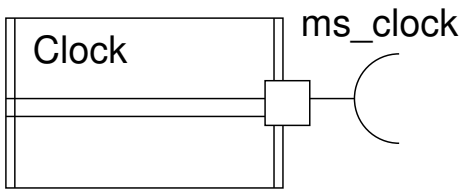






Component Clock

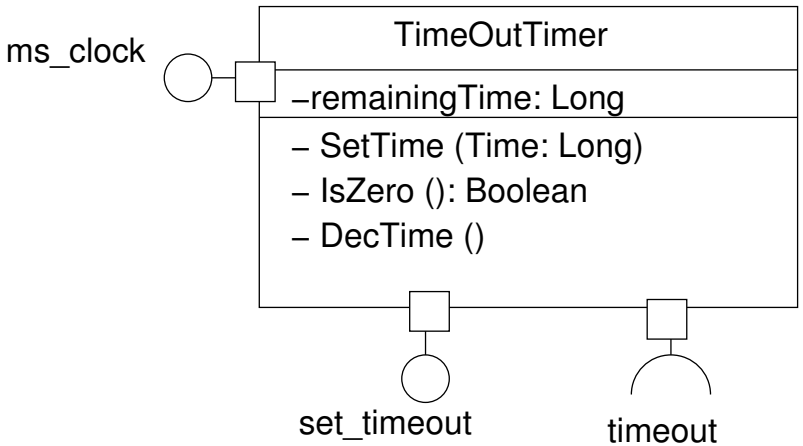
Clock component overview



- The component is an active component since it has to work in parallel with all other components and generates cyclic signals.
- Usually it is a standard component, contained in the operating system. Hence, it is not specified here.

Component TimeOutTimer I

TimeOutTimer component overview



- Additionally, it contains a data type and operators for the remaining time.
- The state machine uses this data.
- The component is a passive component, since it reacts immediately to the input signals of the clock.

Component TimeOutTimer II

TimeOutTimer operations

IsZero()

pre $true$

post $Result = true \Leftrightarrow remaining\_time = 0$

SetTime(x)

pre $x > 0$

post $remaining\_time = x$

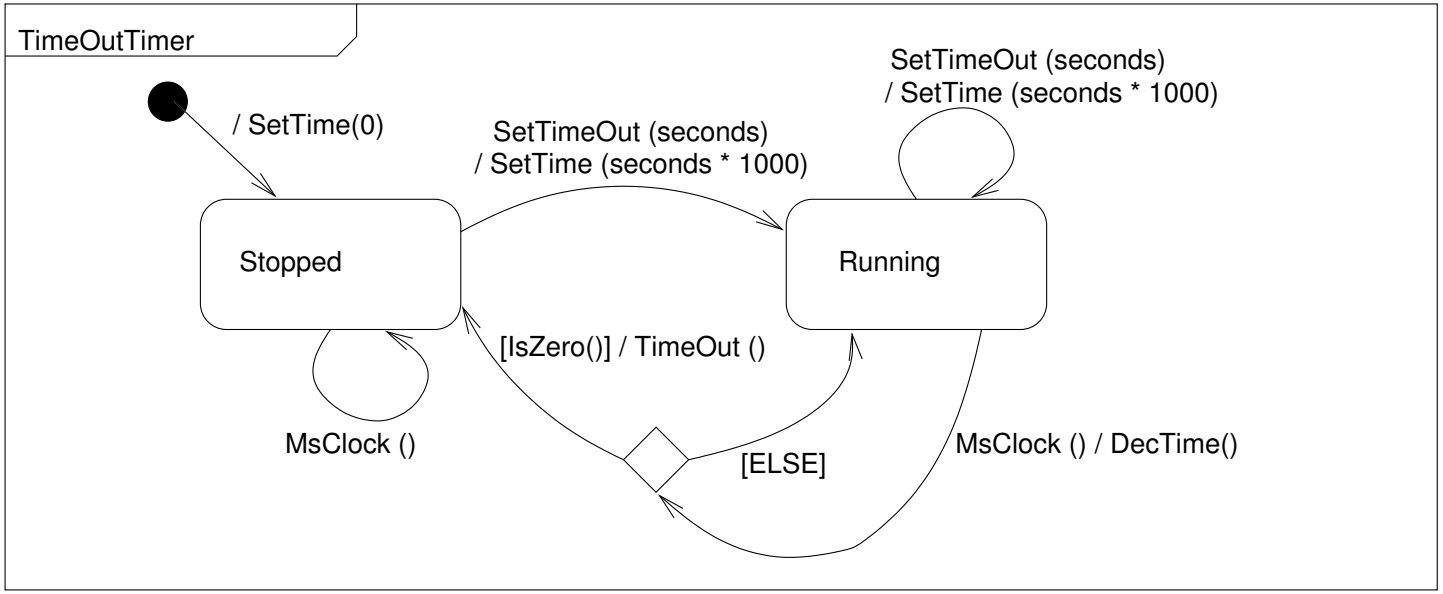
DecTime()

pre $remaining\_time \neq 0$

post $remaining\_time = remaining\_time@pre - 1$

Component TimeOutTimer III

TimeOutTimer state machine



TimeOutTimer invariants

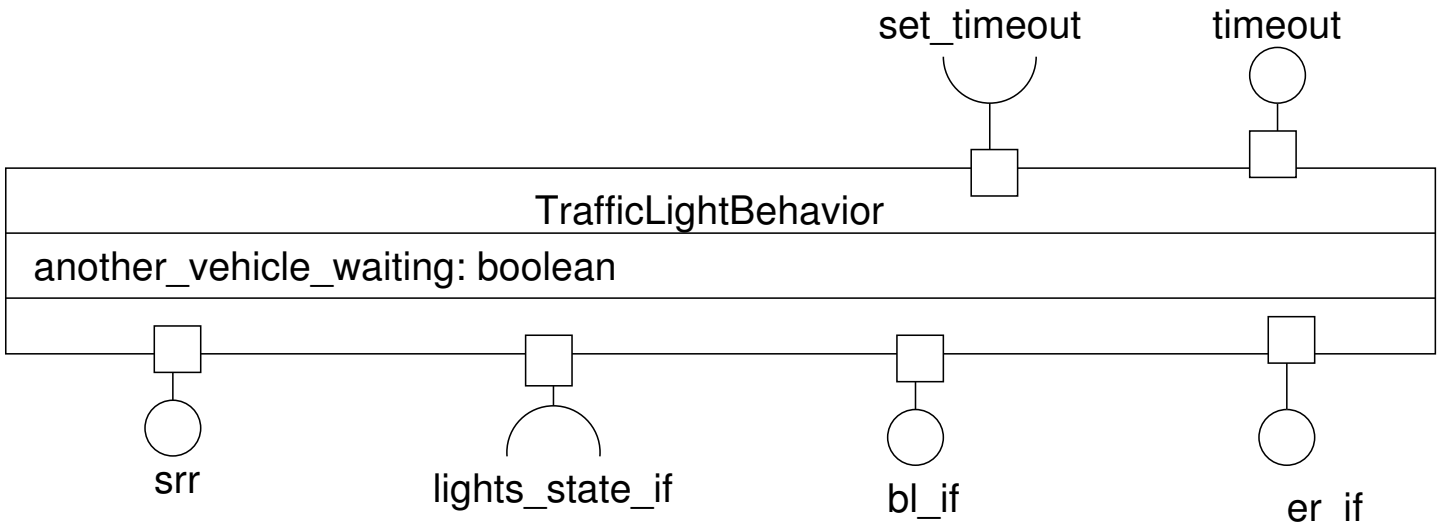
For the state machine and the data of the component the following additional invariant must always be true:

$$\textit{In state Stopped} \Rightarrow \textit{remaining\_time} = 0$$

$$\textit{In state Running} \Rightarrow \textit{remaining\_time} > 0$$

Component TrafficLightBehavior I

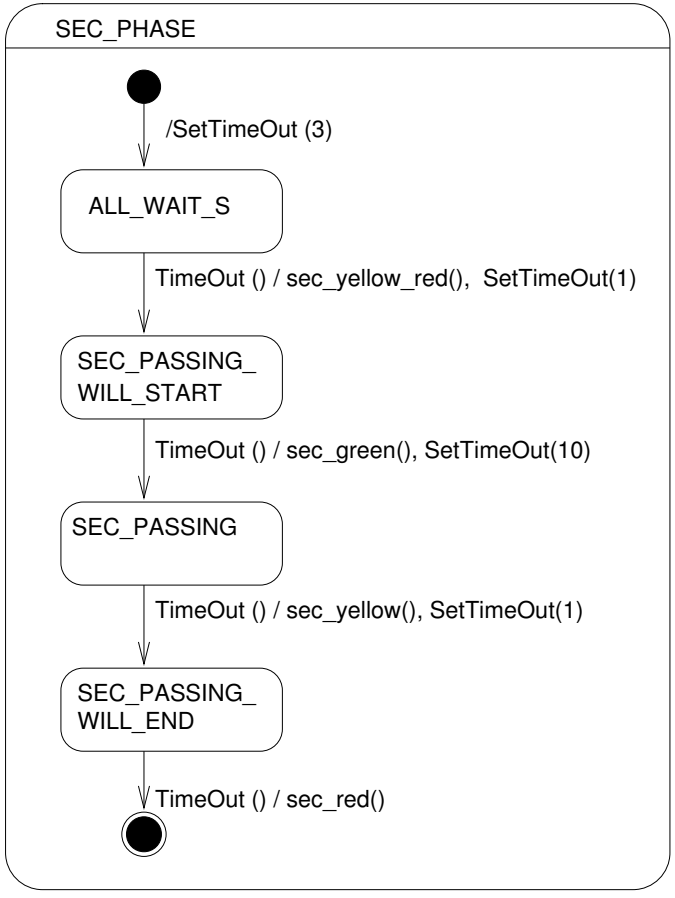
TrafficLightBehavior component overview



- It is a passive component since it reacts immediately to input signals.
- The attribute *another_vehicle_waiting* was added later.

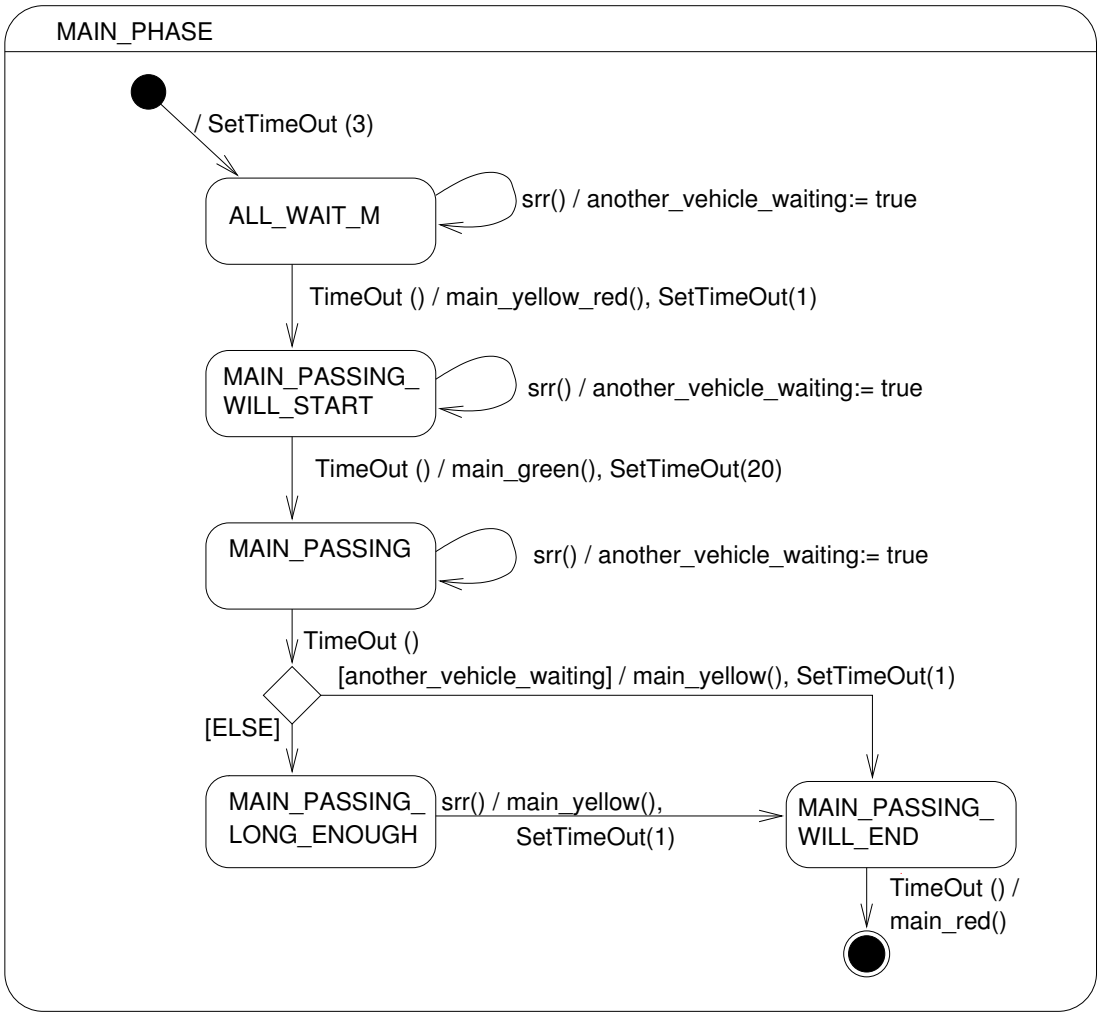
Component TrafficLightBehavior II

TrafficLightBehavior state machine for SecondaryRoadPassing



Component TrafficLightBehavior III

TrafficLightBehavior state machine for MainRoadPassing

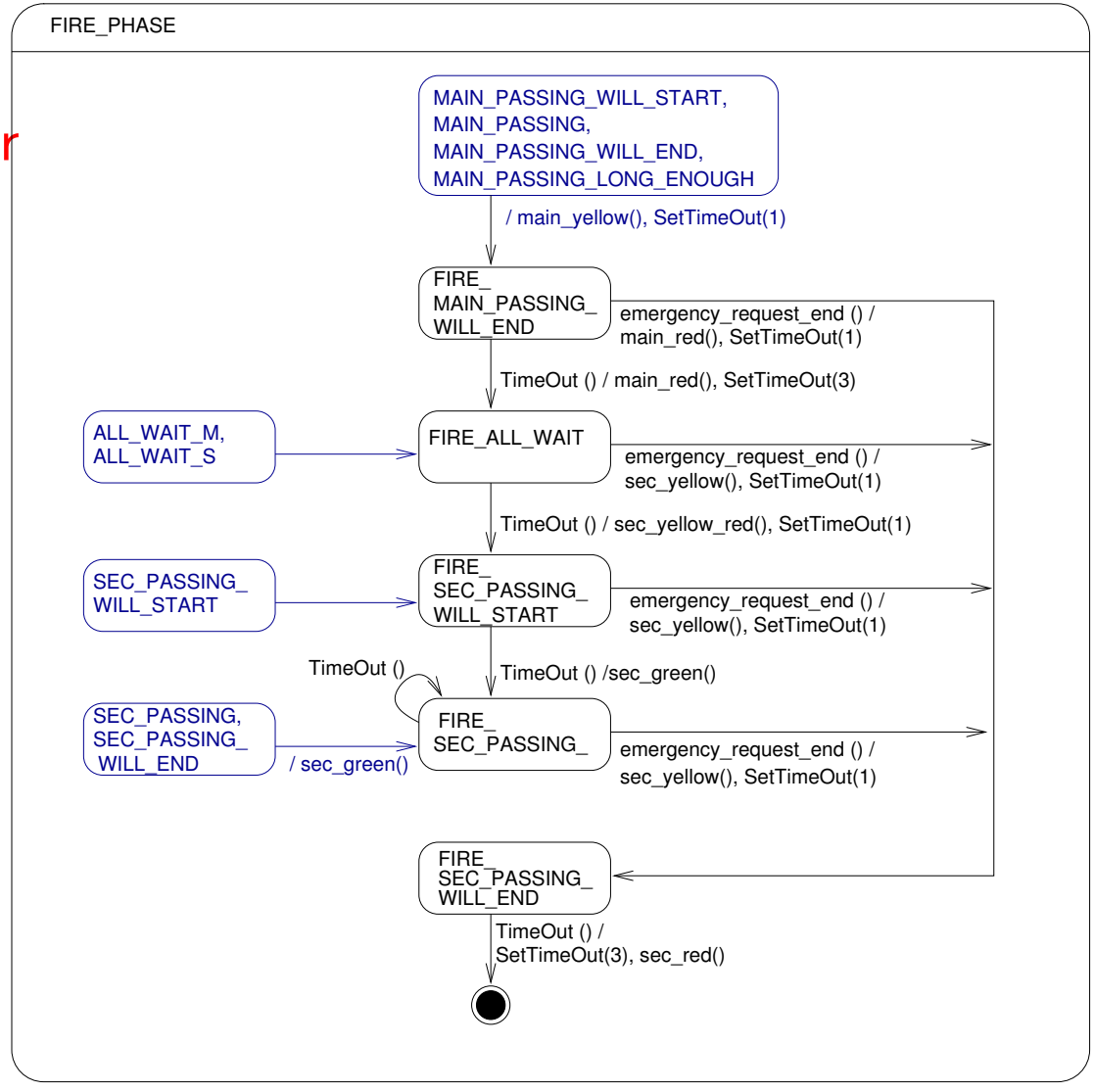


- The attribute *another_vehicle_waiting* is added to store a secondary road request.
- The state machine is using this data.

Component TrafficLightBehavior IV

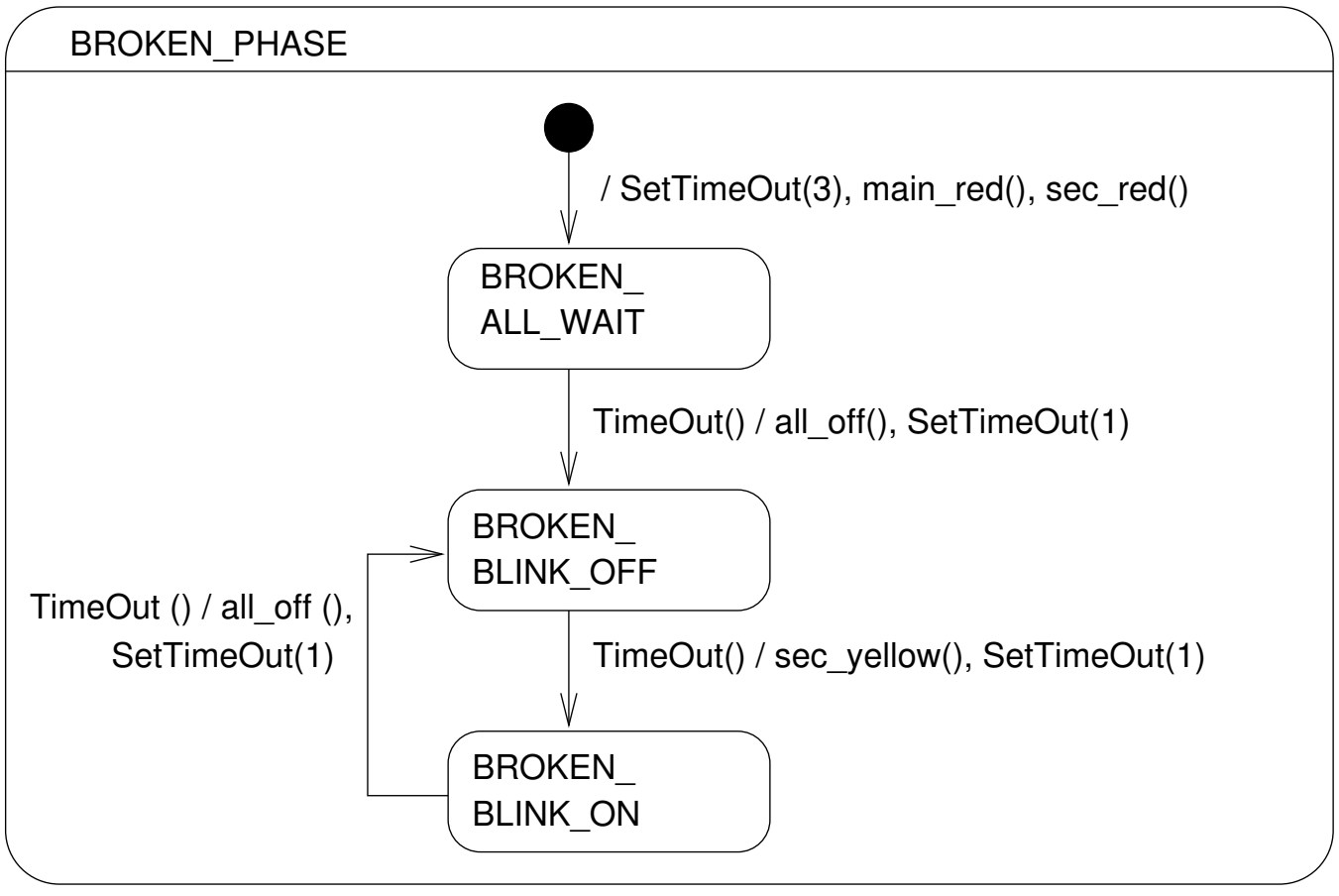
TrafficLightBehavior state machine for EmergencyRequest

In this state machine, the states of the other state machines are repeated (in blue) to express the behavior without changing all other state machines.



Component TrafficLightBehavior V

TrafficLightBehavior State Machine for BrokenLightSafeState



Component TrafficLightBehavior VI

TrafficLightBehavior merged state machine

This component is contained in all subproblems. The attribute *another_vehicle_waiting* is only contained in the subproblem *MainRoadPassing*. Therefore, in the state machines for the other subproblems this attribute must be considered:

- In the subproblem *SecondaryRoadPassing*, the signal *srr* must not change the attribute *another_vehicle_waiting*, since the cars on the secondary road are allowed to pass and they are not waiting.
- In the subproblem *EmergencyRequest*, the signal *srr* must not change the attribute *another_vehicle_waiting*, since the cars on the secondary road are allowed to pass and they are not waiting.

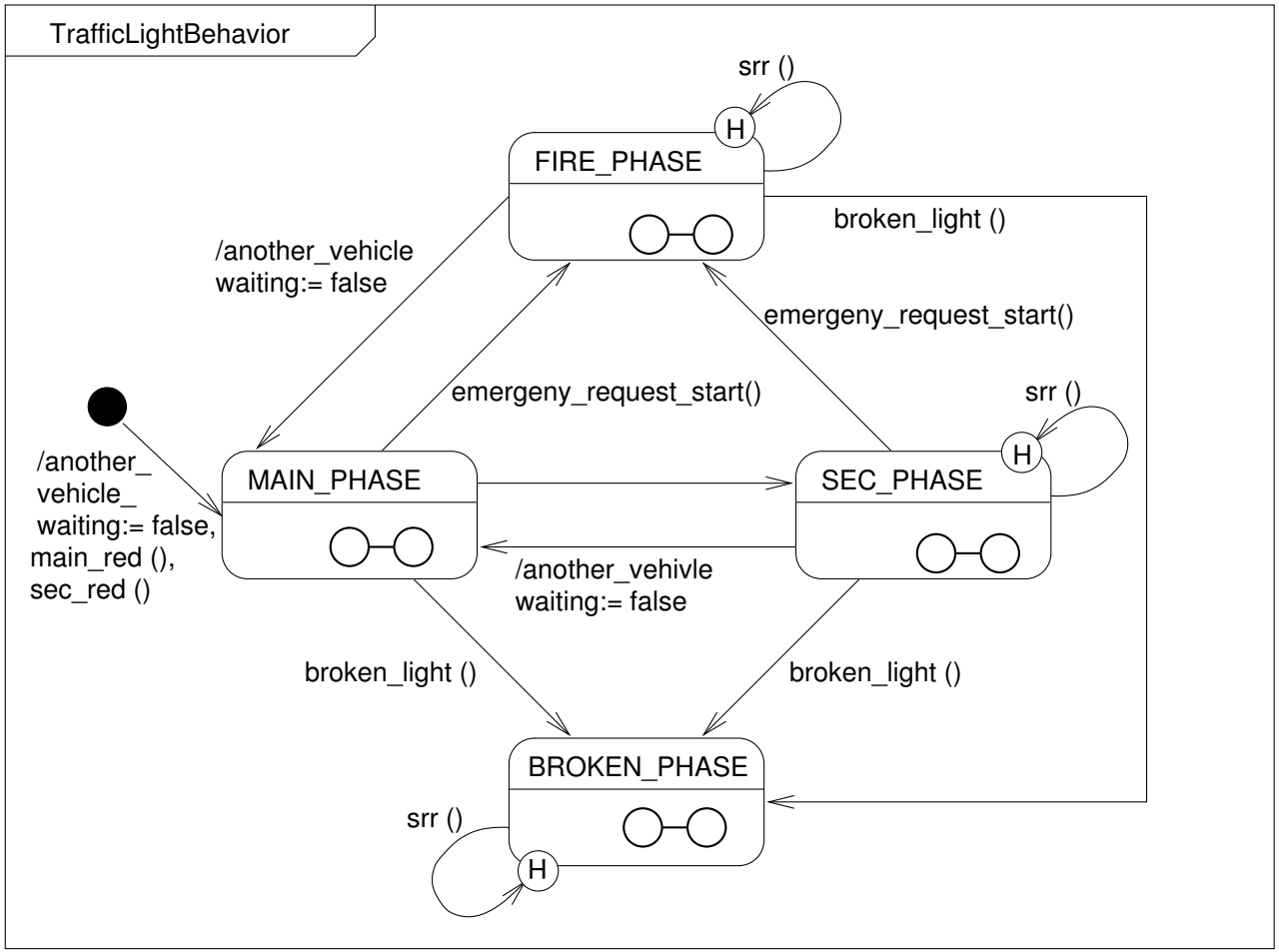
Component TrafficLightBehavior VII

- On start-up, each time the main phase is activated the attribute *another_vehicle_waiting* must be set to false.

Since the subproblems *MainRoadPassing* and *SecondaryRoadPassing* are related sequentially, one state machine will be activated as soon as the other state machine terminates. The state machines for the subproblems *EmergencyRequest* and *BrokenLightSafeState* are parallel and activated with the signals *broken_light()* or *emergency_request_start()*. Once activated, they take control over the output signals. Additionally, the initialization sequence

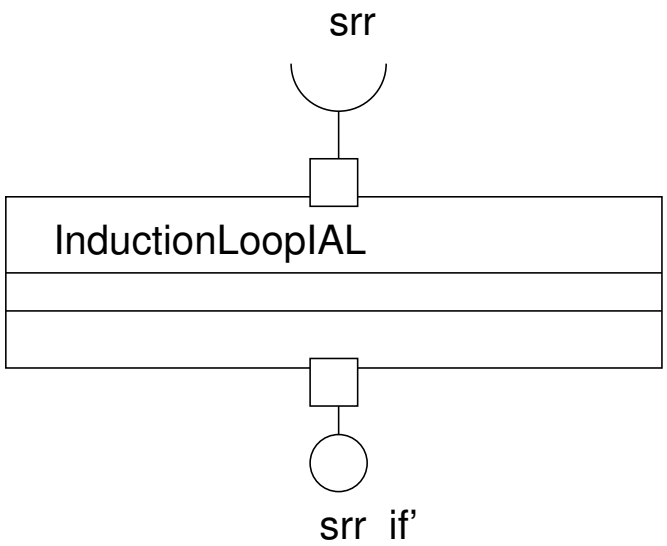
Component TrafficLightBehavior VIII

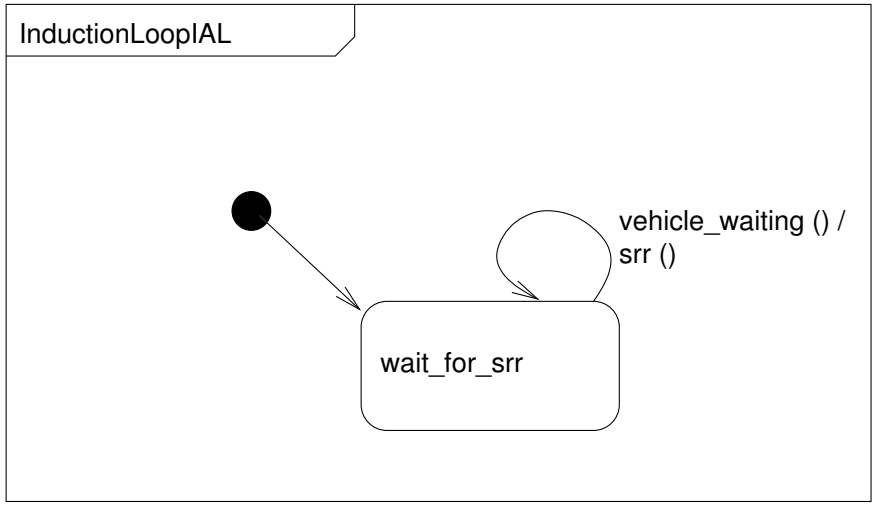
is considered.



Component: InductionLoopIAL I

InductionLoopIAL component overview and state machine for MainRoadPassing



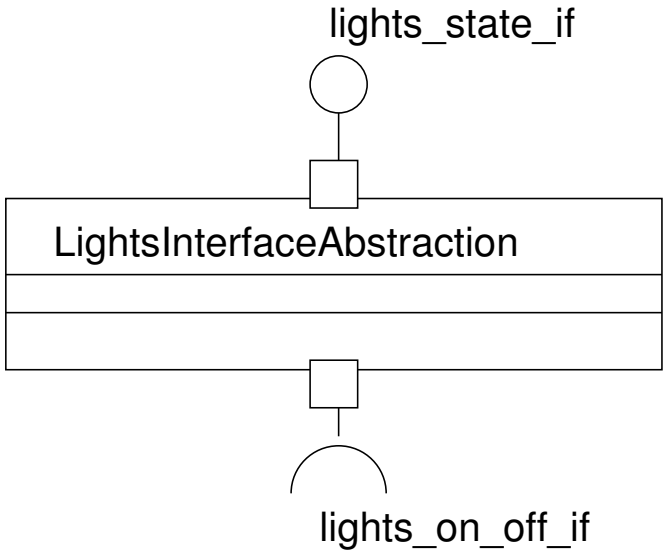


- It is a passive component since it reacts immediately to input signals.

This component is only contained in subproblem *MailRoadPassing*, and no merge is performed.

Component LightsInterfaceAbstraction I

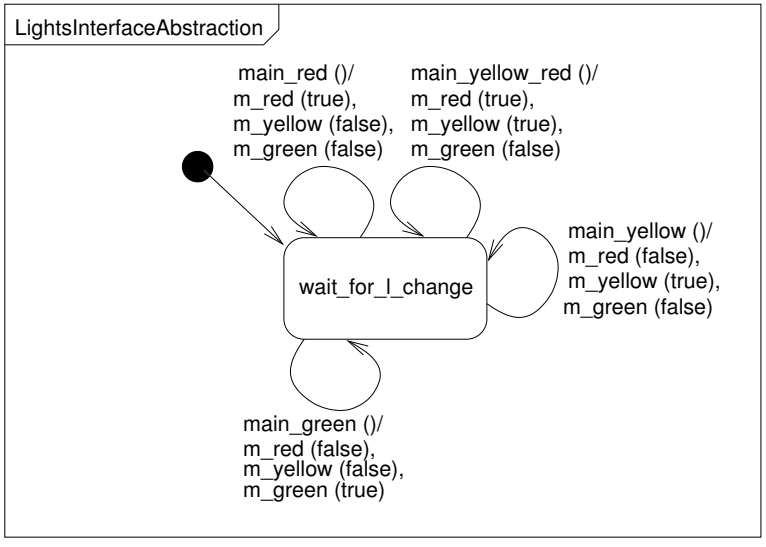
LightsInterfaceAbstraction component overview



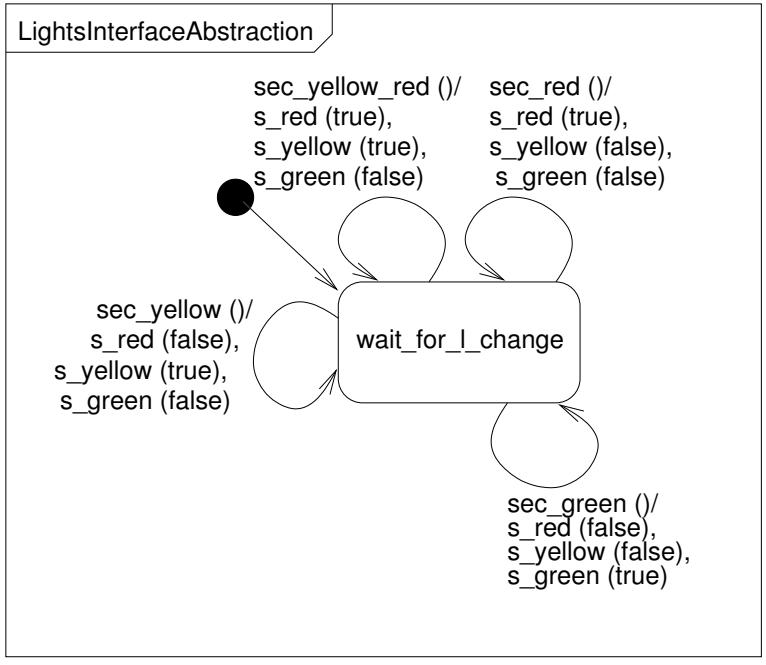
- It is a passive component since it reacts immediately to input signals.

Component LightsInterfaceAbstraction II

LightsInterfaceAbstraction state machine for MainRoad-Passing

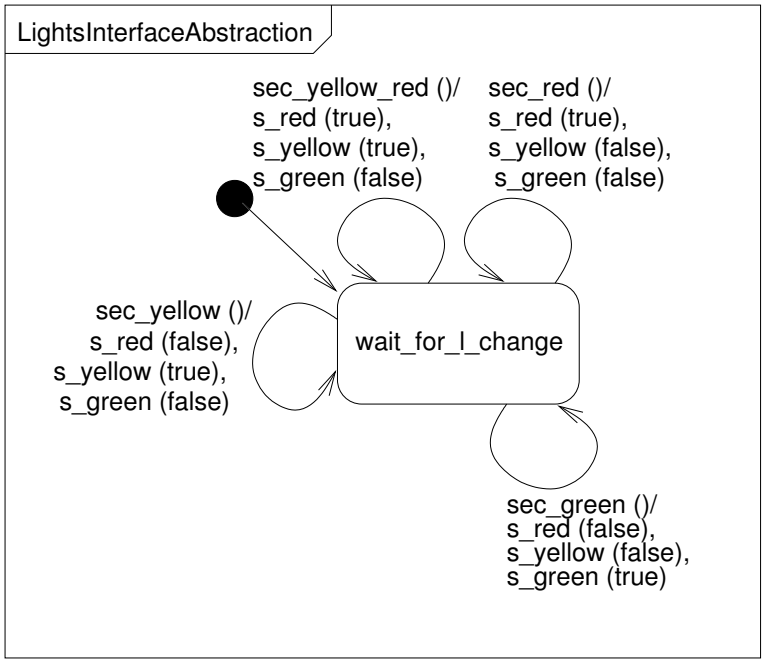


LightsInterfaceAbstraction state machine for Secondary-RoadPassing

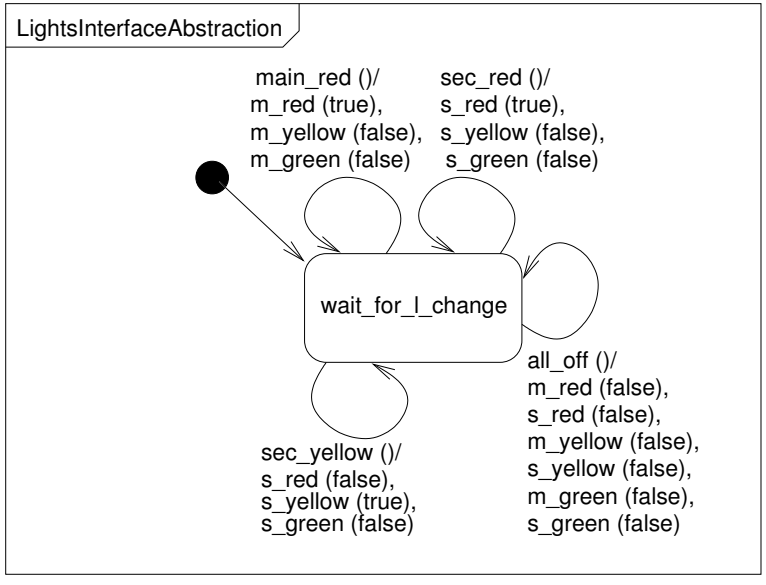


Component LightsInterfaceAbstraction III

LightsInterfaceAbstraction state machine for EmergencyRequest

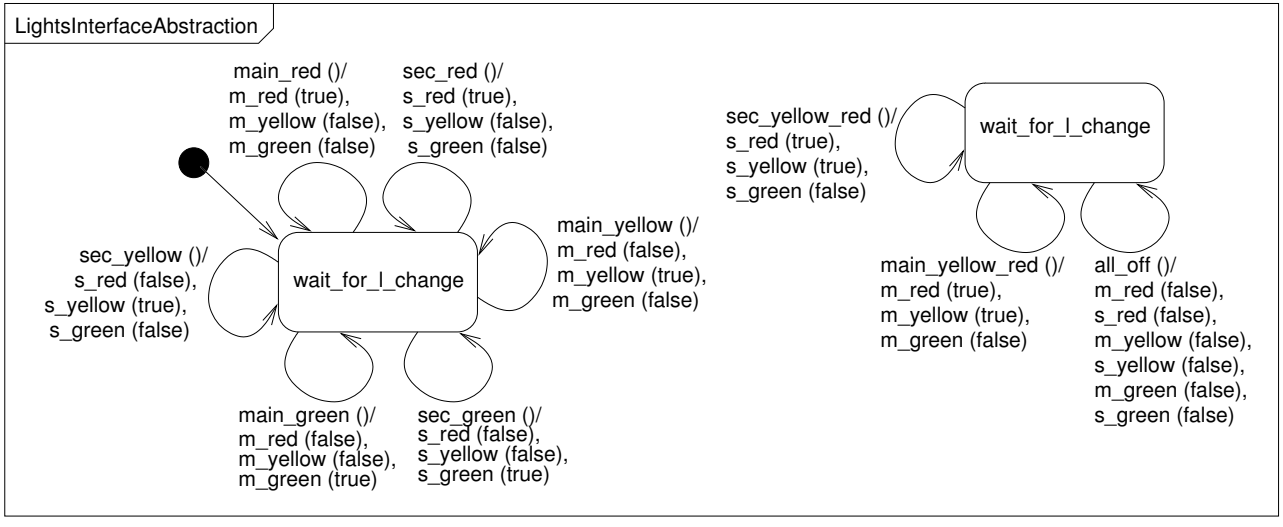


LightsInterfaceAbstraction state machine for BrokenLightSafeState



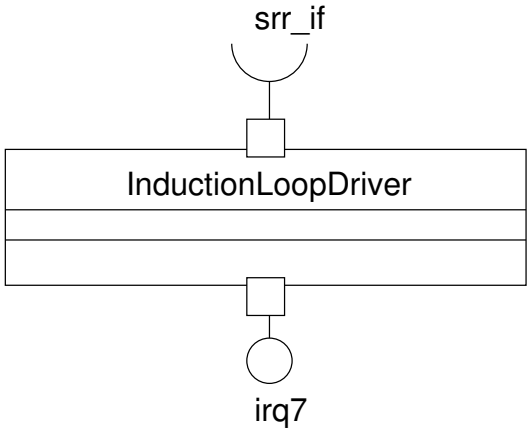
Component LightsInterfaceAbstraction IV

LightsInterfaceAbstraction global state machine



Component: InductionLoopDriver

InductionLoopDriver component overview

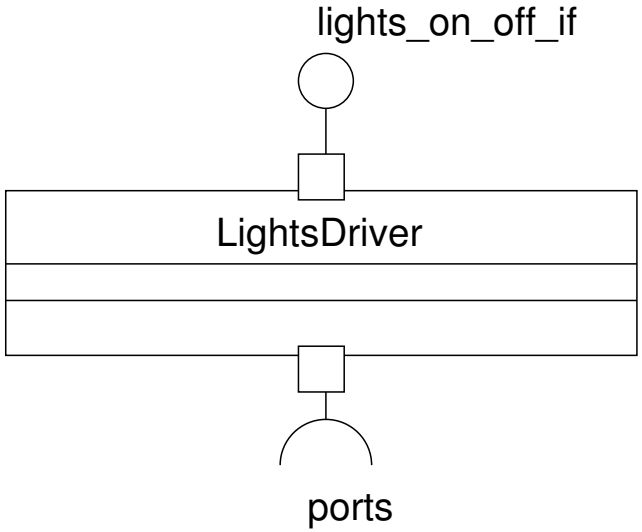


- When this component receives the signal *interrupt_request_7*, it emits a *srr* signal.
- It is a passive component since it reacts immediately to input signals from the Microcontroller.
- It is not specified here since it only converts the interrupt into a signal for the IAL.

This component is only contained in subproblem *MailRoadPassing* and need not be merged.

Component LightsDriver

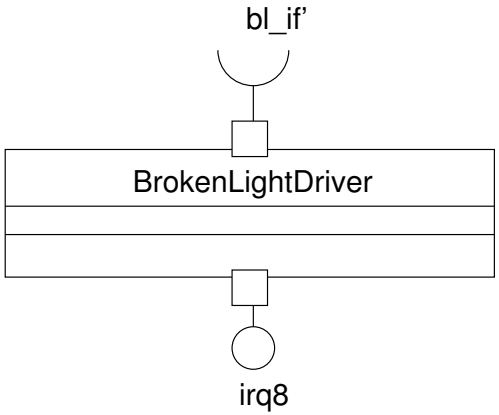
LightsDriver component overview



- This component sets the ports to switch the lights on or off.
- It is a passive component since it reacts immediately to input signals from the IAL.
- It is not specified here since it only passes on the input signals from the IAL to specific ports of the microcontroller.

Component: BrokenLightDriver

BrokenLightDriver component overview

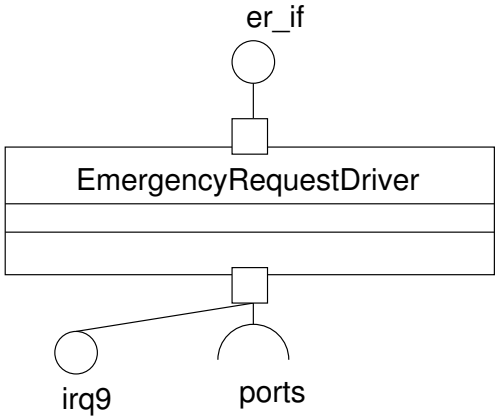


- When this component receives the signal *interrupt_request_8*, it emits a *broken_light* signal.
- It is a passive component since it reacts immediately to input signals from the Microcontroller.
- It is not specified here since it only converts the interrupt into a signal for the IAL.

This component is only contained in subproblem *BrokenLightSafeState* and need not be merged.

Component: EmergencyRequestDriver

EmergencyRequestDriver component overview



- When this component receives *interrupt_request_9*, it reads out the port connected to the emergency request switch and emits the signal *emergency_request_start()* or the signal *emergency_request_end()*.
- It is a passive component since it reacts immediately to input signals from the Microcontroller.
- It is not specified here since it only reads out the input port in case of an interrupt request and generates the signal for the IAL.

This component is only contained in subproblem *EmergencyRequest* and need not be merged.

Component Microcontroller

Microcontroller component overview

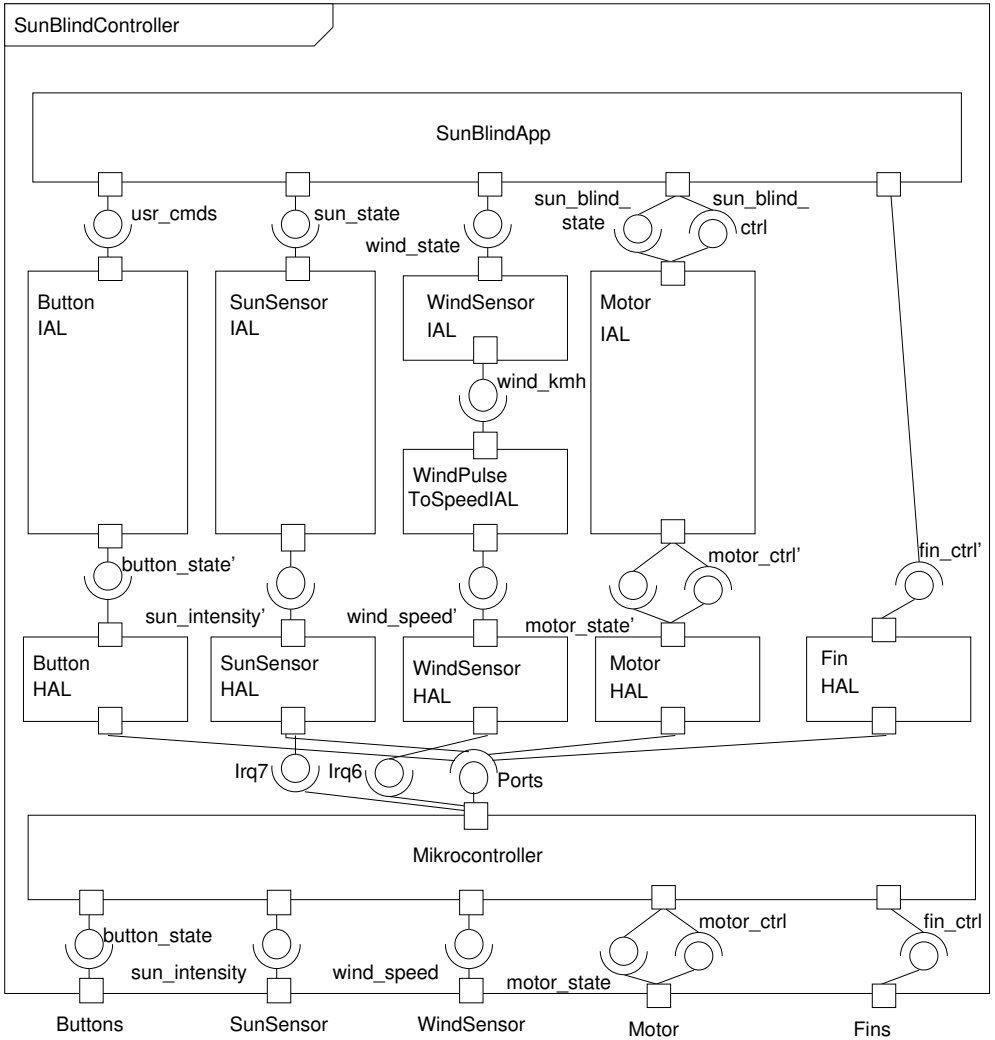
- The *Microcontroller* is a reused existing component.
- It is not specified here since it is described in its datasheet.
- It is an active component since it is a hardware component.
- The component requires and provides at least the same interfaces as shown in the global software architecture.

Validation

- The state machines are consistent with the interface behavior from Phase 8. All states are covered. Additional states ending with *_PASSING_WILL_START* are introduced.
- Each architectural component is covered, and in all state machines each possible input signal (as specified in Phase 7) is taken into account. The interface classes are the same as in Phase 7.

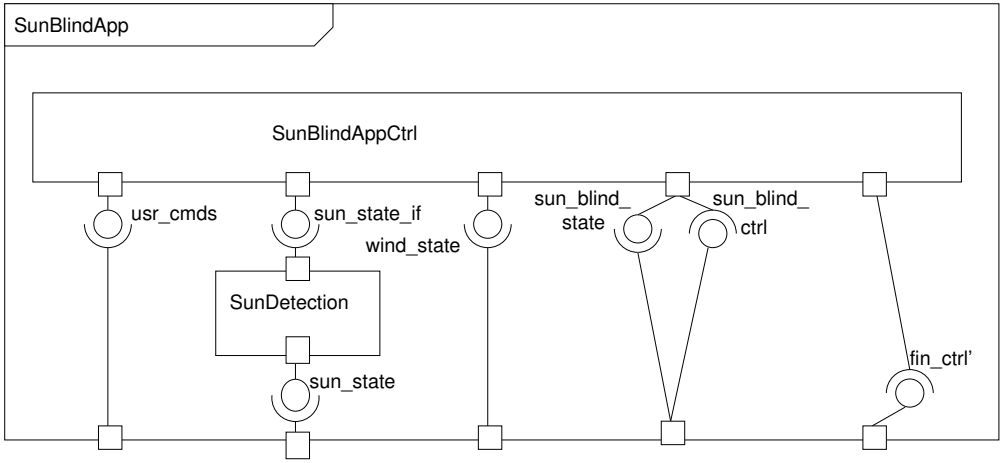
Example 2: sun blind control

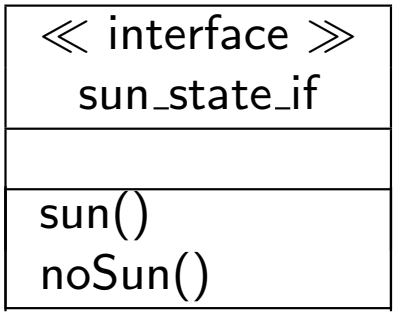
Global software architecture (repetition)



Component SunControlApplication (repetition)

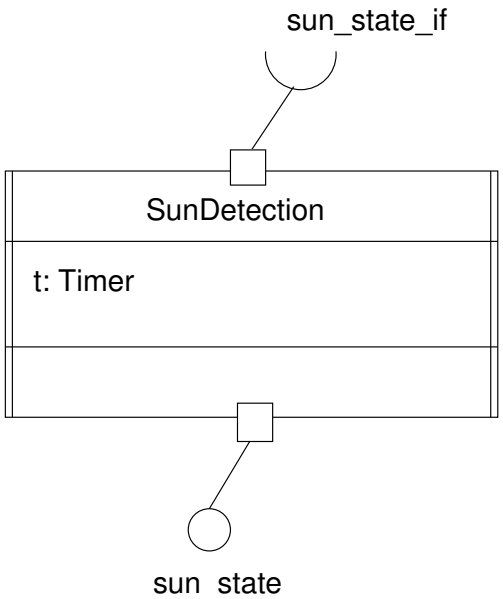
This component can be deomposed as follows:





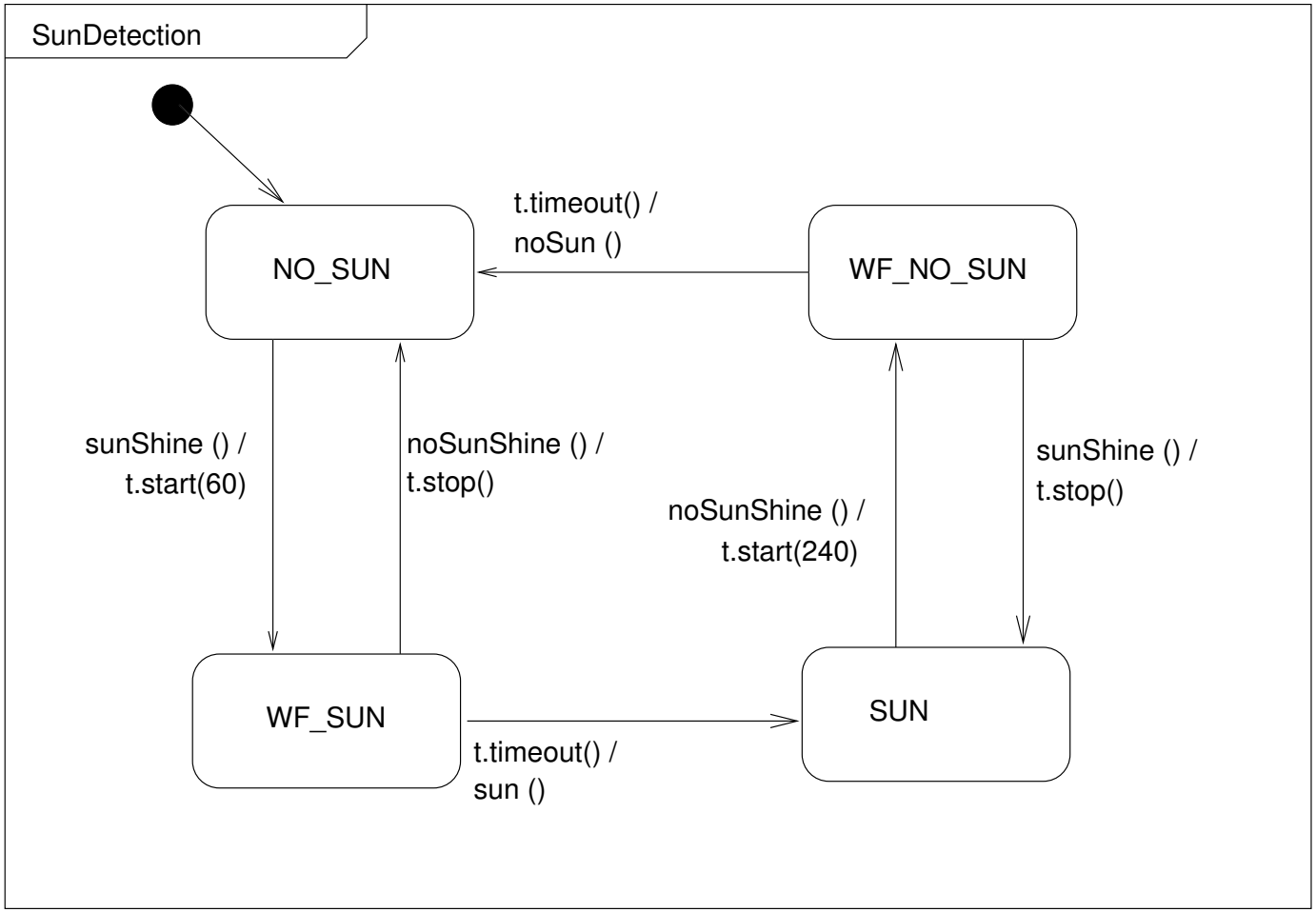
Component SunDetection I

SunDetection component overview



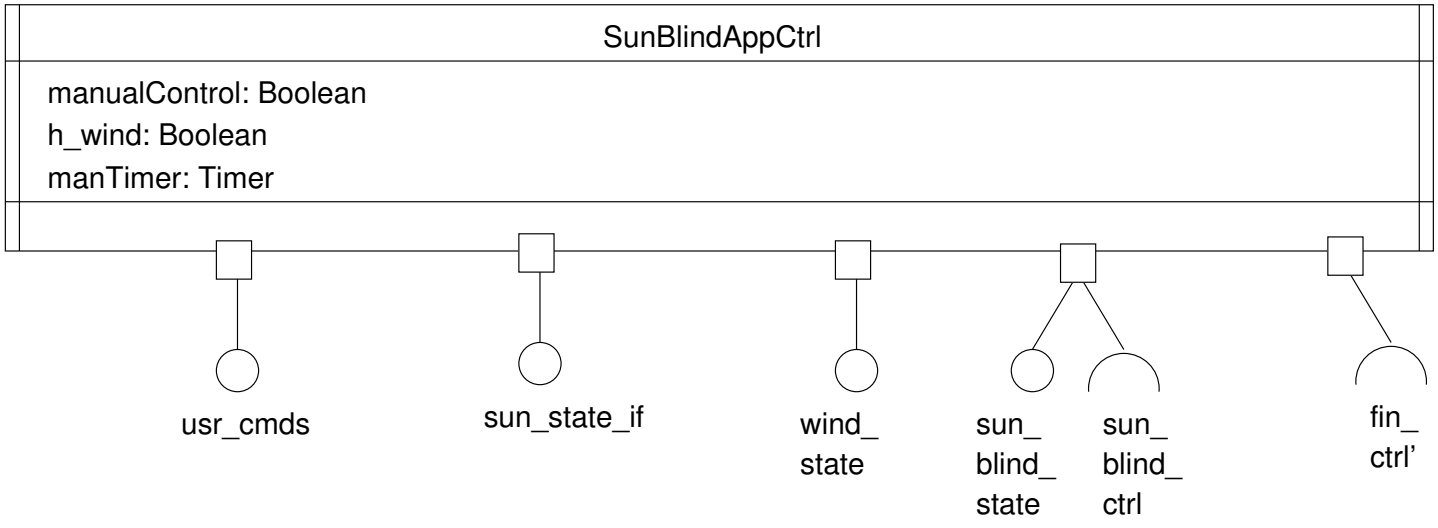
Component SunDetection II

SunDetection state machine



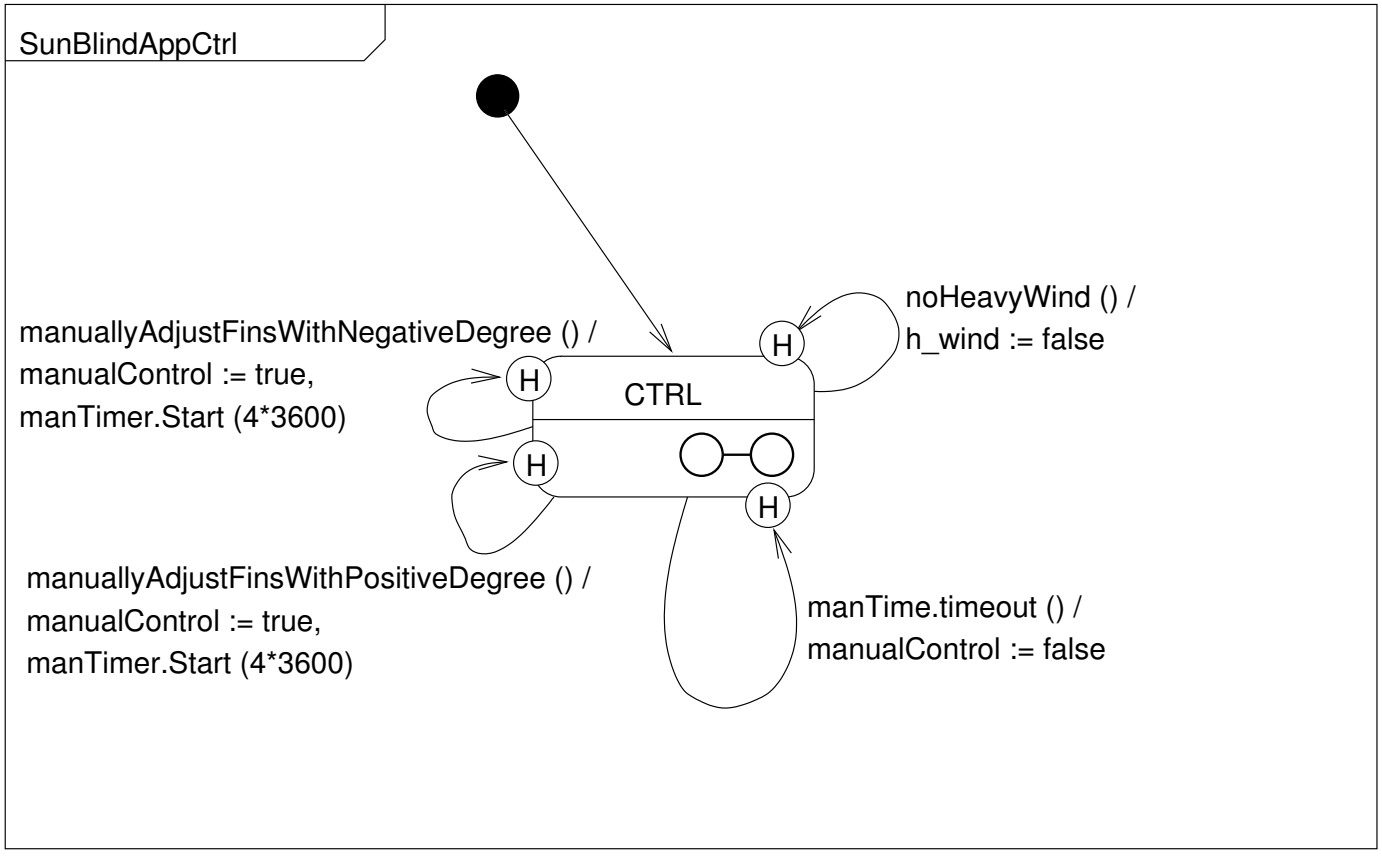
Component SunBlindAppCtrl I

SunBlindAppCtrl component overview

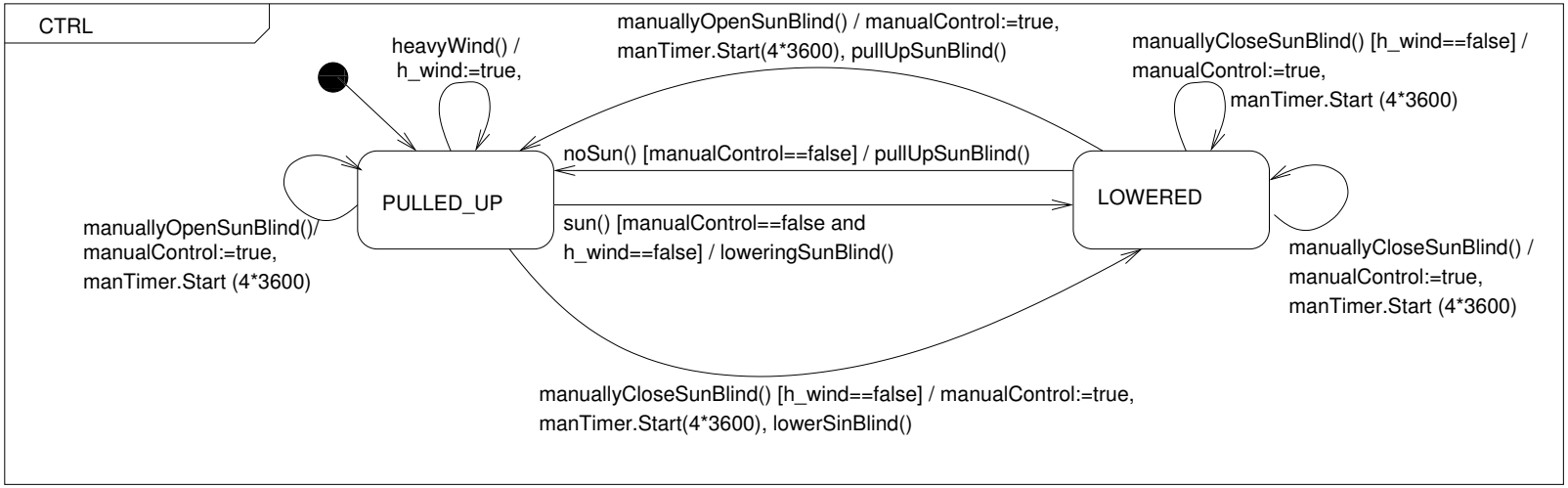


Component SunBlindAppCtrl II

SunBlindAppCtrl state machine for SunControl

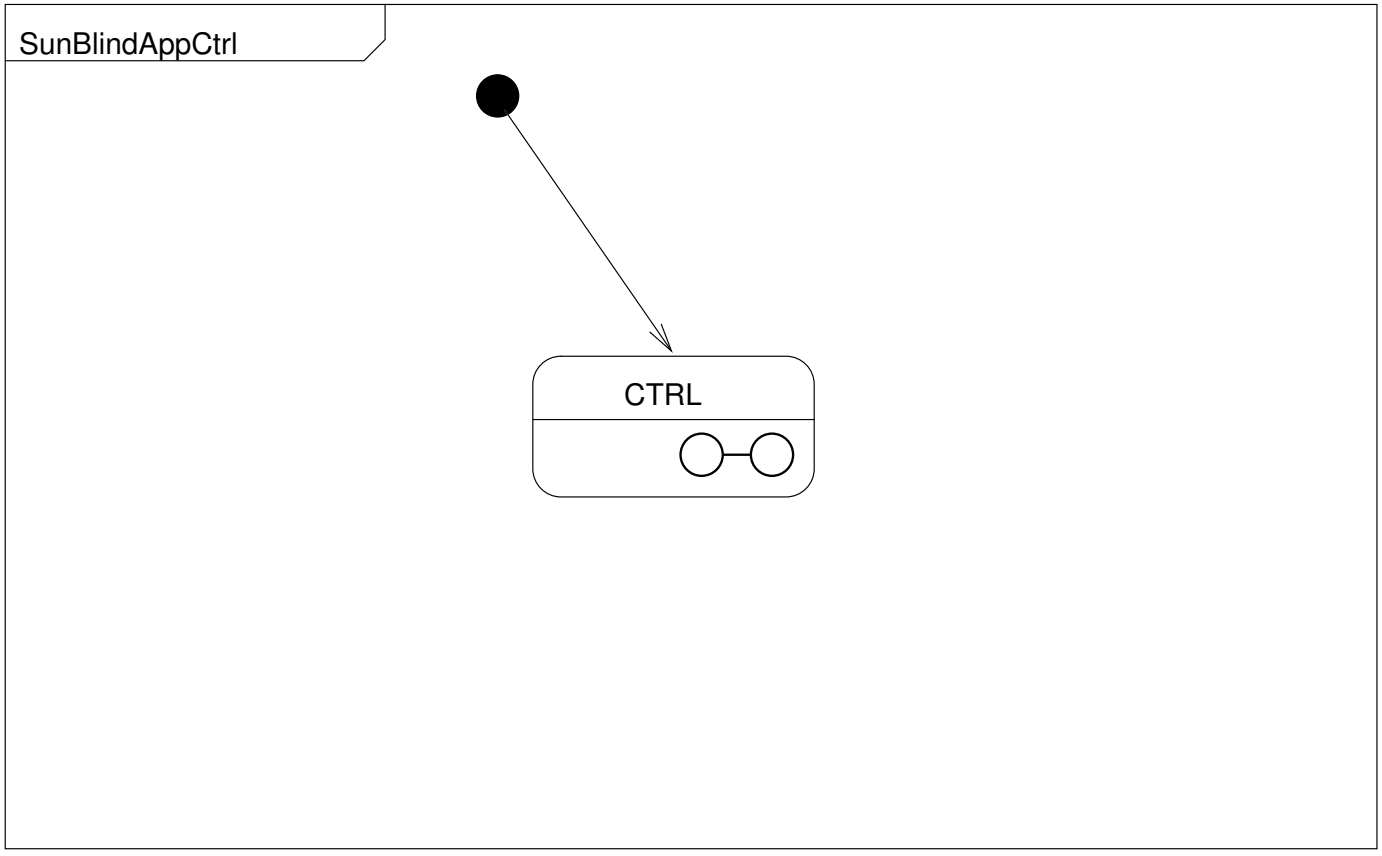


Component SunBlindAppCtrl III

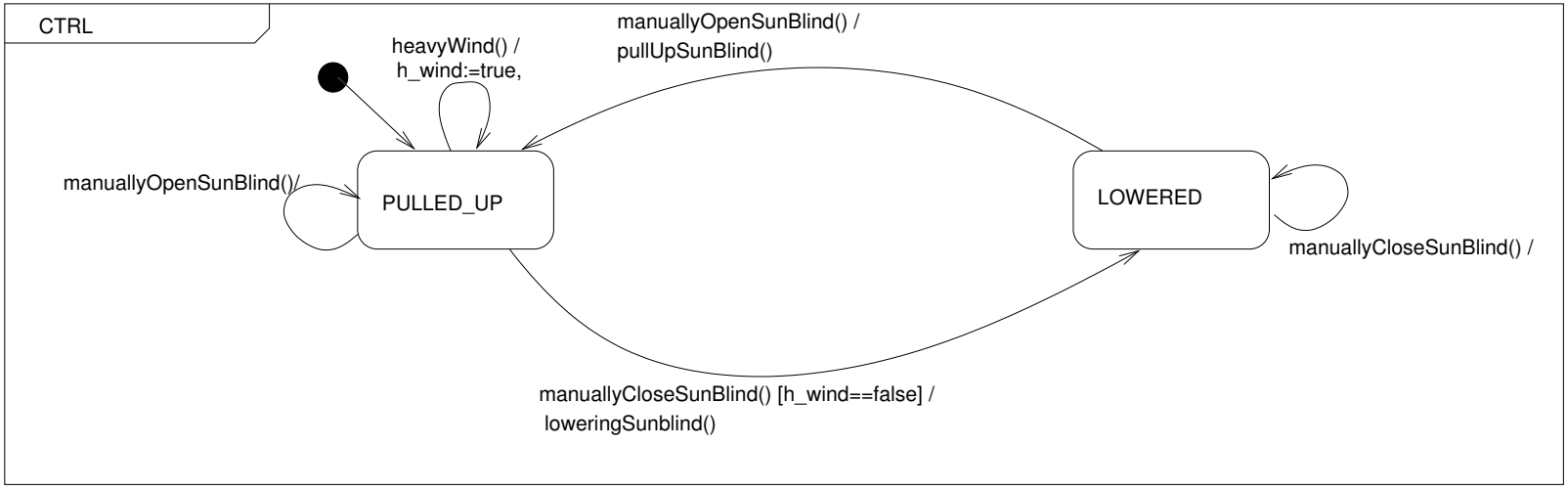


Component SunBlindAppCtrl IV

SunBlindAppCtrl state machines for UserControl

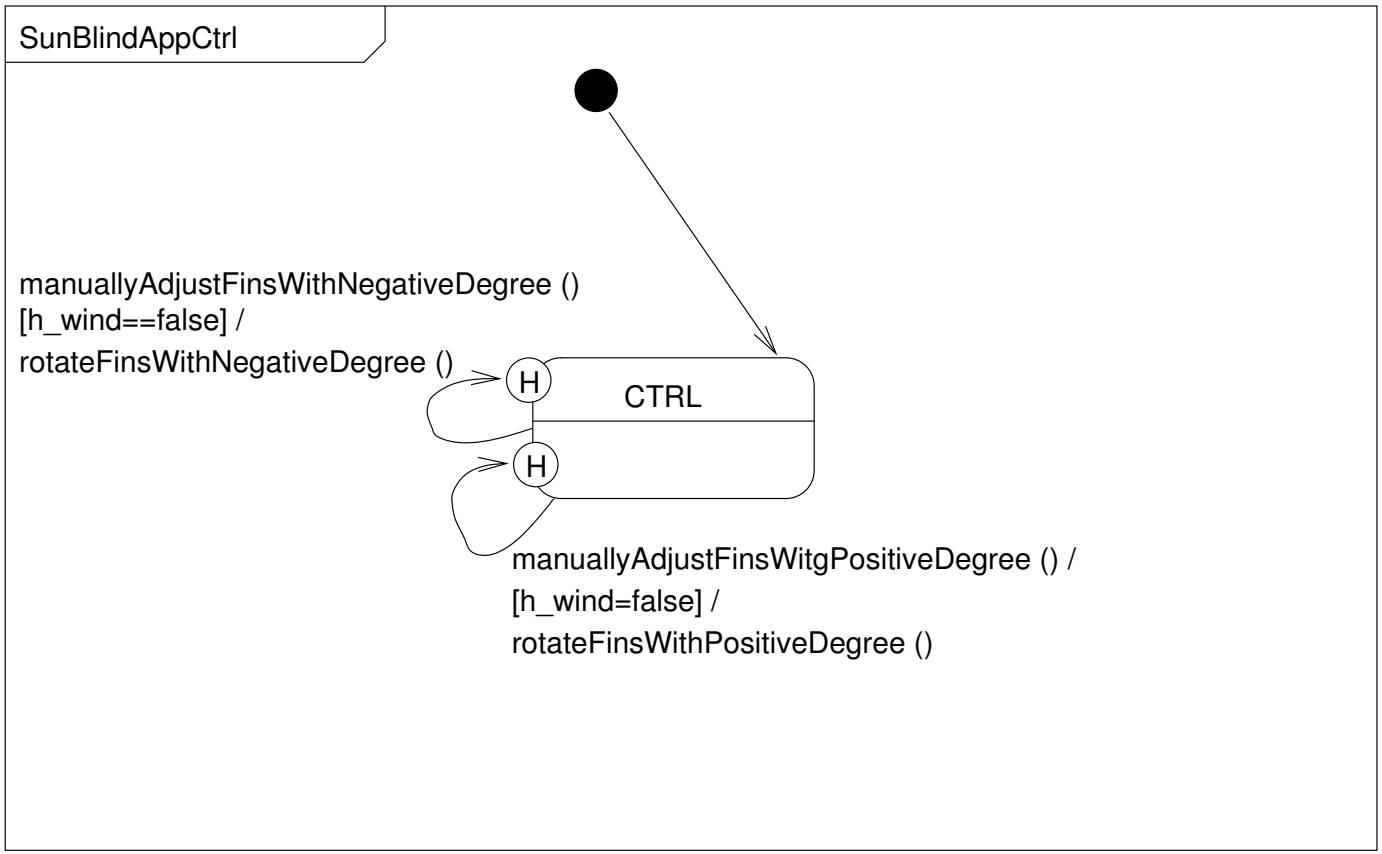


Component SunBlindAppCtrl V



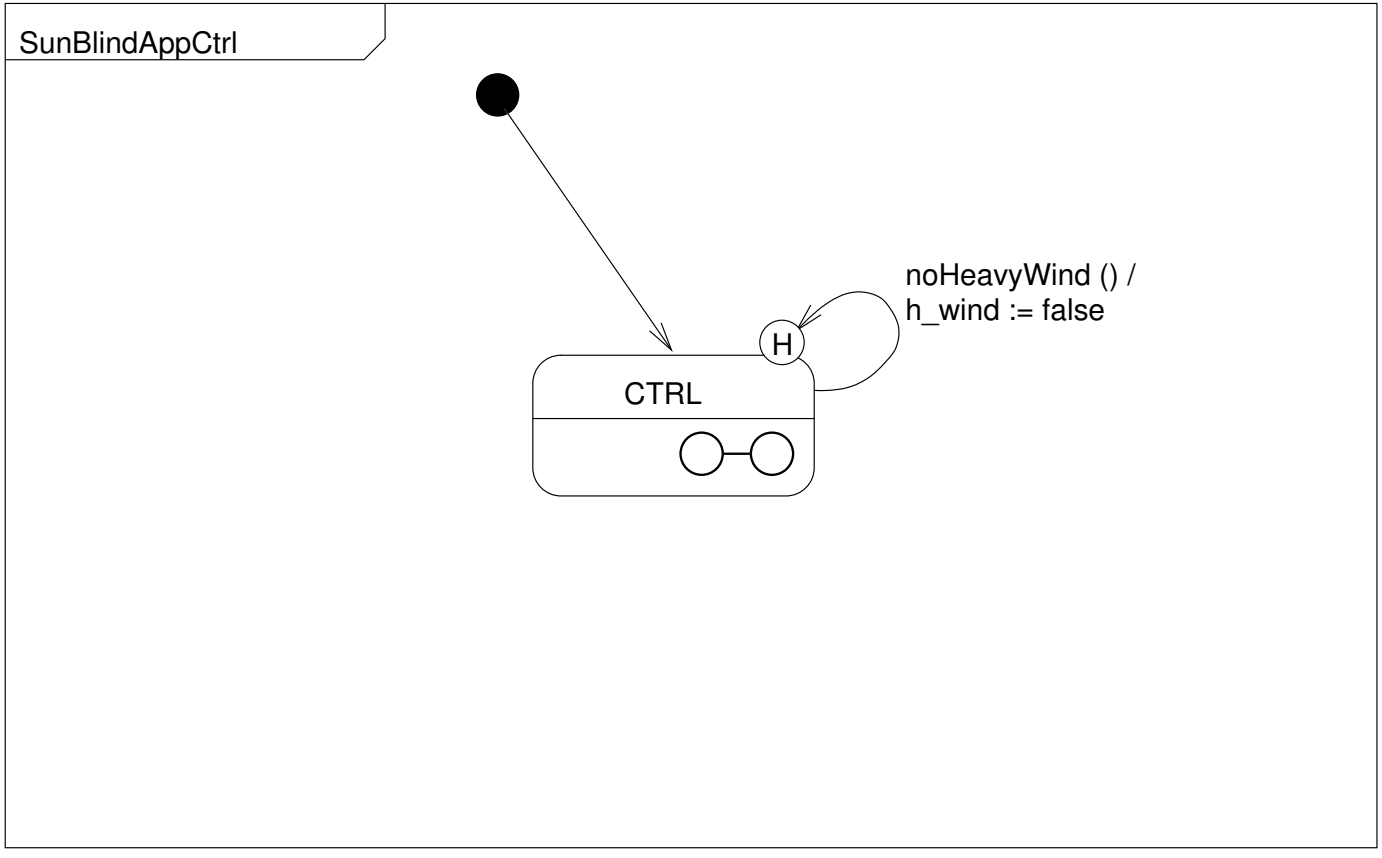
Component SunBlindAppCtrl VI

SunBlindAppCtrl state machine for FinsControl

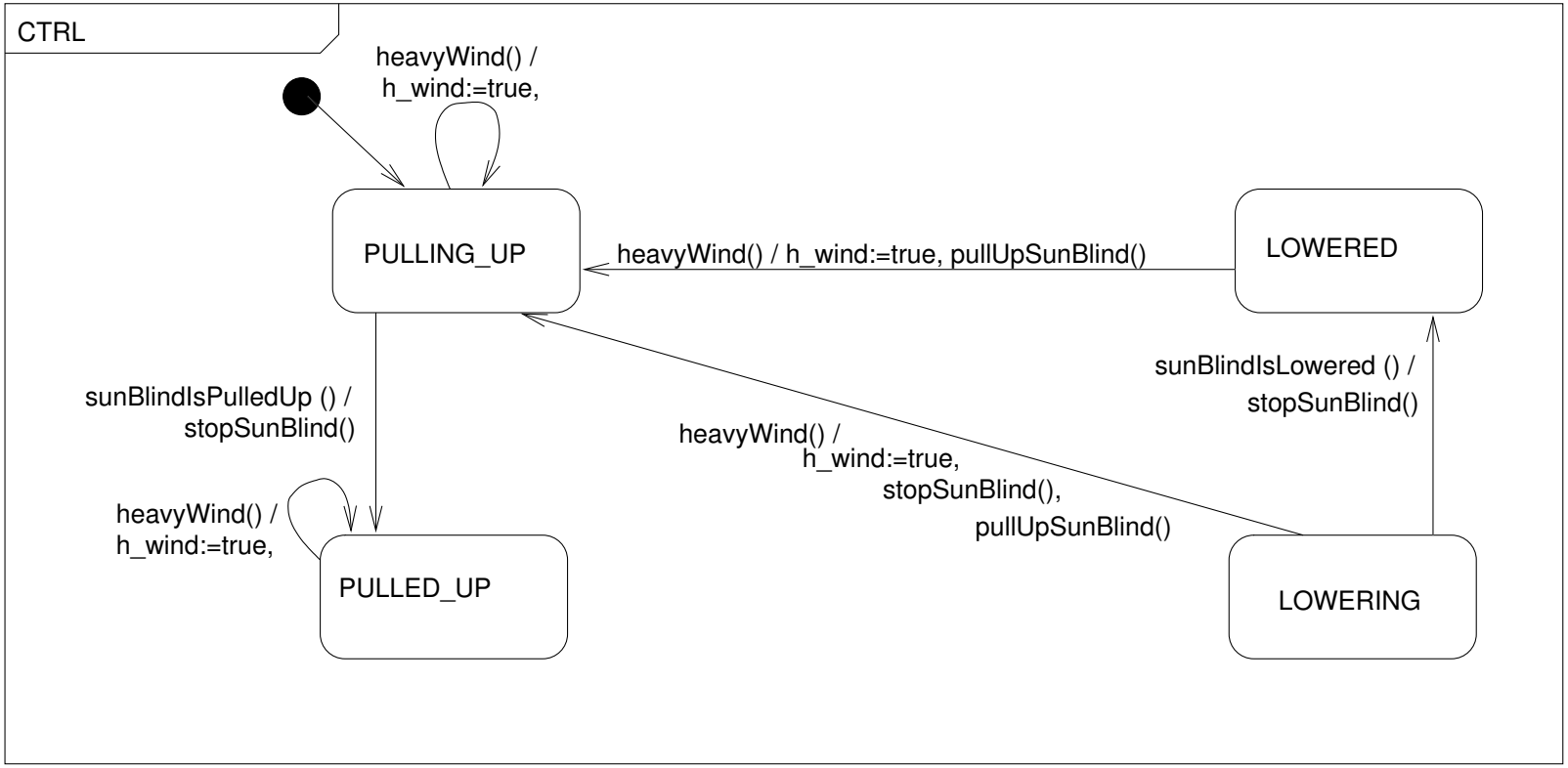


Component SunBlindAppCtrl VII

SunBlindAppCtrl state machines for NoDestructControl

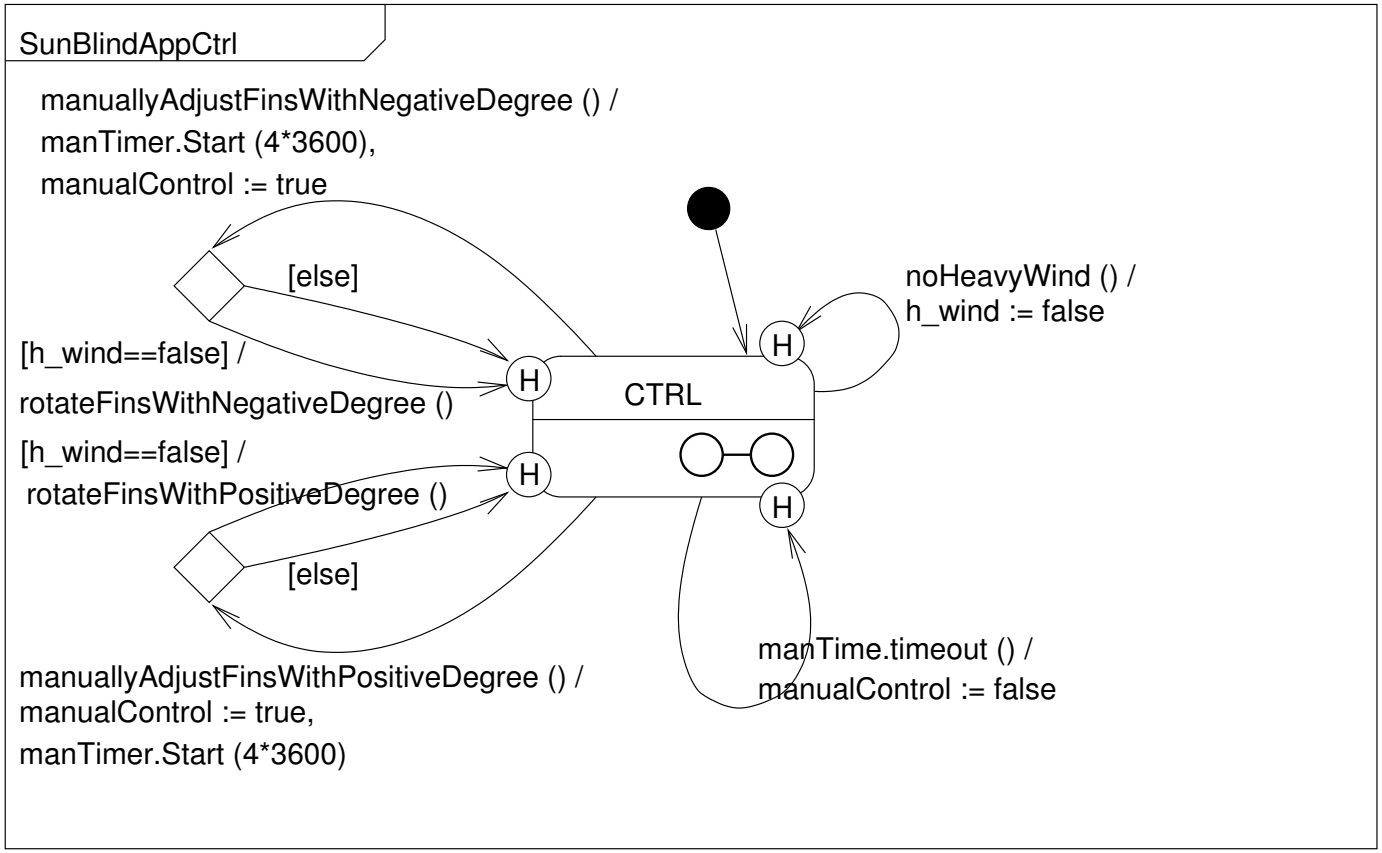


Component SunBlindAppCtrl VIII

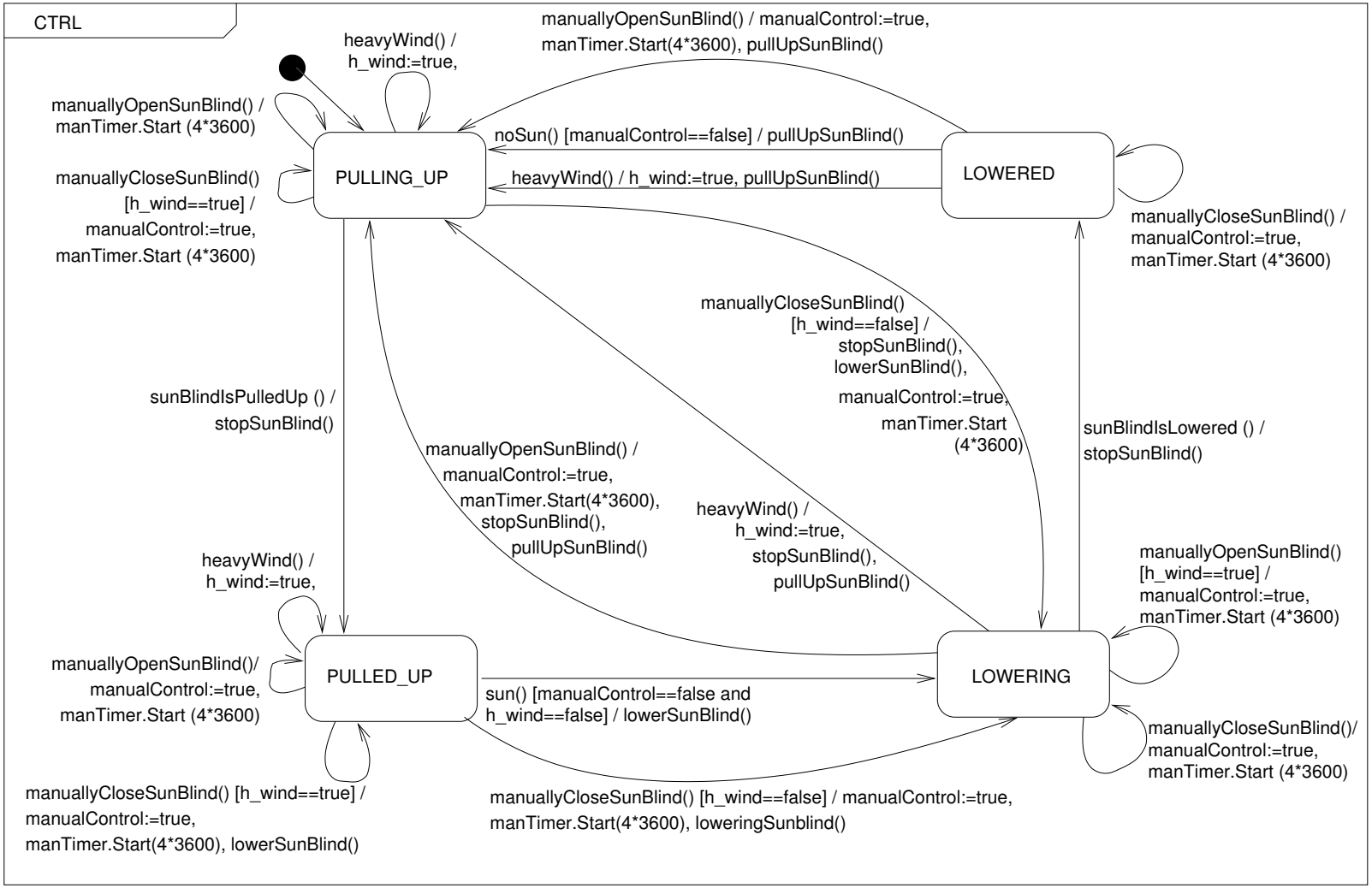


Component SunBlindAppCtrl IX

SunBlindAppCtrl merged state machine

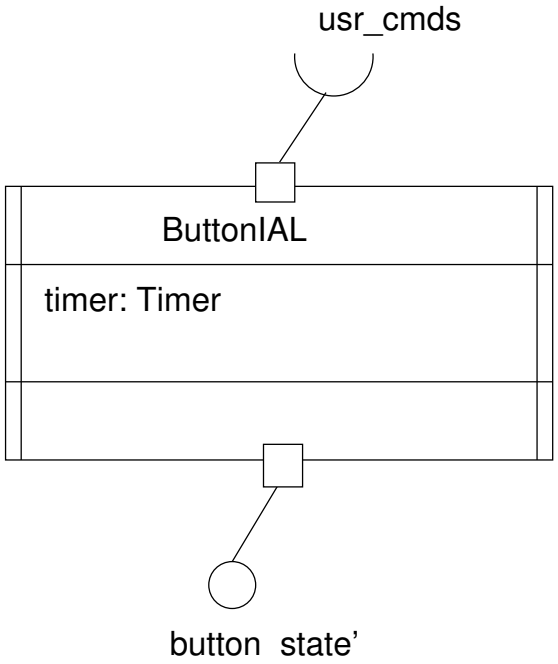


Component SunBlindAppCtrl X



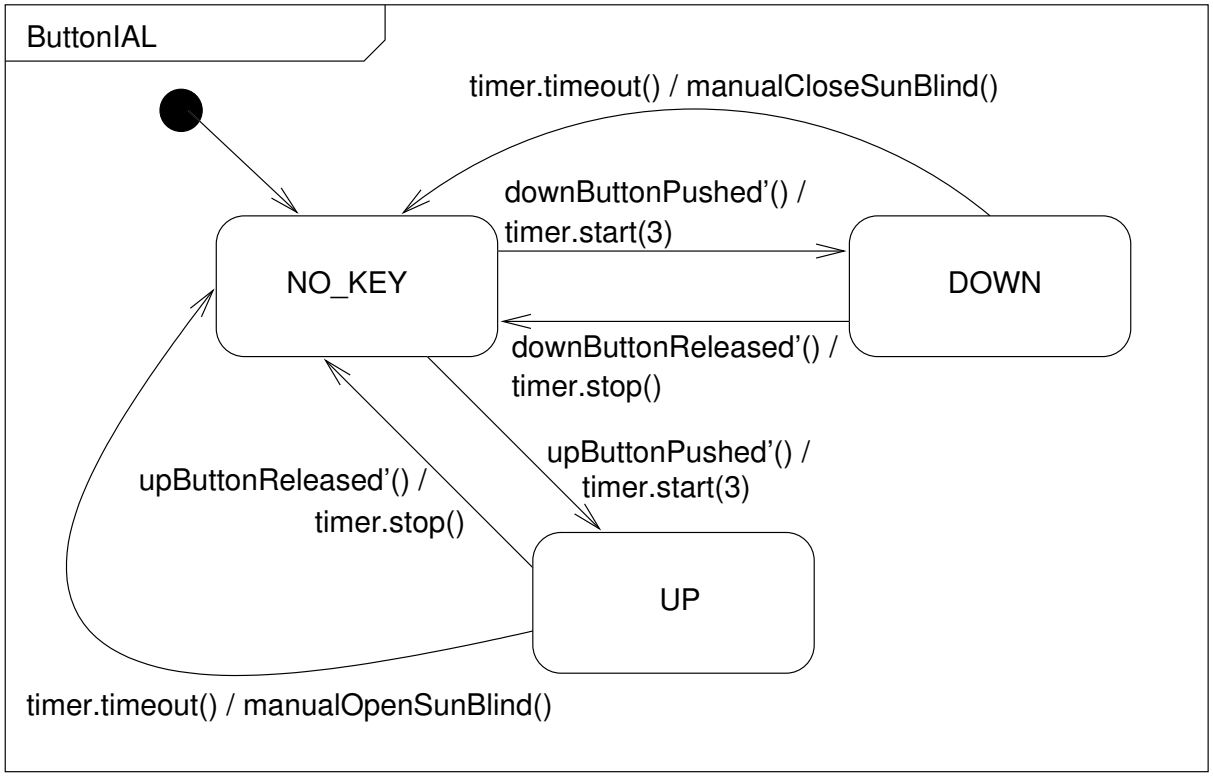
Component ButtonIAL I

ButtonIAL component overview



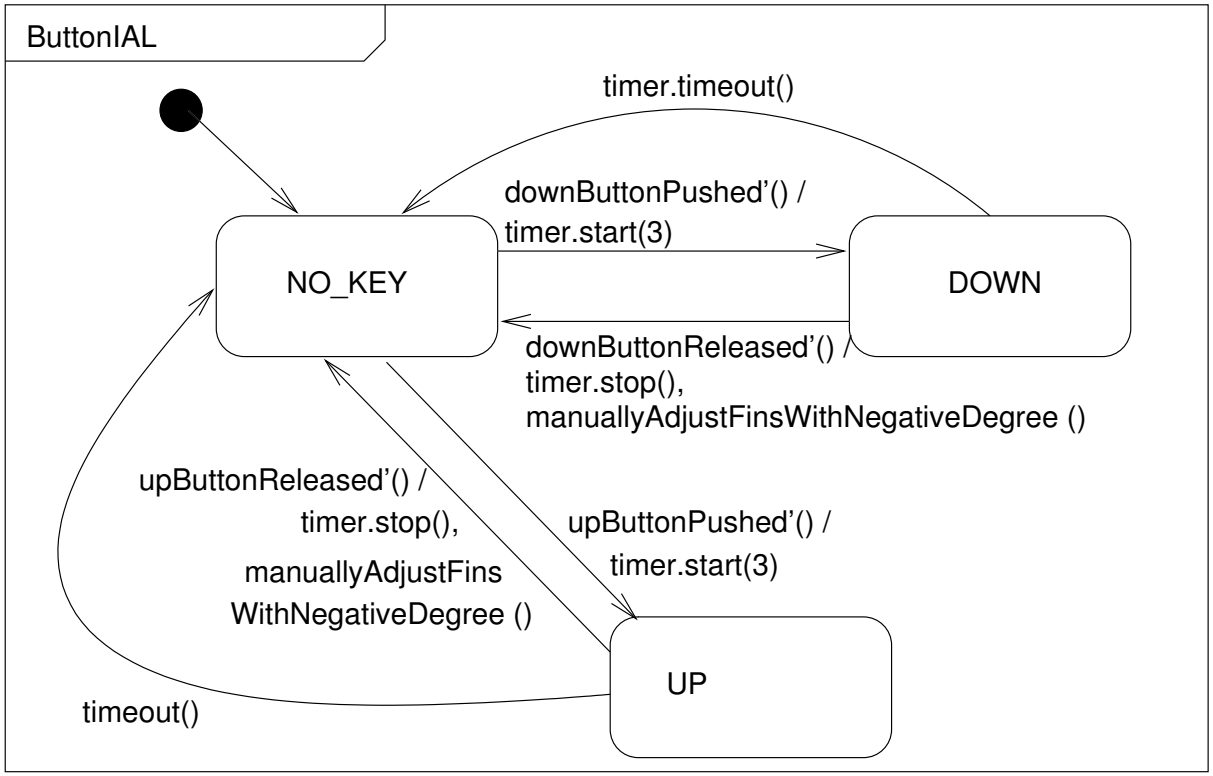
Component ButtonIAL II

ButtonIAL state machine for SunControl and UserControl



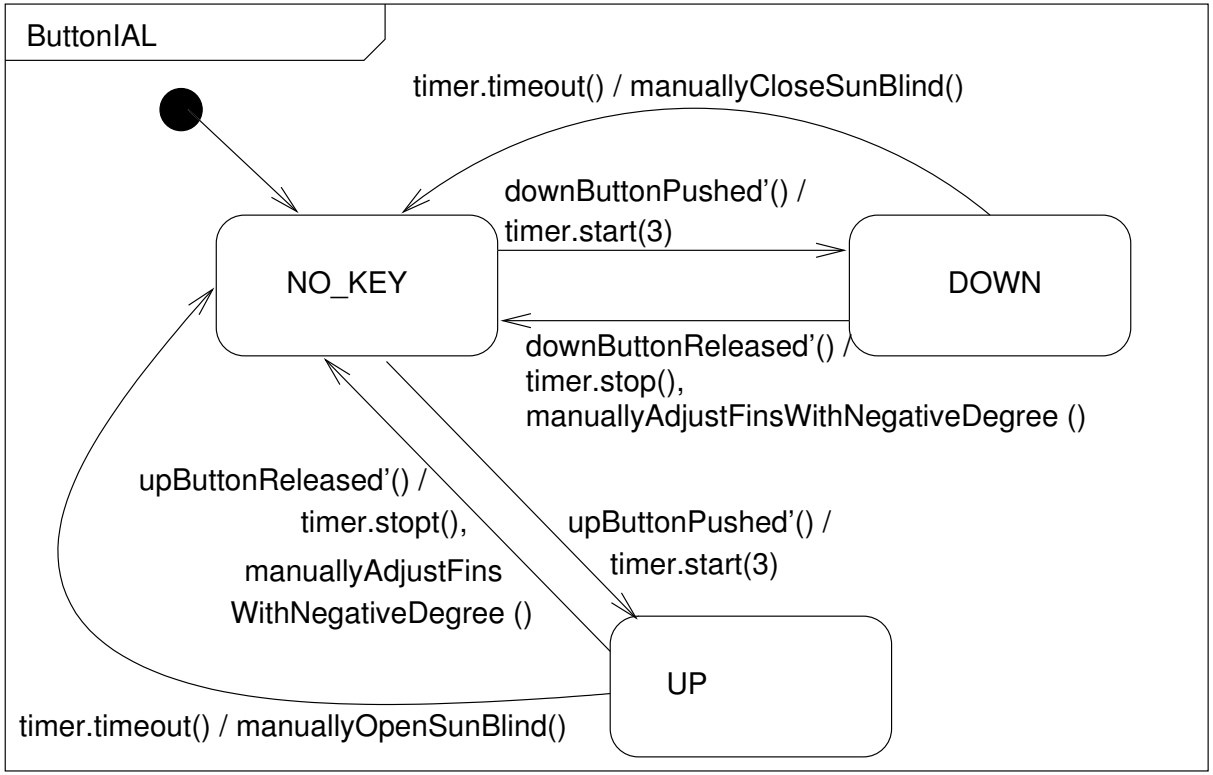
Component ButtonIAL III

ButtonIAL state machine for FinsControl



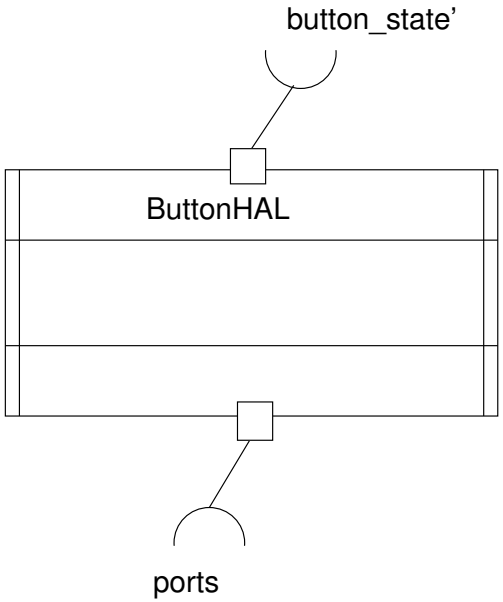
Component ButtonIAL IV

ButtonIAL merged state machine



Component ButtonHAL

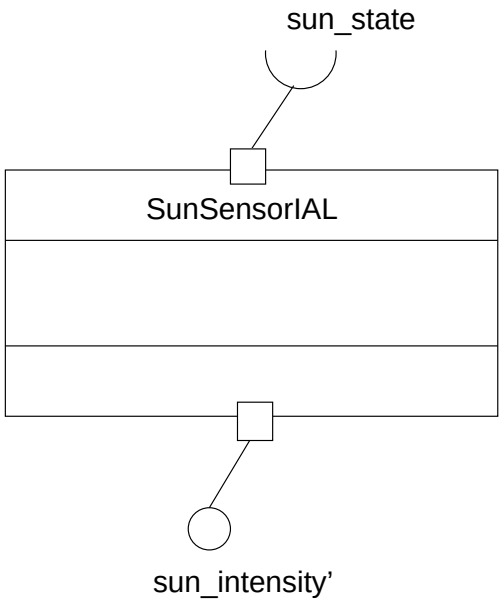
ButtonHAL component overview



This component checks the port of the microcontroller representing the state of the buttons every 20 ms and generates the *pressed* and *released* signals if the state of the port changes.

Component SunSensorIAL I

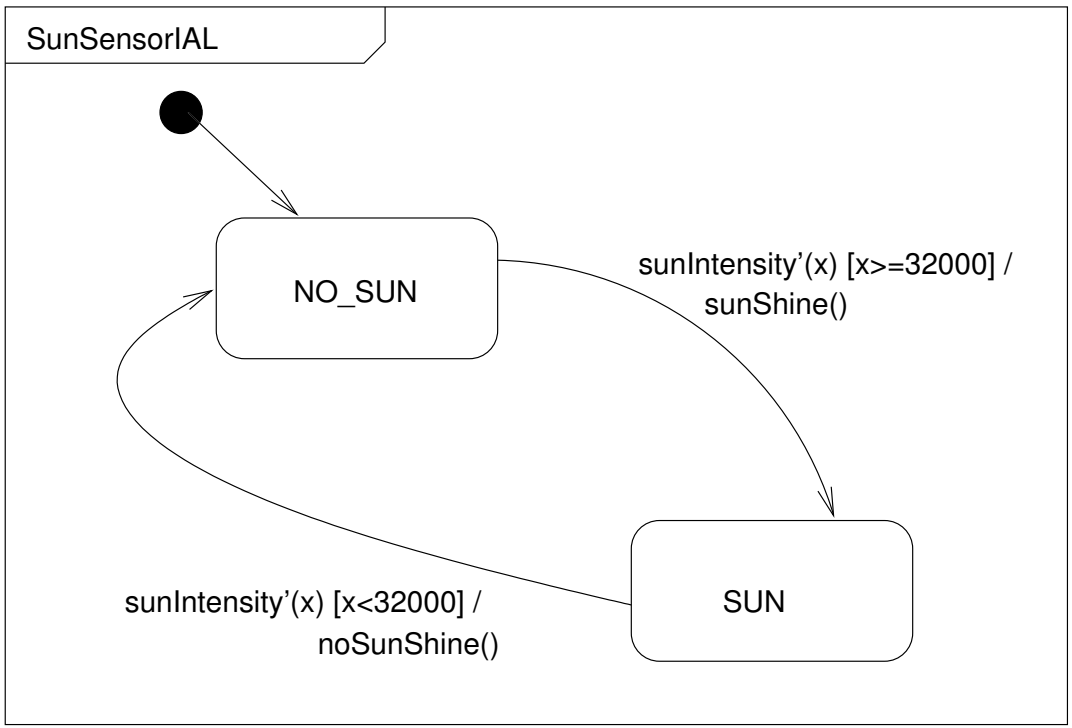
SunSensorIAL component overview



This component is contained only in the subproblem SunControl and need not to be merged.

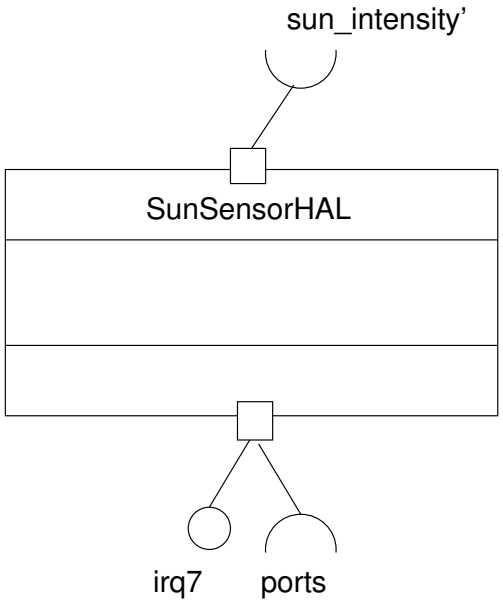
Component SunSensorIAL II

SunSensorIAL state machine



Component SunSensorHAL

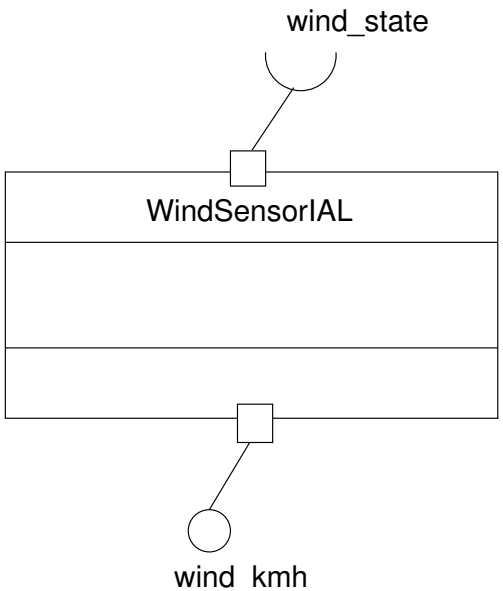
SunSensorHAL component overview



When this component receives an intrerrupt from *irq7*, it reads out the port of the microcontroller where a new intentity is stored and sends the *sunIntensity* signal.
This component is contained only in the subproblem SunControl and need not to be merged.

Component WindSensorIAL I

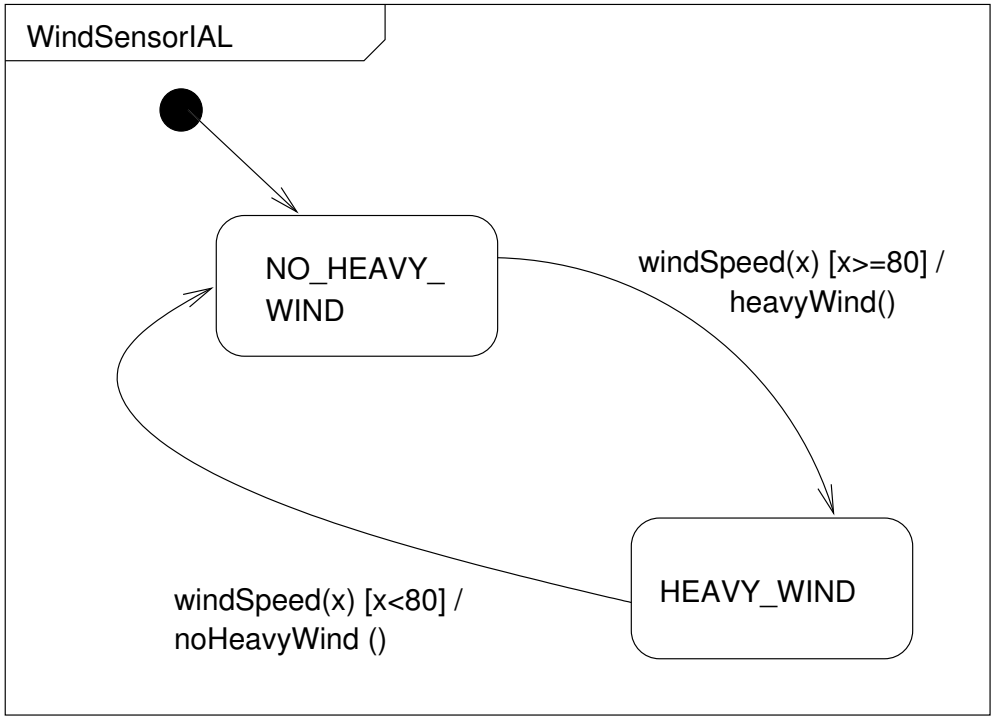
WindSensorIAL component overview



This component is the same for all subproblems and need not to be merged.

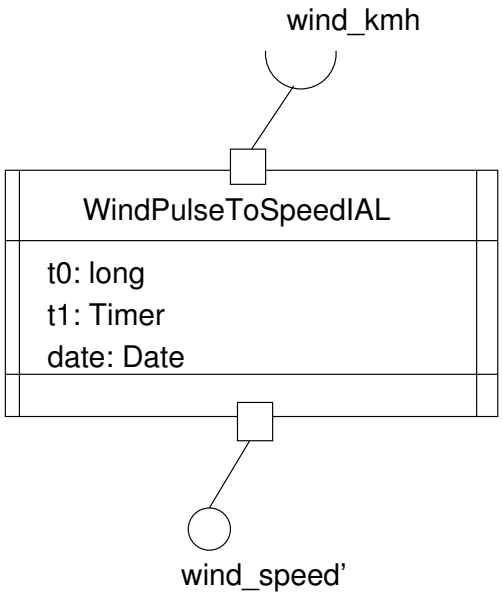
Component WindSensorIAL II

WindSensorIAL state machine



Component WindPulseToSpeedIAL I

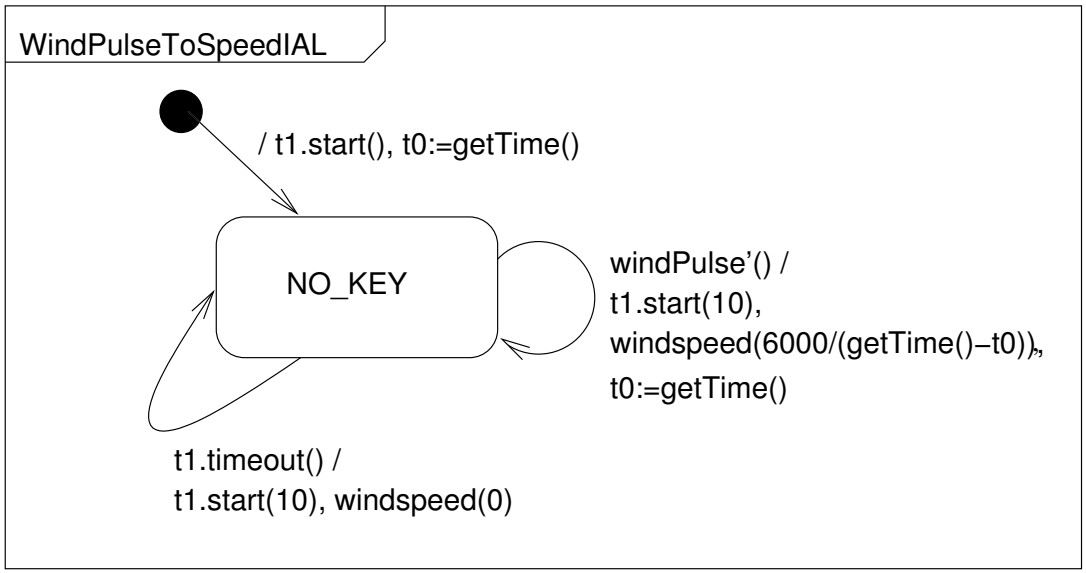
WindPulseToSpeedIAL component overview



This component is the same for all subproblems and need not to be merged.

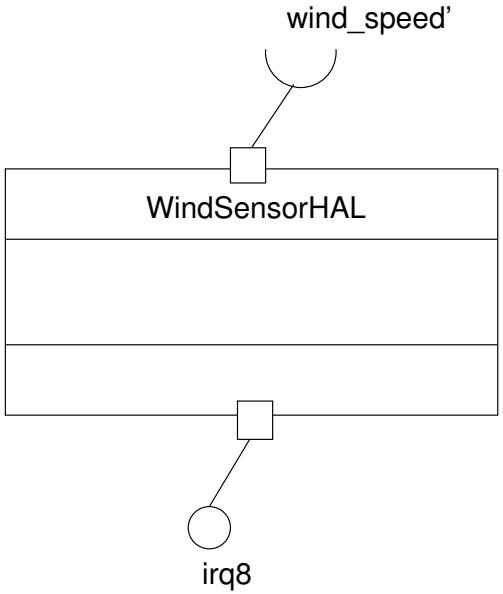
Component WindPulseToSpeedIAL II

WindPulseToSpeedIAL state machine



Component WindSensorHAL

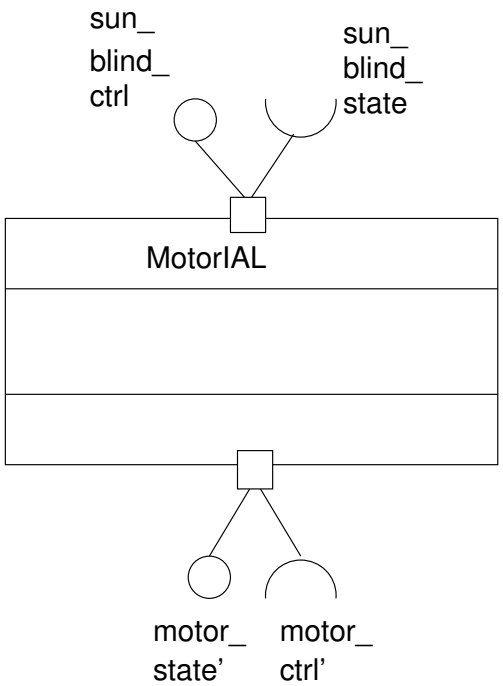
WindSensorHAL component overview



When this component receives the signal *interrupt_req_8()*, it emits a *windPulse* signal.
This component is the same for all subproblems and need not to be merged.

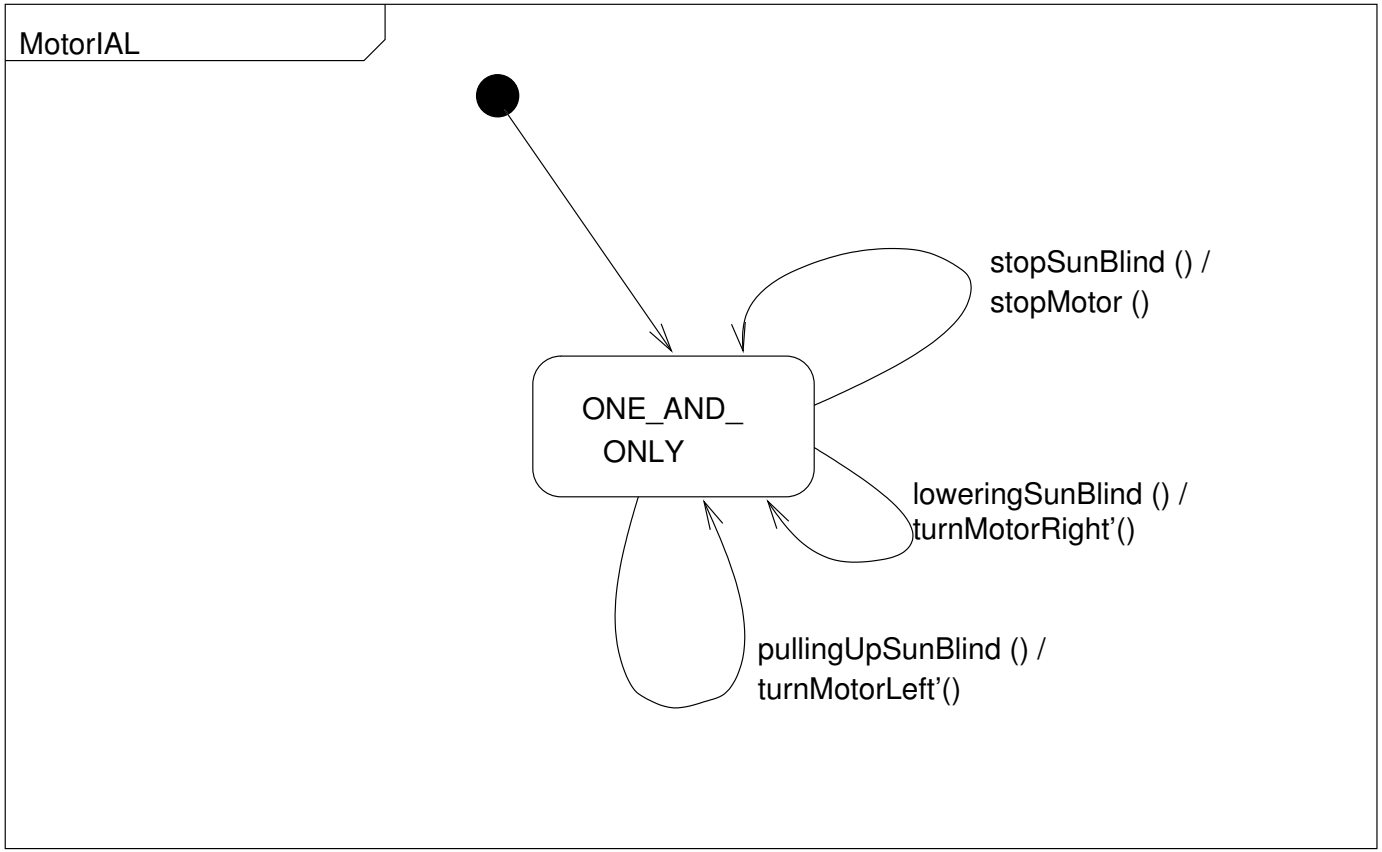
Component MotorIAL I

MotorIAL component overview



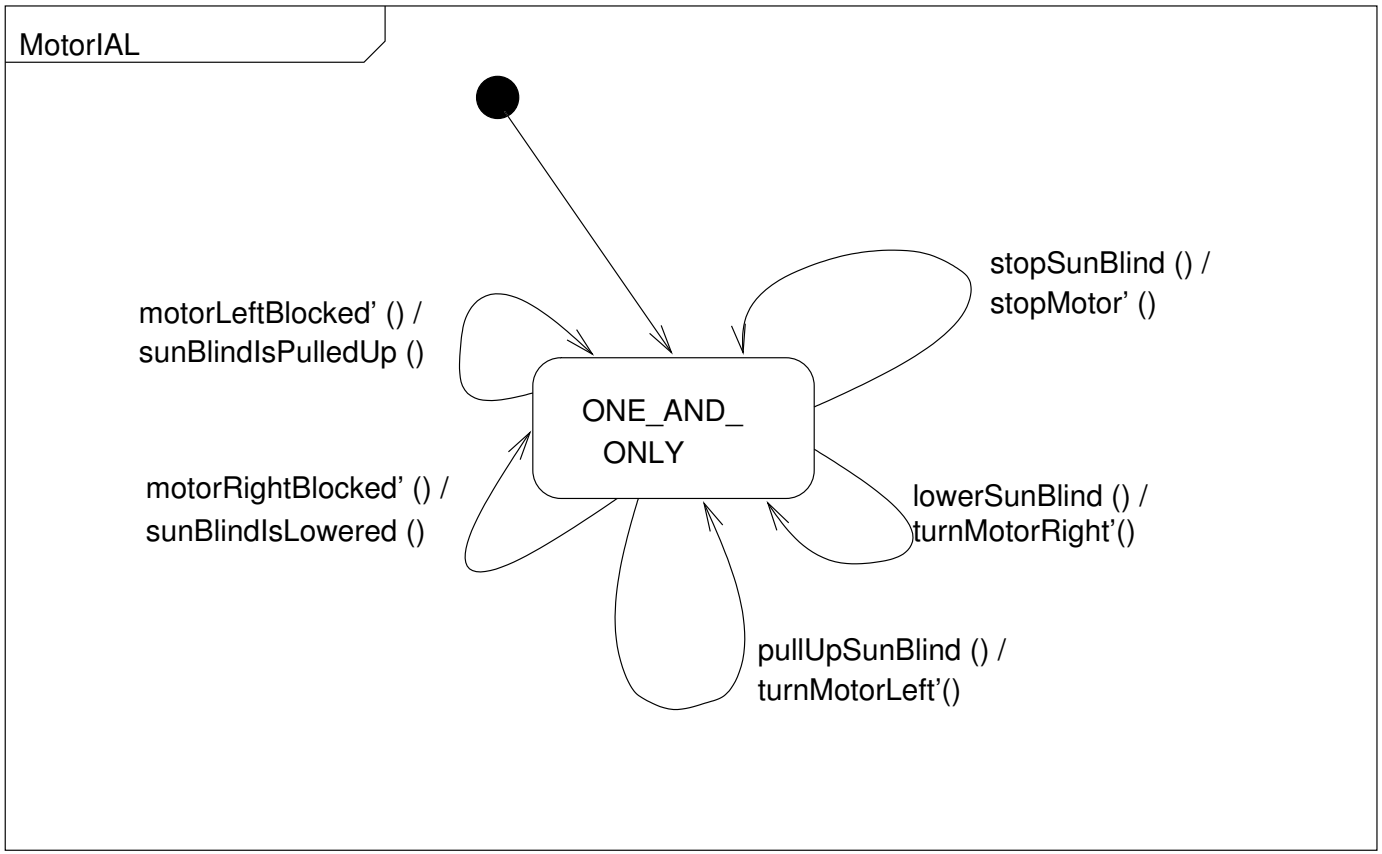
Component MotorIAL II

MotorIAL state machine for SunControl and UserControl



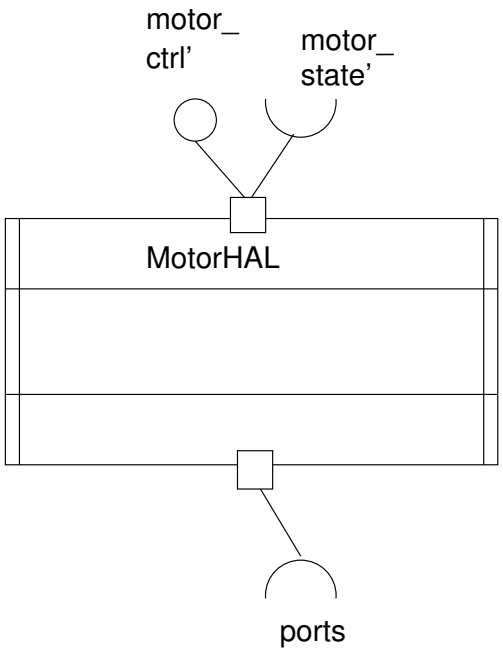
Component MotorIAL III

MotorIAL merged state machine (Signals for NoDesctructControl added)



Component MotorHAL I

MotorHAL component overview



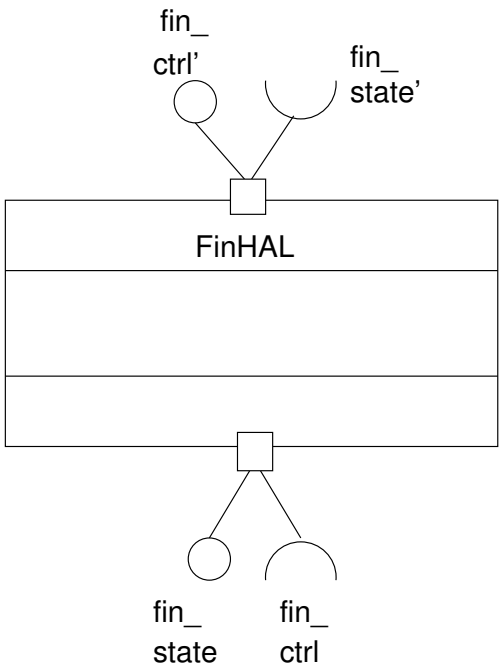
This component sets the ports to turn the motor left or right or to stop it. It would be a passive component for the subproblems SunControl, UserControl and FinsControl since it reacts immediately to input signals. But it has to be an active

Component MotorHAL II

component, since it also checks the port of the microcontroller representing the state of the motor every 10 ms and generates the *blocked* signals if the state of the port changes (subproblem NoDestructControl).

Component FinHAL

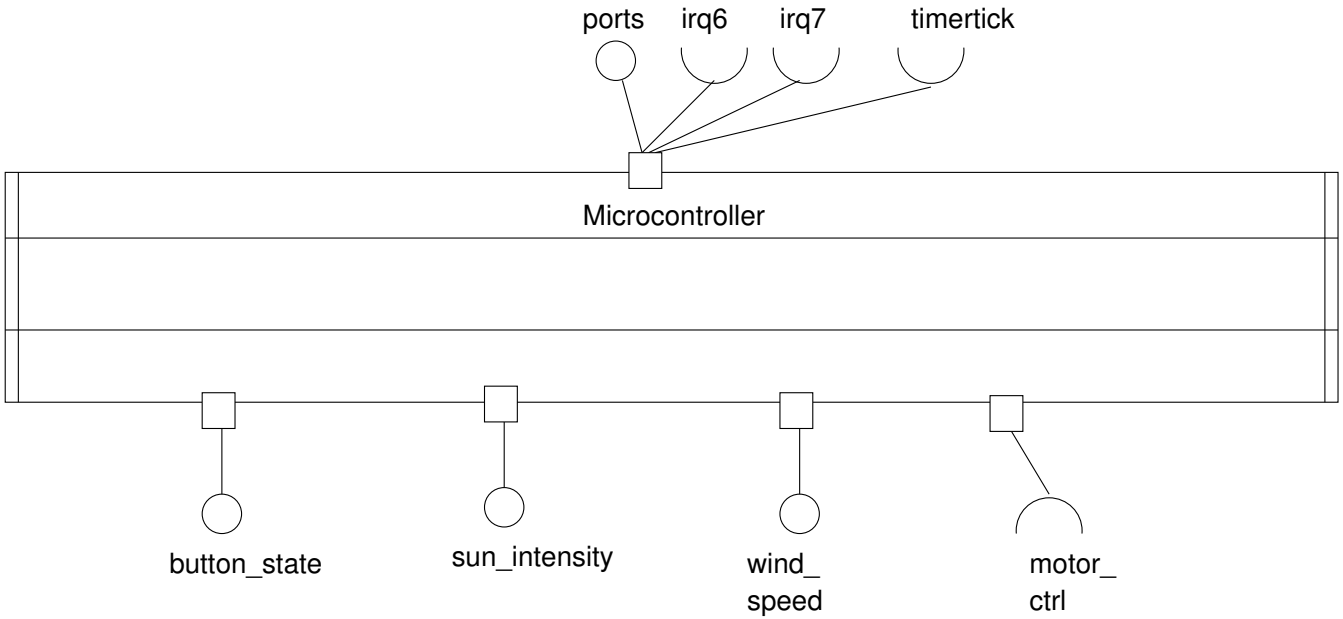
FinIAL component overview



This component sets the ports to turn the fins left or right.
This component is contained only in the subproblem FinsControl and need not to be merged.

Component Microcontroller

Microcontroller component overview



The *Microcontroller* is a reused existing component. In this Figure, only the interfaces relevant for the problem are shown.

Validation

- The state machines behave as described in the sequence diagrams of Step 8. All states are covered.
- The interface classes are the same as in Phase 7.
- SunDetection, ButtonIAL, and WindPulseToSpeedIAL are active classes, because they contain timers. SunBlindAppCtrl is active, because it contains the active class SunDetection. MotorHAL is an active component since it works in parallel with the other components.
- The state machines handle all possible signals in all states.

Embedded Systems

WS 08/09

Maritta Heisel

Maritta.Heisel(AT)uni-duisburg-essen.de

Denis.Hatebur(AT)uni-duisburg-essen.de

University Duisburg-Essen – Faculty of Engineering
Department of Computer Science
Workgroup Software Engineering

Overview of development process (DePES) I

1. Describe system in use
2. Describe system to be built
3. Decompose problem
4. Derive a machine behavior specification for each subproblem
5. Design global system architecture
6. Derive specifications for all components of the global system architecture
7. Design an architecture for all programmable components of the global system architecture that will be implemented in software

Overview of development process (DePES) II

8. Specify the behavior of all components of all software architectures, using sequence diagrams
9. Specify the software components of all software architectures as state machines
10. Implement software components and test environment
11. Integrate and test software components
12. Integrate and test hardware and software

Phase 10: Implement software components and test environment

...

7. Design a software architecture for all components of the global system architecture that should be implemented in software
8. Specify the behavior of all components of all software architectures, using sequence diagrams
9. Specify the software components of all software architectures as state machines
10. **Implement software components and test environment**
11. Integrate and test software components
12. Integrate and test hardware and software

Phase 10: Implement software components and test environment

input:	software component behavior from Phase 8	sequence diagrams with annotated states
	specification of merged components of Phase 9	*different notations*
output:	test software for software components	programming language or test language
	implemented software components	programming language
validation:	run tests	test results

Notations and concepts

- Java
- Implementation of modules
- Module tests

Constants

- in C++:

  ```
  #define X 2
  ```

  or `const int X = 2;`
- in Java: `final static int X = 2;`

Classes in Java

- Always one file for one class, no header files
- Name of constructor = name of class = name of file
- There is no destructor, the garbage collection frees memory of unused objects (e.g. for objects set to *null*).
- The entry point of an application is
  `public static void main (String[] args)`
  in one class (file).

Note: Java is case sensitive.

Objects in Java

- All objects that are not simple data types such as `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double` have to be created explicitly.
- For some simple data types predefined classes exist (e.g. `Integer` for `int`). These classes provide methods like, e.g., `toString`.
- Create an object `m` of the class `Integer`:
  `Integer m = new Integer(0);`
- Or: `Integer m; m = new Integer(0);`
- Attributes and methods declared as `static` belong to the *class* and are the same for all objects.
- `super` can be used to access functionality of the the superclass.
- `this` can be used to reference the object itself.

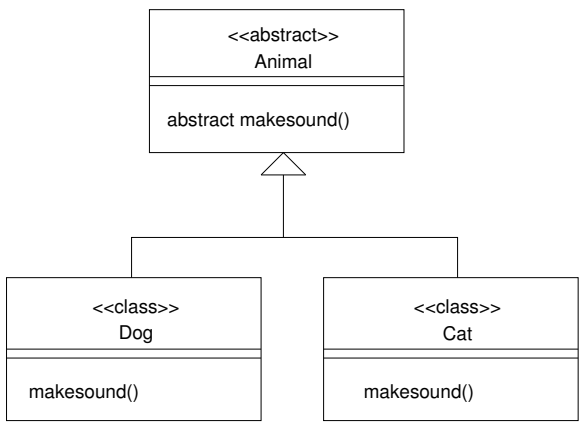
Classes: Example


```
public class MainInit {
     private Integer i;      // Attributes
     private String s;

     public MainInit() {     // Constructor
        s = new String();
        i = new Integer(7);
     }

     public static void main(String [] args) {
           MainInit m = new MainInit();
     }
}
```

Abstract Classes

- Methods that are not yet implemented can be defined as `abstract`.
- A class with at least one abstract method is an abstract class.
- Abstract classes are useful for inheritance.
- Java supports no multiple inheritance.
- An abstract class has to be *extended* by another class where the method has to be implemented (keyword: `extends`).
- It is not possible to create objects of an abstract class.

Abstract Classes: Example

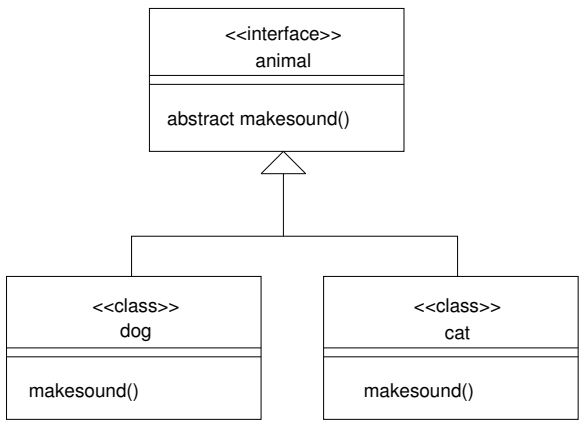



```
public abstract class Animal {
   public Animal(){}
   public abstract void makeSound();
}
public class Cat extends Animal {
   public Cat(){}
   public void makeSound()
   {System.out.println("miaow");}
}
public class Dog extends Animal {
   public Dog(){}
   public void makeSound()
   {System.out.println("woof-woof");}
}
public class Main{
 public static void main(String[] args) {
    Animal a1 = new Dog();
    Animal a2 = new Cat();
    a1.makeSound(); a2.makeSound();
 }
}
```

Interface Classes

- A class where all methods are abstract can be declared as an `interface`.
- A class can implement several interfaces (keyword `implements`), but it can only extend one other class.
- A class can be accessed using the interface.
- An interface class has no constructor.
- Solves problem of multiple inheritance.

Interface Classes: Example




```
public interface Animal {
   public void makeSound();
}
public class Cat implements Animal {
   public Cat(){}
   public void makeSound()
   {System.out.println("miaow");}
}
public class Dog implements Animal {
   public Dog(){}
   public void makeSound()
   {System.out.println("woof-woof");}
}
public class Main{
 public static void main(String[] args) {
    Animal a1 = new Dog();
    Animal a2 = new Cat();
    a1.makeSound(); a2.makeSound();
 }
}
```

Exceptions

- Methods can throw exceptions

```
public int div(int n1, int n2) throws Exception {
   if (n2==0) throw Exception;
   ...
}
```

- Exceptions can be handled with `try` and `catch`

```
try {
   a = div(b,c);
} catch (Exception e) {
   System.out.println(e); //println internally calls e.toString()
}
```

- Other exceptions are e.g. `IOException`, `ClassNotFoundException`, `ArithmeticException`, `OutOfMemoryError`, `Error`.

Assertions

- Assertions can be used to specify (and check) the pre-conditions and possibly post-conditions of a method.
- Assertions are defined using

  ```
  assert condition==true: "ErrorString";
  ```

- Assertions are introduced in Java 1.5, they are only evaluated using the execution switch “evaluate assertions”:

  ```
  java -ea
  ```

- Exceptions thrown by assertions cannot be caught.

Threads ($\triangleq$ Tasks with shared memory)

- Threads are necessary for active components.
- Systems with one processor use time slices to simulate parallelism for several threads.
- The started thread runs "in parallel" to the other parts of the software.
- Threads in Java support the following functionality:

`start`	starts the Thread
`interrupt`	stops the thread
`setName`	Assigns a name to the thread.
`sleep`	Waits the time given as a parameter in ms.
`yield`	Advises the scheduler to run another thread at this time.
`setPriority`	Assigns a priority to the thread. Threads with a higher priority get more execution time or they preempt the other tasks (depends on OS).

Example: Threads

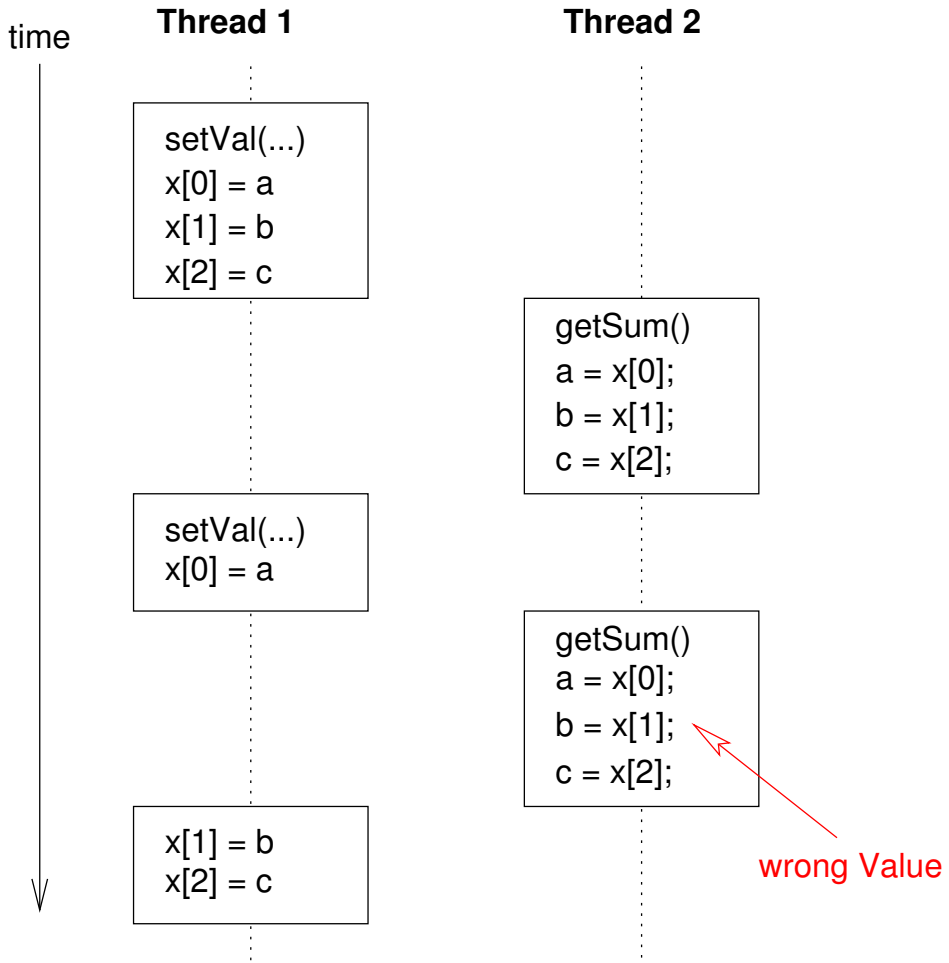

```
import java.lang.*;       // import library functionality
public class Clock extends Thread{
    private ms_clock clk;                    // Attributes
    public Clock(ms_clock call) {            // Constructor
        clk = call;
        this.start();
    }
    public void run () {                     // Thread
        while (true) {                       // endless Loop
            clk.MsClock();
            try {
                Thread.sleep(1);  //this.sleep(1) is also possible
            } catch ( Exception e ) { // when thread is interrupted
                System.out.println( e );  // using interrupt()
            }
        }
    }
}
```

Example: Threads with runnable interface (alternative way)


```
import java.lang.*;
public class Clock implements Runnable{
    private Thread clockThread;          // Attributes
    private ms_clock clk;
    public Clock(ms_clock call) {        // Constructor
        clk = call;
        clockThread = new Thread (this);// an object of type Thread
        clockThread.start();             //  must be created with the
    }                                    //  object of Clock as a param.
    public void run () {                 // Thread
        while (true) {                   // endless Loop
            clk.MsClock();
            try {
                Thread.sleep(1);
            } catch ( Exception e ) {
                System.out.println( e );
            }
        }
    }
}
```

Must be used if the class extends another class.

Problem: Concurrent access to variables



`getSum()` is executed before the array is filled completely.

Solution: Synchronize


```
public class SharedMemSum {
   private int[] x;
   public SharedMemSum () {
       x = new int [3] ;
       x[0]=0; x[1]=0; x[2]=0;
   }
   synchronized void setVal(int a, int b, int c) { //called from T. 1
       x[0] = a; x[1] = b; x[1] = c;
   }
   synchronized int getSum() { //called from Thread 2
       int a,b,c;
       a=x[0]; b=x[1]; c=x[2];
       return a+b+c;
   }
}
```

The statement `synchronized` makes a method non-interruptable.

Implementation of modules / software components I

- A module has provided and required interfaces that can be connected with other modules
- A module only uses functionality from its required interfaces, from the programming language, and a limited set of operations of the operating system (e.g., tasks, threads, memory allocation, timers, messages, synchronization mechanisms).
- Active components are implemented using threads.

Implementation of modules / software components II

- Interfaces can be implemented using
  - directly called methods,
  - methods in interface classes, or
  - messages, e.g.
    - delegates in *C♯*,
    - events in Java,
    - signals and slots in C++ with the Qt-library,
    - messages in the Windows API.

Messages usually allow asynchronous communication (with queuing) and in some frameworks multiple consumers are allowed.

Implementation of interfaces I

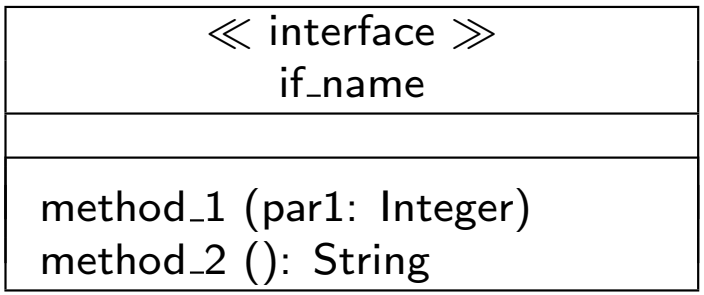
Loose and tight coupling is possible for interfaces between components.

- For tight coupling methods are called directly.
- Tight coupling of objects can only be implemented if the called object is created by the calling class.
- Tight coupling usually needs less resources and can also be implemented using non-oo languages.
- For loose coupling (methods in interface classes) the objects are created by a different class. The object that uses an interface of another object needs a reference to the used object. The reference can be provided in the constructor or in an additional method to connect the objects.

Implementation of interfaces II

- With loose coupling, the object can be implemented independently of its environment. The object becomes a component. Messages also implement a loose coupling. Loose coupling is better for the implementation of module tests.

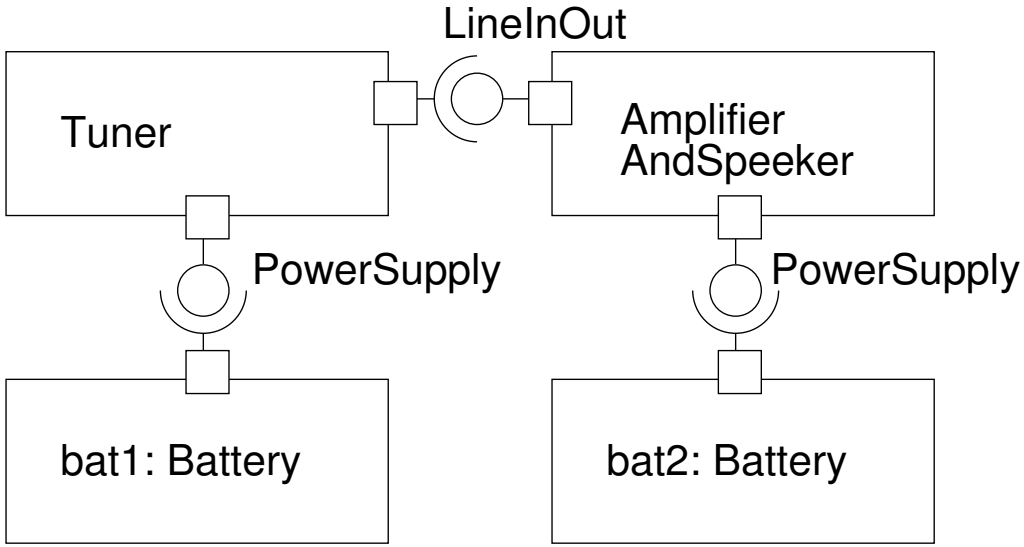
Implementation of interfaces in Java




```
package project_name;
public interface if_name {
    public void method_1 (int par1);
    public String method_2 ();
}
```

The project name should be added as a package. Otherwise additional parameters are necessary to compile the project.
Note: `int` is a simple data type and and `String` is a class.

Implementation of provided interfaces in Java I



Each provided interface is defined as a interface class, e.g.:

```
public interface LineInOut {
   public void transmitMusic();
}

public interface PowerSupply {
   public void powerOn();
}
```

Implementation of provided interfaces in Java II

A component can implement / provide several interfaces, e.g.:

```
public class AmplifierAndSpeeker implements
                              LineInOut, PowerSupply  {
   public AmplifierAndSpeeker (){} //constructor

   public void transmitMusic() { Play;}
   public void powerOn() { Action2;}
}
```

All provided operations must be implemented as methods.

Implementation of required interfaces in Java I

A component can use / require several interfaces, defined as interface classes.

```
public class Tuner implements PowerSupply {
   private LineInOut outputDevice;
   public Tuner(){ outputDevice = NULL; }
   public void connectTo(LineInOut par) {outputDevice = par;}

   public void powerOn() {
     while (true) {
        if (outputDevice!=NULL) outputDevice.transmitMusic();
   }
}
```

Implementation of required interfaces in Java II

- The required interfaces become private attributes (`outputDevive` of type `LineInOut`).
- The component has to provide a method to connect the component to the required component (`connectTo`). In this connect method, the private attributes is initialized.
- Using this private attribute, the connected component can be used. It should only be used if it is initialized (`if (outputDevice!=NULL) ...`).

Alternatively, it is possible to leave out the method `connectTo` and initialize the connected interface in the constructor.

- The component `Tuner` also provides the interface `PowerSupply` and implements the method `powerOn`.

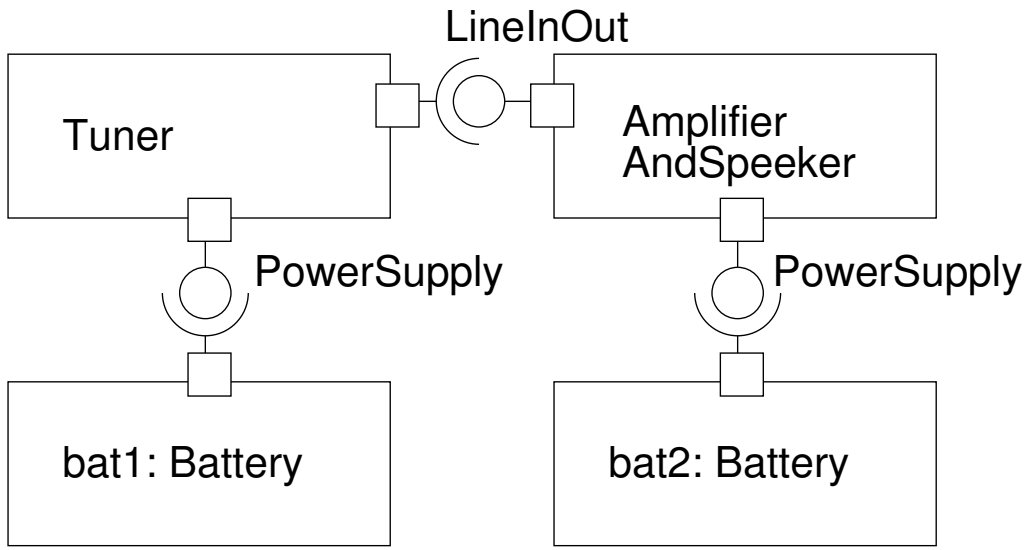
Implementation of required interfaces in Java III


```
public class Battery {
   public Battery(){ suppliedDevice=NULL }
   private PowerSupply suppliedDevice;
   public void connectTo(PowerSupply suppliedDevice ) {
suppliedDevice.powerOn();
   }
}
```

The component `Battery` powers on the supplied device when connected. It requires the interface `PowerSupply`.

Implementation of required interfaces in Java IV

The components *bat1*, *bat2*, *myTuner*, and *myAmp* can be connected as follows:



Implementation of required interfaces in Java V

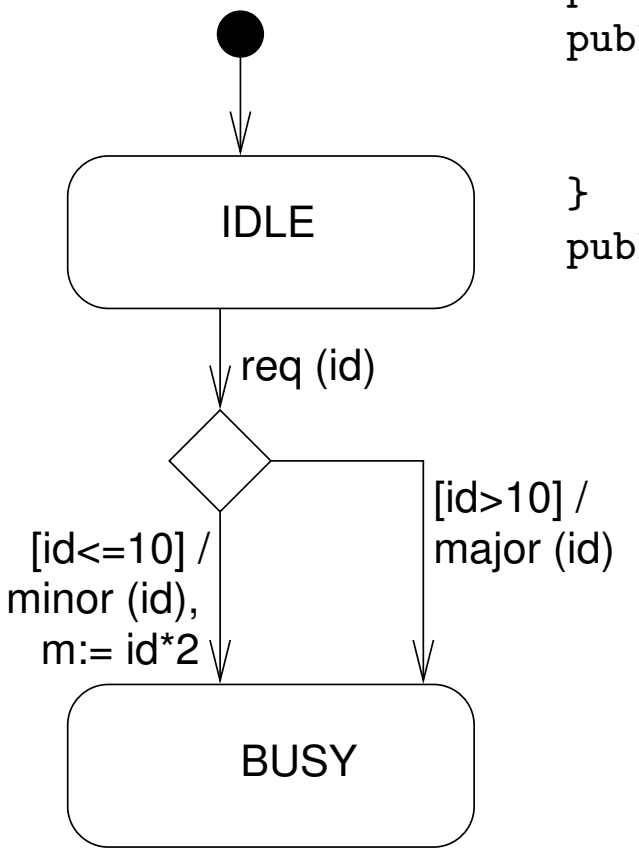

```
AmplifierAndSpeeker myAmp = new AmplifierAndSpeeker();
Tuner myTuner = new Tuner();
Battery bat1 = new Battery();
Battery bat2 = new Battery()

myTumer.connectTo(myAmp);
bat1.connectTo(myTuner);
bat2.connectTo(myAmp);
```

Implementation of state machines in Java I

Can be implemented in different ways; here just one alternative is shown.

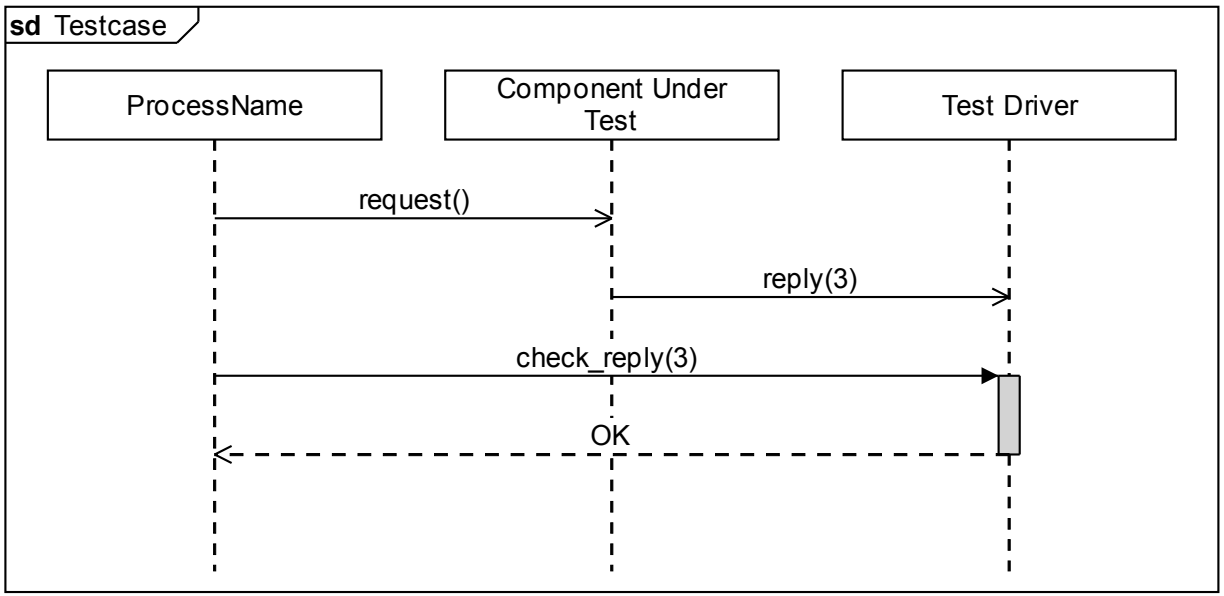
- Define a constant for each state.
- Set the initial state in the constructor.
- Create a method for each incoming signal.
- Add a case distinction containing all possible states to this method.
- In each case emit the specified output signals and set the new state of the state machine.
- Add appropriate handling for the parameters of the signals.
- Add a default case.

Implementation of state machines in Java II




```
public class ComponentName implements provided_if {
    static final int IDLE = 0, BUSY = 1;
    private int state;
    private req_if ri;   private m;
    public ComponentName (req_if ri_) {
        state = IDLE // Init state
        ri = ri_;
    }
    public void req(int id) {
        switch (state) {
            case IDLE:
                if (id<=10) {
                   if (ri!=NULL) ri.minor (id);
                   m = id*2; state = BUSY;
                } else {
                   if (ri!=NULL) ri.major (id);
                   state = BUSY;
                }
                break;
            default:
                assert false: "FSM error Req";
        }
    } ...
}
```

Module Tests I



Sequence diagrams can be transformed into test cases.

- A signal to the module can be implemented as a method call to the module (must belong to a provided interface).

Module Tests II

- For each required interface, a test driver that provides the required interfaces is necessary. It has to store the signals and the parameters called from the module to be tested. The test driver also needs an interface to check which methods (with which parameters) were called.
- Each signal in the sequence diagram from the tested module will be implemented as a command to the driver to check which methods (with which parameters) were called.
- If a timing constraint is given, it must be checked that the signal (method call) occurred within the given interval (not before and not too late).
- If a sequence diagram contains a combined fragment *ALT*, then for each alternative a test case must be implemented.

Module Tests III

- If the sequence diagram contains variables, suitable sample values must be used for testing. Try to use also values at the limit of the range and out of the range.
- If a sequence diagram describes a typical interactions, also implement test cases for similar interactions and exceptional behavior. (In this case you have to check the state machines to find out the expected behavior.)

Phase 10: Implement software components and test environment

input:	software component behavior from Phase 8	sequence diagrams with annotated states
	specification of merged components of Phase 9	*different notations*
output:	test software for software components	programming language or test language
	implemented software components	programming language
validation:	run tests	test results

Executing Phase 10

In general, the procedure to implement and test software components using an object oriented programming languages can be described as follows:

1. Create interface classes for all internal interfaces (also for subcomponents).
2. Implement test drivers and cases for all components (except HAL) according to the sequence diagrams of Phase 8.
3. Create classes for all (sub-)components and implement them.
   - Implement actions as private methods.
   - Implement the state machine.
   - Implement the active classes with threads.
   - Check all classes if there is a concurrent access to complex variables and resolve this problem with the `synchronized` statement.
4. Run test cases.

Remarks I

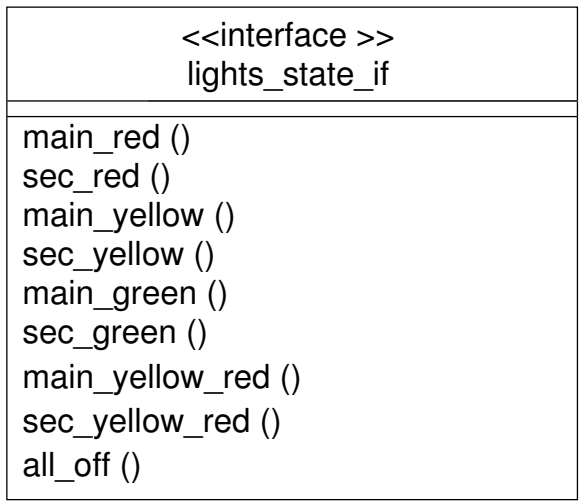
- Only the software components are implemented in this phase. They will be connected / integrated in the next phase.
- The validation of this phase is performed by running the test cases (unit test).
- The HAL is difficult to test because the hardware is directly connected to the HAL. Therefore, manual tests using measurement equipment and debugging tools should be performed.
- Real-time functionality must be tested in an emulator.

Remarks II

- The software components are implemented using some simple heuristics. For embedded systems, usually a static connection between components is established. The connectors in the composite structure diagrams can be implemented e.g. as data streams, method calls, asynchronous messages, or hardware access.
- This development process allows developing statically linked software components with the possibility of reuse.

Example 1: traffic light control

Create interface classes

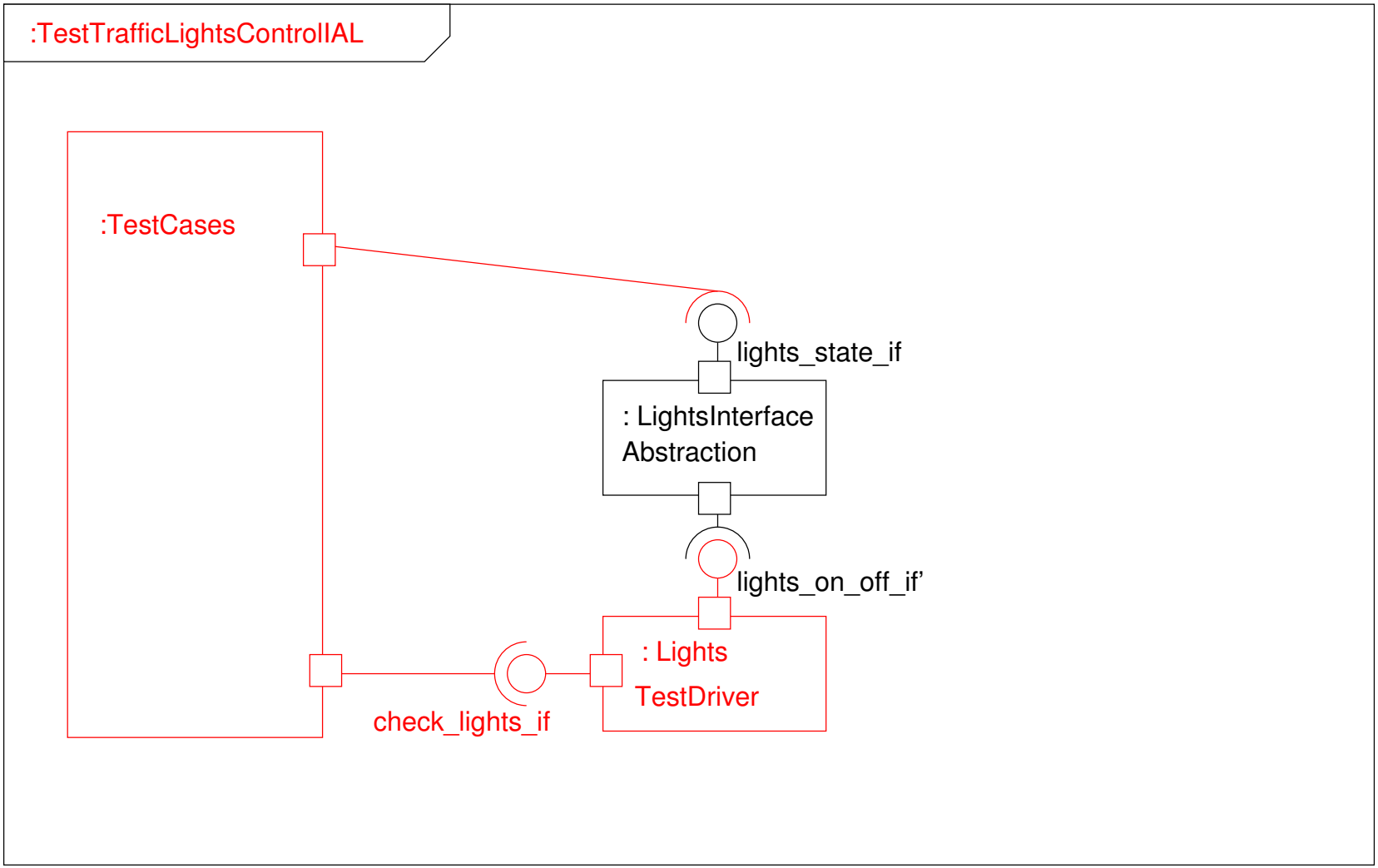



```
package tlc;
public interface lights_state_if {
    public void main_red();
    public void sec_red();
    public void main_yellow();
    public void sec_yellow();
    public void main_red_yellow();
    public void sec_red_yellow();
    public void main_green();
    public void sec_green();
    public void all_off();
}
```

Implement test environment

- The test environment consists of the test drivers for all required interfaces and the test cases.
- The test drivers stores the called methods and their parameters.
- The test drivers provides an interface to check the stored methods and parameters.
- Tests should be performed for all components, except HAL.
- The Hardware Abstraction Layer should be tested manually, because you need hardware to test them.

Test environment for LightsInterfaceAbstraction

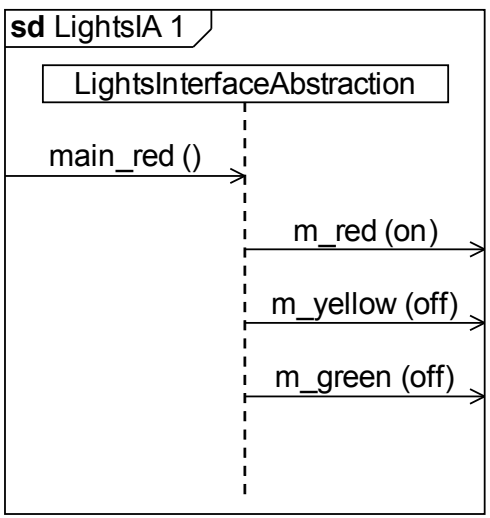


Test cases according Phase 8.

Implementation of LightsTestDriver for lights_on_off_if′

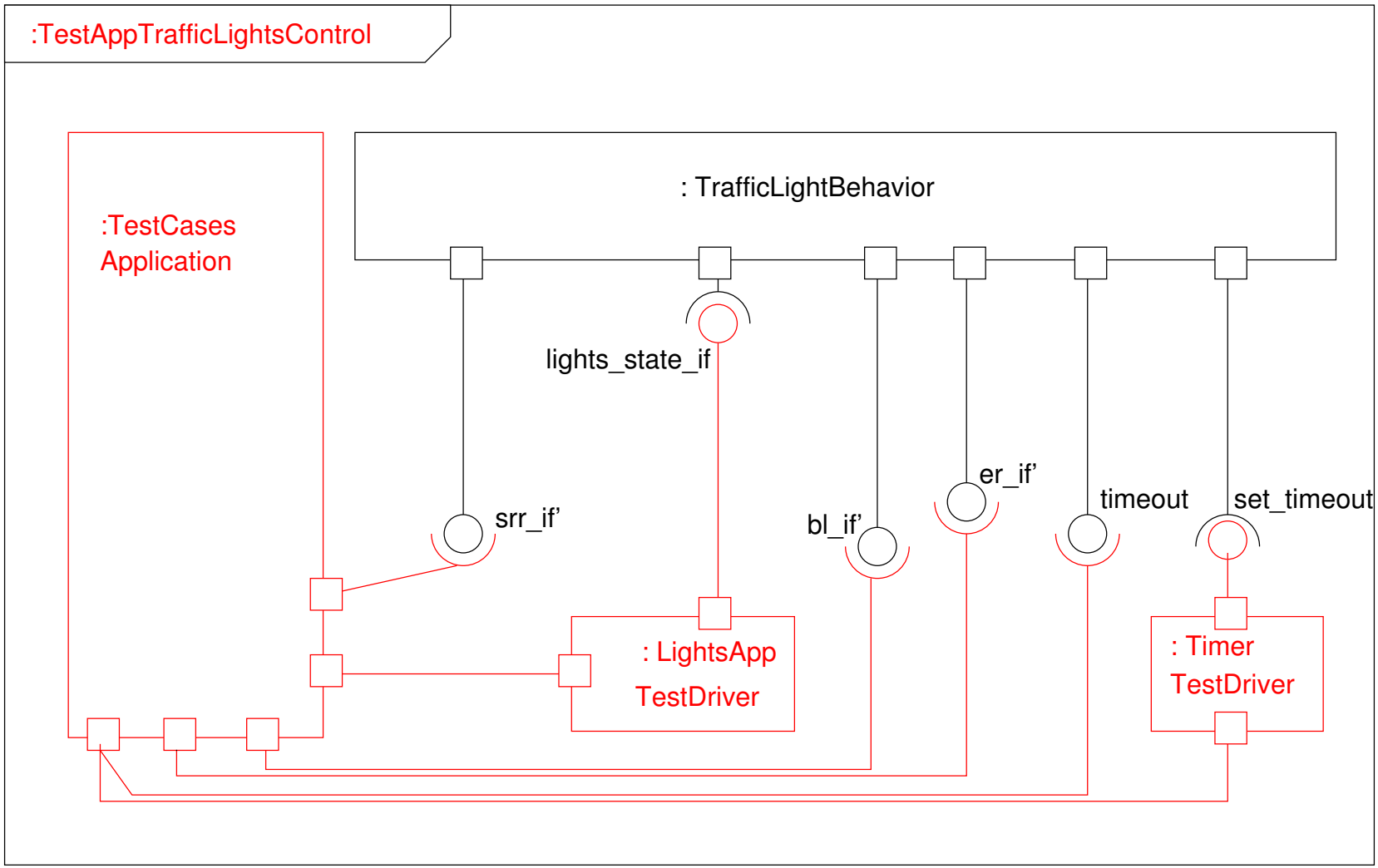

```
class LightsTestDriver implements lights_on_off_if_ {
    private boolean _m_red = false;
    private boolean _s_red = false;
    private boolean _m_yellow = false;
    private boolean _s_yellow = false;
    private boolean _m_green = false;
    private boolean _s_green = false;

    public void m_red (boolean x) { _m_red = x;}
    public void s_red (boolean x) { _s_red = x;}
    public void m_yellow (boolean x) { _m_yellow = x;}
    public void s_yellow (boolean x) { _s_yellow = x;}
    [...]

    public boolean checkColor(boolean mr, sr, my, sy, mg, sg) {
        return ( (_m_red == mr) && (_s_red == sr) &&
                 (_m_yellow == my) && (_s_yellow == sy) &&
                 (_m_green == mg) && (_s_green == sg) ) ;
    }
}
```

Implementation of test cases for LightsInterfaceAbstraction




```
package tlc;
import junit.framework.TestCase;
public class LightsInterfaceAbstractionTest extends
                                                TestCase {
    LightsInterfaceAbstraction lia;
    LightsTestDriver ltd;
    public void testInit() {
           // Initialize the test environment and the SUT
        ltd = new LightsTestDriver();
        lia = new LightsInterfaceAbstraction(ldt);
           // check that all lights are off ofter init
        assertTrue("one light is on", ltd.checkColor
          (false, false, false, false, false, false));
    }
    public void test_lightsia_1() {
        // send input signal
        lia.main_red();
        // checks result using the test driver
        assertTrue("not only m_red on", ltd.checkColor
          (true, false, false, false, false, false));
    }
    ...
}
```

Test environment for the application component



Test cases according to Phase 8.

Implementation of test driver for set_timeout

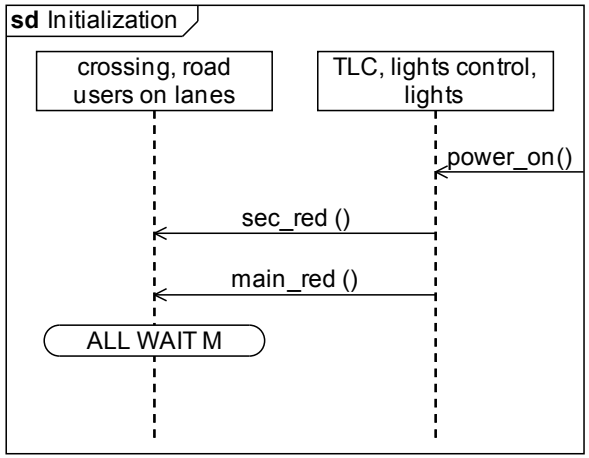

```
package tlc;
class TimerTestDriver implements set_timeout {

  int sec = -1;
  public void SetTimeOut(int seconds) {
    sec = seconds;
  }
  public boolean checkSetTimeOut(int second) {
    boolean ret = (sec == second); sec = -1;
    return ret;
  }
}
```

Reset of *sec* to enable two consecutive checks with same value.

Implementation of LightsTestDriver for lights_state_if


```
class LightsAppTestDriver implements lights_state_if {
    int color = 0;
    public final static int M_R = 1;
    public final static int S_R = 2;
    public final static int M_RY = 3;
    [...]
    public final static int ALL_OFF = 9;

    public void main_red(){ color = M_R; }
    public void sec_red(){color = S_R; }
    public void main_yellow(){color = M_Y; }
    [...]
    public void all_off(){color = ALL_OFF;}

    public boolean checkColor(int colorNr) {
        boolean ret = (colorNr == color);
        color = 0;
        return ret;
    }
}
```

Implementation of test cases for TrafficLightBehavior I



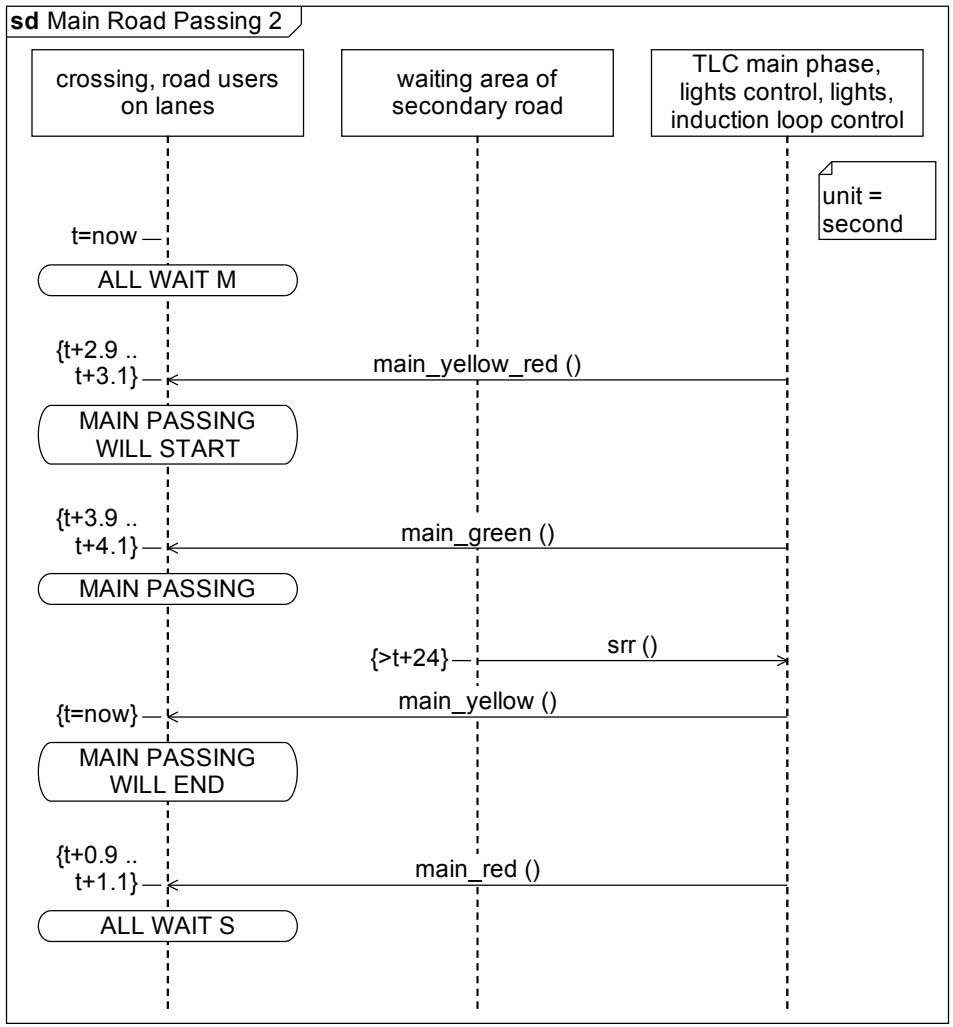

```
package tlc;
import junit.framework.TestCase;
public class TrafficLightBehaviorTest  extends TestCase {
    TrafficLightBehavior tlb;
    LightsAppTestDriver lia;
    TimerTestDriver tot;

    public void testInitialization() {
      // Initialize the test environment and the SUT
              (System unter test).
```

Implementation of test cases for TrafficLightBehavior II


```
        tlb = new TrafficLightBehavior();
        lia = new LightsAppTestDriver();
        tot = new TimerTestDriver();
        tlb.ConnectTo(tot, lia);
      // checks lights using the test driver
        assertTrue("main_red not set", lia.checkColor(lia.M_R));
    }
    ...
}
```

Only *main_red* is checked due to limitation of the test driver since only the last called is stored.

Implementation of test cases for TrafficLightBehavior III



Implementation of test cases for TrafficLightBehavior IV

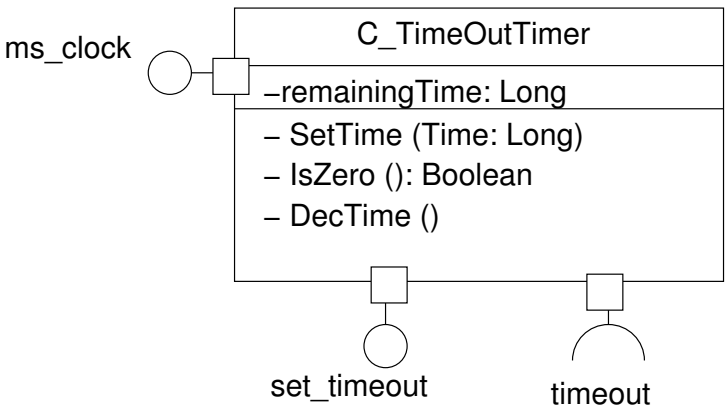

```
package tlc;
import junit.framework.TestCase;
public class TrafficLightBehaviorTest  extends TestCase {
    ...
    public void testMainRoadPassing2() {
        // ALL_WAIT_M
        // simulate elapsed timer
         tlb.Timeout();
        // checks result using the test driver
         assertTrue("main_red_yellow not set", lia.checkColor(lia.M_RY))
        // checks timer setting using the test driver
         assertTrue("timeout wrong", tot.checkSetTimeOut(1));
        // simulate elapsed timer
         tlb.Timeout();
        // checks result using the test driver
         assertTrue("main_green not set", lia.checkColor(lia.M_G));
        // checks timer setting using the test driver
         assertTrue("timeout wrong", tot.checkSetTimeOut(20));
        // MAIN_PASSING
        // simulate elapsed timer
         tlb.Timeout();
```

Implementation of test cases for TrafficLightBehavior V


```
        // sends directly a signals to the provided interfaces
         tlb.srr()
        // checks result using the test driver
         assertTrue("main_yellow not set", lia.checkColor(lia.M_Y));
        // checks timer setting using the test driver
         assertTrue("timeout wrong", tot.checkSetTimeOut(1));
        // MAIN_PASSING_WILL_END
        // simulate elapsed timer
         tlb.Timeout();
        // checks result using the test driver
          assertTrue("main_RED not set", lia.checkColor(lia.M_r));
        // ALL_WAIT_S
    }
    ...
}
```

The test runs faster than reality.

Create classes for all (sub-)components




```
package tlc;
public class TimeOutTimer implements
                ms_clock, set_timeout {
    private timeout to;
    private long remaining_time = 0;

    public TimeOutTimer(timeout timeout_par)
    {
        to = timeout_par;
    }
    public void SetTimeOut(int seconds) {}
    public void MsClock() {}
    private void SetTime(long time) {}
    private boolean IsZero() {}
    private void DecTime() {}
}
```

All methods of all provided interfaces have to be implemented (here: *SetTimeOut*, *MsClock*). An empty function body ({}) is used to avoid error messages of the compiler.

Implement actions as private methods

IsZero()

pre $true$

post $Result = true \Leftrightarrow remaining\_time = 0$

SetTime(time)

pre $time \geq 0$

post $remaining\_time = time$

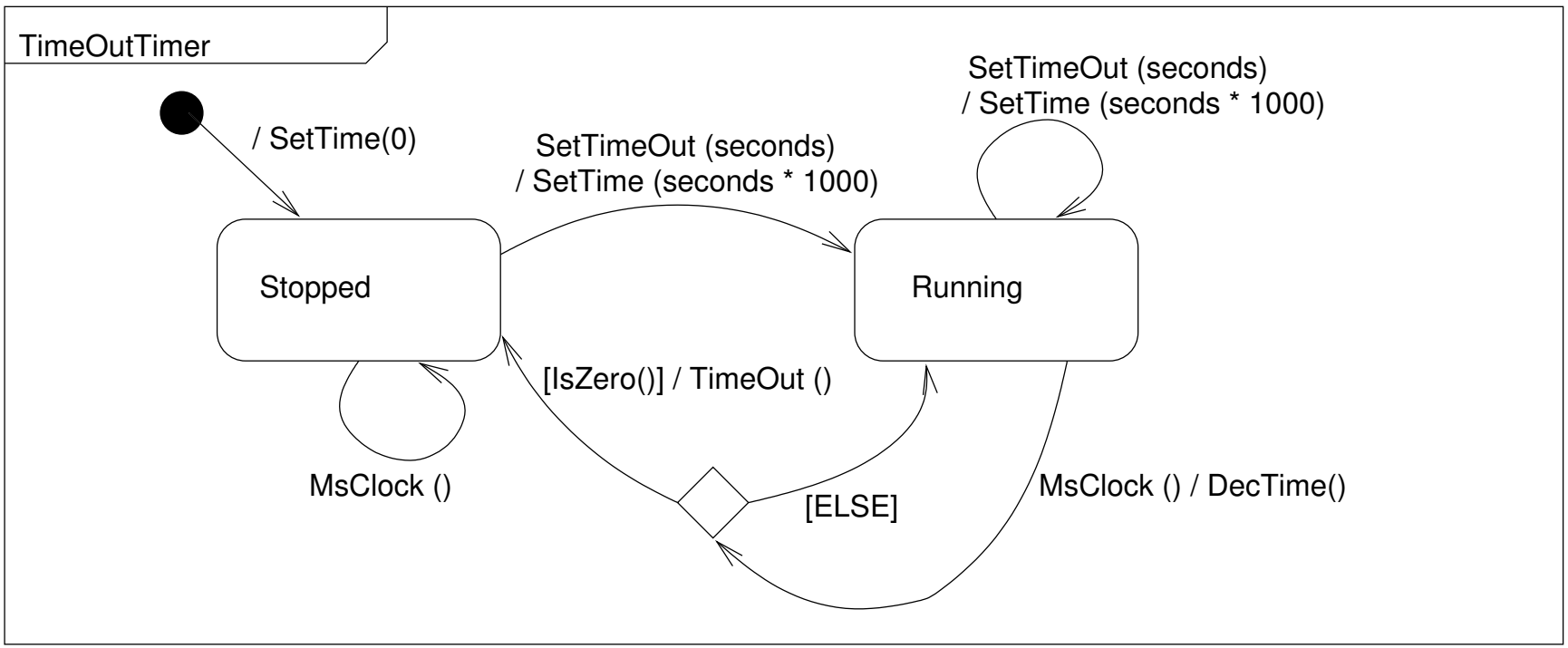
DecTime()

pre $remaining\_time \neq 0$

post $remaining\_time = remaining\_time@pre - 1$

```
package tlc;
public class TimeOutTimer implements
                ms_clock, set_timeout {
    ...
    private long remaining_time = 0;
    ...
    private boolean IsZero() {
        return (remaining_time == 0);
    }
    private void SetTime(long time) {
        assert (time >= 0): "PRE: SetTime";
        remaining_time = time;
    }
    private void DecTime() {
        assert remaining_time!=0:
               "PRE: DecTime";
        remaining_time = remaining_time - 1 ;
    }
}
```

Implement the state machine as public methods I



Implement the state machine as public methods II

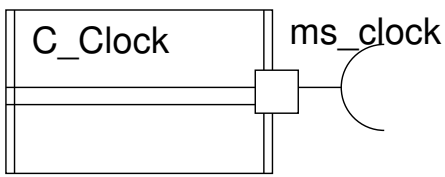

```
package tlc;
public class TimeOutTimer implements ms_clock, set_timeout {
    static final int STOPPED = 0;
    static final int RUNNING = 1;
    private int state;
    ...
    public TimeOutTimer(timeout timeout_par) {
        ...
        SetTime(0); state = STOPPED;
    }
    public void SetTimeOut(int seconds) {
        switch (state) {
            case STOPPED:
                SetTime(seconds*1000); state = RUNNING; break;
            case RUNNING:
                SetTime(seconds*1000); break;
            default:
                assert false: "FSM error TimeOutTimer.SetTimeOut";
        }
    } ...
```

Do not forget the *break*-statement. Otherwise, the next case will also be executed.

Implement the state machine as public methods III


```
    ...
    public void MsClock() {
        switch (state) {
            case STOPPED:           // do nothing
                break;
            case RUNNING:
                DecTime();
                if (IsZero()) {
                    state = STOPPED;
                    to.Timeout();  // external interface
                }                   // else: do nothing
                break;
            default:
                assert false: "FSM error TimeOutTimer.MsClock";
        }
    } ...
}
```

Implement the active classes with threads




```
package tlc;
import java.lang.*;
public class Clock extends Thread{
    private ms_clock clk;
    public Clock(ms_clock call) {
        clk = call;
        this.start();
    }
    public void run () {
        while (true) {
            clk.MsClock();
            try {
                Thread.sleep(1);
            } catch ( Exception e ) {
                System.out.println( e );
            }
        }
    }
}
```

Check for concurrent access

For all classes: Check what methods of one class are called from different threads.

- No *synchronized* statement is necessary because no complex attributes are shared between methods called by different threads.

Validation: run tests

Output of JUnit test environment:

- Test result with one error:

```
Testsuite: tlc.TrafficLightBehaviorTest
Tests run: 33, Failures: 1, Errors: 0, Time elapsed: 0,217 sec
Testcase: testInit(tlc.TrafficLightBehaviorTest): FAILED
main_red not set
junit.framework.AssertionFailedError: main_red not set
    at tlc.TrafficLightBehaviorTest.testInit(TrafficLightBehaviorTest.java:103)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:
Test tlc.TrafficLightBehaviorTest FAILED
```

- Test result with no errors:

```
Testsuite: tlc.TrafficLightBehaviorTest
Tests run: 33, Failures: 0, Errors: 0, Time elapsed: 0,213 sec
```

Example 2: sun blind control

Implementation of SunBlindContoller

See Netbeans-Project on http://swe.uni-due.de.

Phase 11: Implement software components and test environment

...

7. Design a software architecture for all components of the global system architecture that should be implemented in software
8. Specify the behavior of all components of all software architectures, using sequence diagrams
9. Specify the software components of all software architectures as state machines
10. Implement software components and test environment
11. **Integrate and test software components**
12. Integrate and test hardware and software

Phase 11: Integrate and test software components

input:	global software architecture from Phase 7	composite structure diagrams
	software behavior from Phase 6	sequence diagrams with annotated states
	implemented software components from Phase 10	programming language
output:	implemented software	programming language
	test software for integrated software	programming language or test language
validation:	run tests	test results

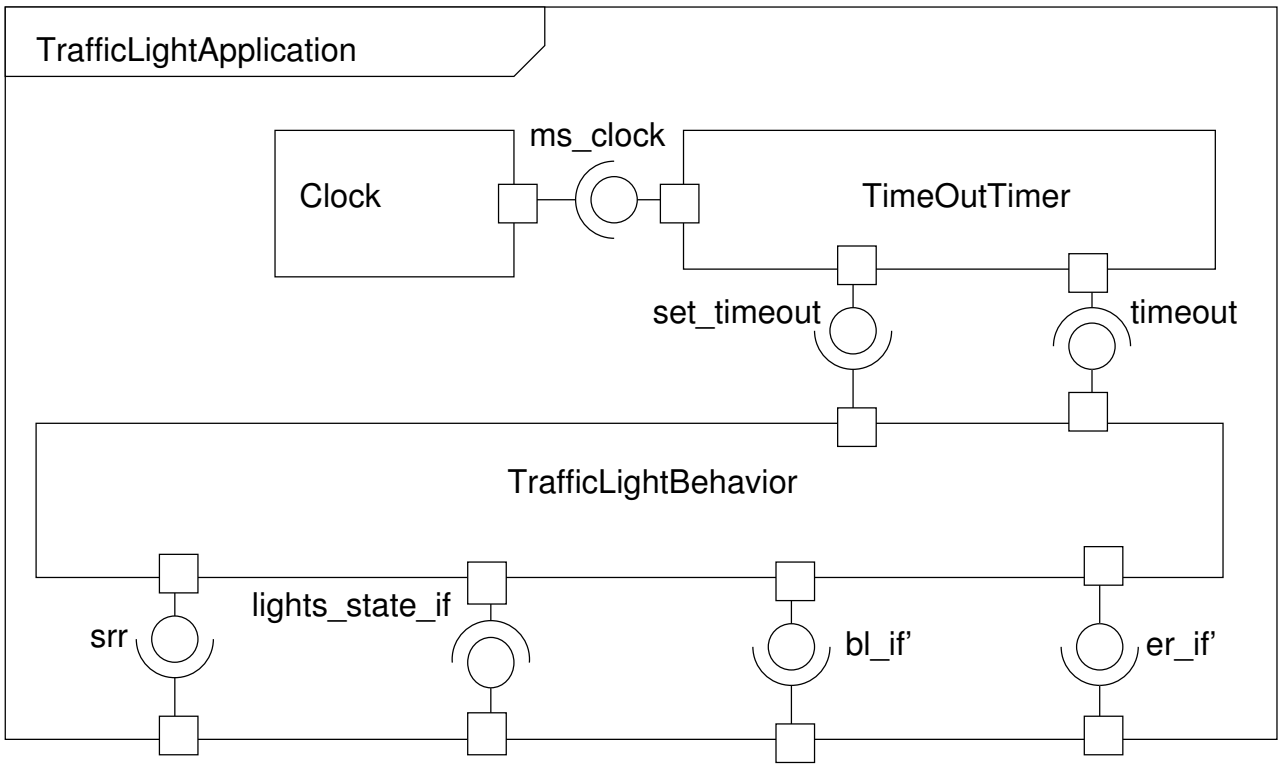
Executing Phase 11

In general, the procedure to compose the software out of components using an object oriented programming languages can be described as follows:

1. Create a class for each subcomponent containing other components to initialize all objects according to the subcomponent-architecture.
2. Create a class `MainInit` to initialize all objects according to the architecture from Phase 7.
3. Implement test cases for the connected components (except HAL) according to the sequence diagrams of Phase 6.
4. Run test cases.

Example 1: traffic light control

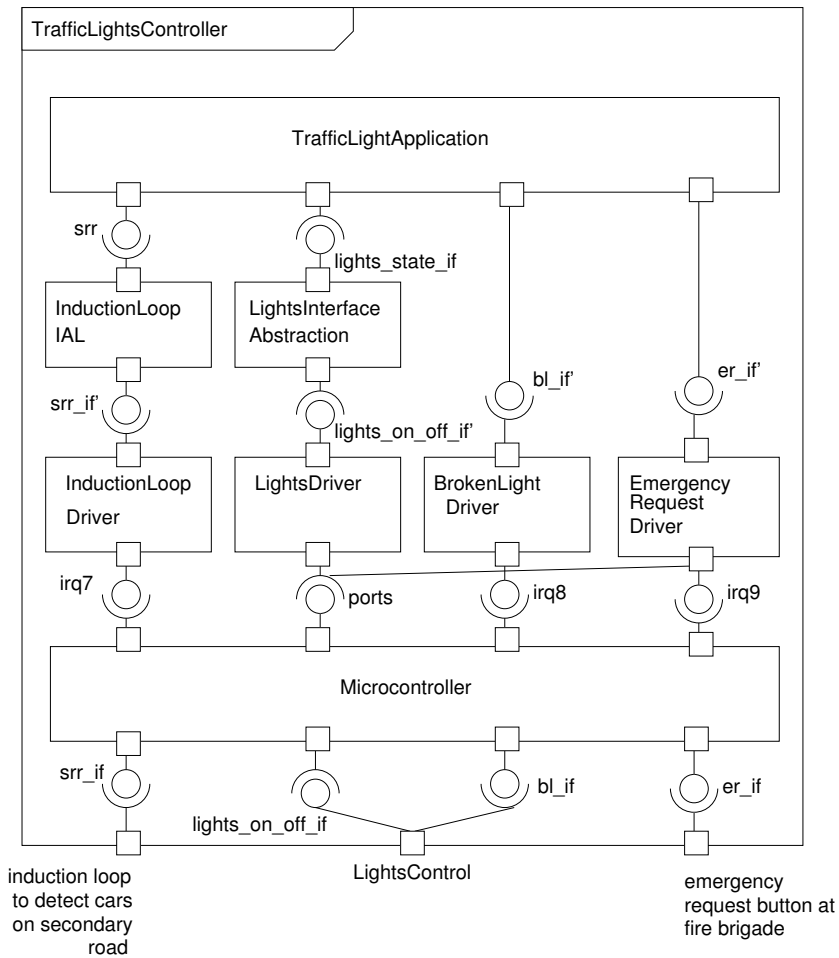
Create class containing all components of TrafficLightsApplication I



Create class containing all components of TrafficLightsApplication II


```
package tlc;
public class TrafficLightApplication {
    // declare all components
    private TrafficLightBehavior tlb;
    private TimeOutTimer tot;
    private Clock clk;

    public TrafficLightApplication () { // constructor
        // create all components, connect where possible
        tlb = new TrafficLightBehavior();
        tot = new TimeOutTimer(tlb);
        clk = new Clock(tot);
    }

    public void connectTo(lights_state_if lsi) {
        // connect remaining interfaces
        tlb.connectTo(tot, lsi);
    }
}
```

Create main class containing all components I



Create main class containing all components II

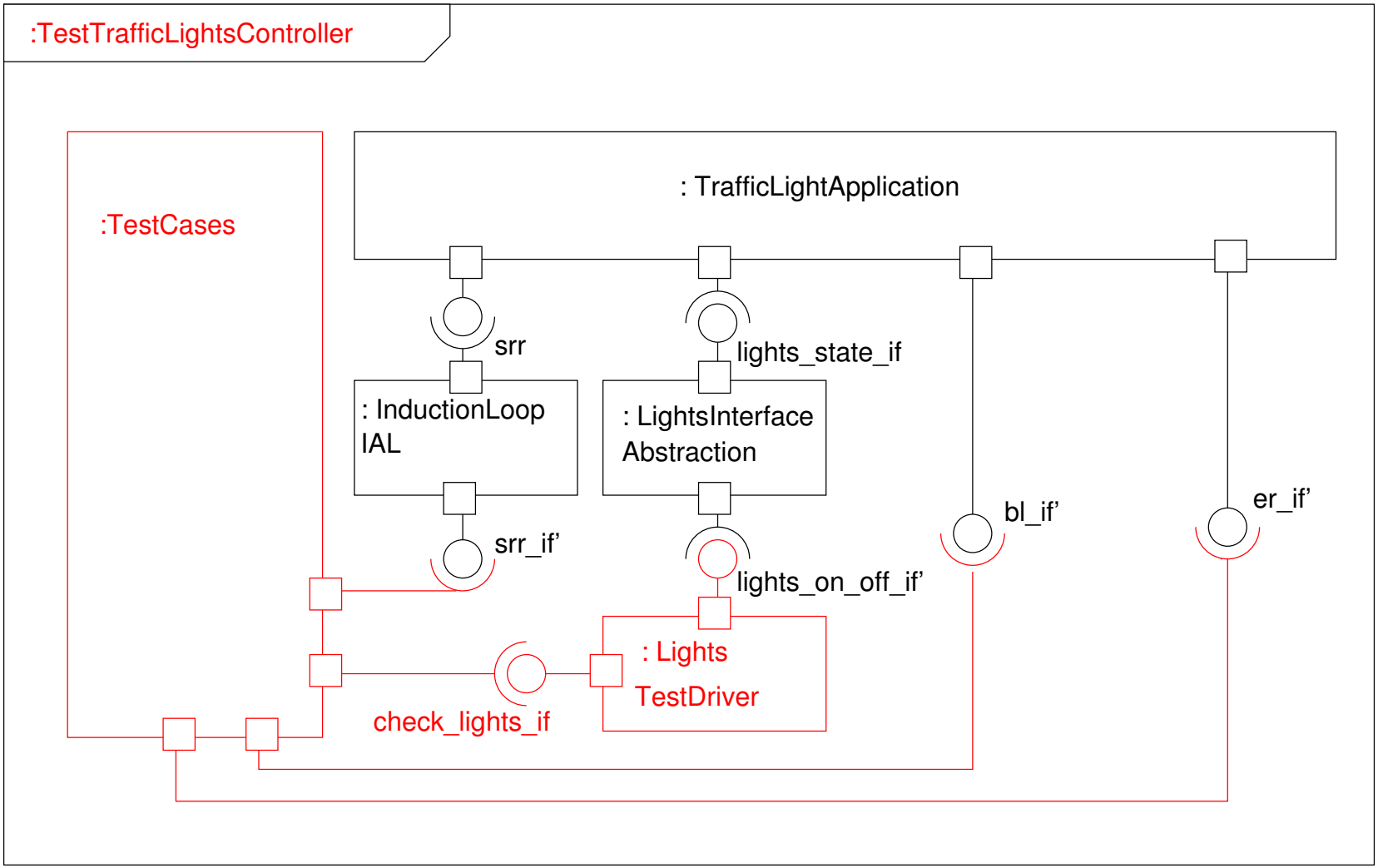

```
package tlc;
public class MainInit {
    // declare all components
    private BrokenLightDriver bld;
    private EmergencyRequestDriver erd;
    private InductionLoopDriver ild;
    private LightsInterfaceAbstraction lia;
    private LightsDriver ld;
    private TrafficLightsApplication tla;

    public MainInit() {
        // create all components, connect all components
        ld = new LightsDriver();                // Actuators
        lia = new LightsInterfaceAbstraction (ld);
        tla = new TrafficLightApplication();    // Application
        bld = new BrokenLightDriver(tlb);       // Sensors
        erd = new EmergencyRequestDriver(tlb);
        ild = new InductionLoopDriver(tlb);
        tla.ConnectTo(lia);
              // Connect components and start Application
    }
```

Create main class containing all components III


```
    public static void main(String[] args) {
        MainInit m = new MainInit();
    }
}
```

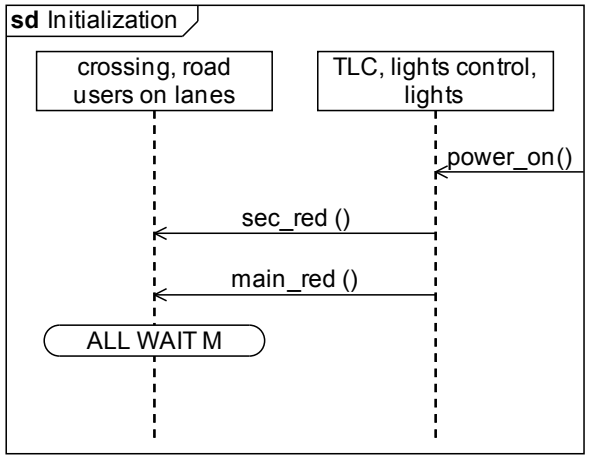
Parameters are used to create connections according to the software architecture. The order of initialisation is important. Start with objects without required interfaces.

Test environment for integrated software



Test cases according to Phase 6.

Implementation of test cases for TrafficLightControl I

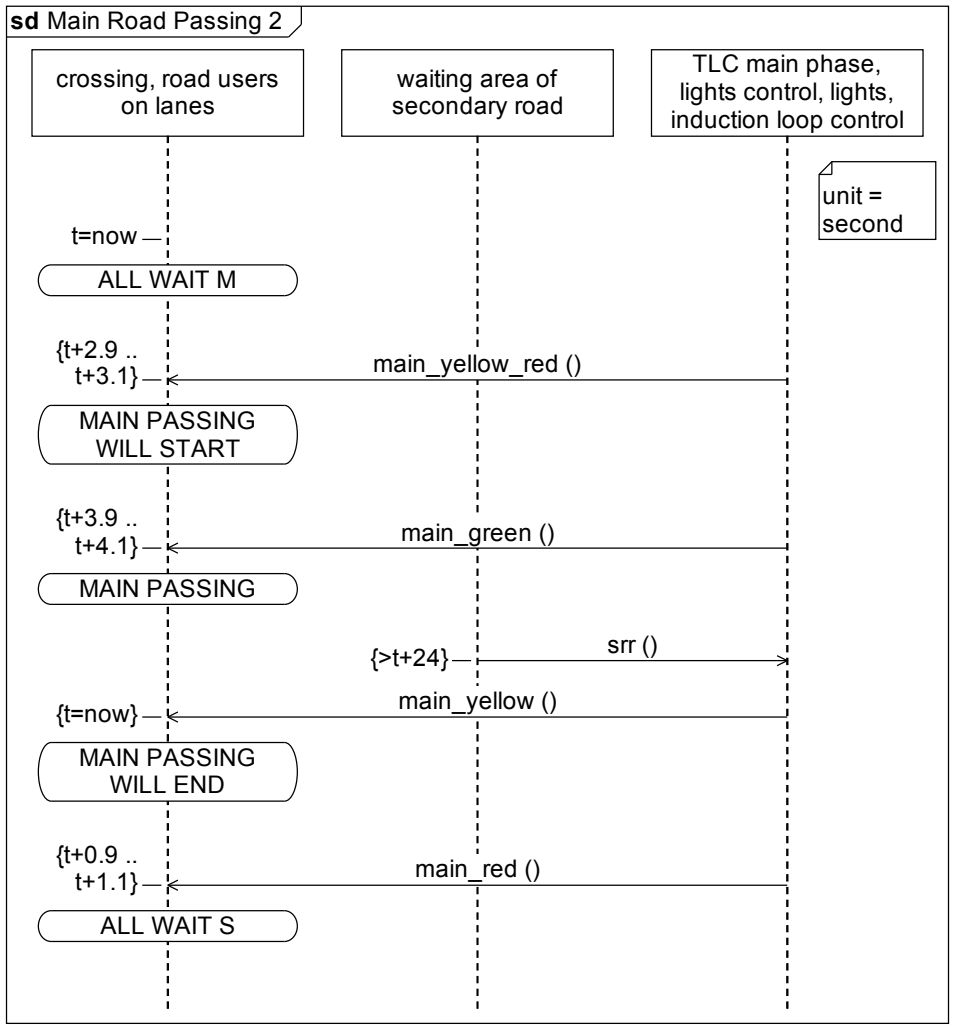



```
package tlc;
import junit.framework.TestCase;
public class TrafficLightBehaviorTest  extends TestCase {
    // declare components
    TrafficLightApplication tla;
    LightsInterfaceAbstraction lia;
    LightsTestDriver ltd;
    InductionLoopIAL il;

    public void testInitialization() {
```

Implementation of test cases for TrafficLightControl II


```
      // Initialize the test environment and the SUT
            (System unter test).
        tla = new TrafficLightApplication ();
        ltd = new LightsTestDriver();
        lia = new LightsInterfaceAbstraction (ltd);
        il. = new InductionLoopIAL(tla);
        tla.ConnectTo(lia);

      // checks lights using the testdriver
        assertTrue("main_red and sec_red not set",
           ltd.checkColor(true, true, false, false, false, false));
    }
    ...
}
```

Implementation of test cases for TrafficLightControl III



Implementation of test cases for TrafficLightControl IV


```
package tlc;
import junit.framework.TestCase;
public class TrafficLightBehaviorTest extends TestCase {
    ...
    public void testTLC_Main_Road_Passing_2() {
        // ALL_WAIT_M
        // wait 3 seconds
         Thread.sleep(3000);
        // checks lights state using the testdriver
         assertTrue("main_red_yellow not set",
            lia.checkColor(true, true, true, false, false, false));
        // wait 1 second
         Thread.sleep(1000);
        // checks lights state (sec_green) using the testdriver
         assertTrue("main_green not set",
            lia.checkColor(false, true, false, false, true, false));
        // MAIN_PASSING
        // wait 21 seconds
         Thread.sleep(21000);
        // sends directly the srr signal to the provided interfaces
         il.srr()
```

Implementation of test cases for TrafficLightControl V


```
        // checks result using the testdriver
         assertTrue("main_yellow not set",
            lia.checkColor(false, true, true, false, false, false));
        // MAIN_PASSING_WILL_END
        // wait 1 second
         Thread.sleep(1000);
        // checks lights state using the testdriver
         assertTrue("main_red not set",
            lia.checkColor(true, true, false, false, false, false));
        // ALL_WAIT_S
    }
    ...
}
```

Validation: run tests

Output of JUnit test environment:

- Test result with one error:

```
Testsuite: tlc.TrafficLightBehaviorTest
Tests run: 24, Failures: 1, Errors: 0, Time elapsed: 3,345 sec
Testcase: testInit2(tlc.TrafficLightBehaviorTest): FAILED
main_red not set
junit.framework.AssertionFailedError: main_red and sec_red not set
    at tlc.TrafficLightBehaviorTest.testInitialization(TrafficLightBehaviorTest.java:103)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:
Test tlc.TrafficLightBehaviorTest FAILED
```

- Test result with no errors:

```
Testsuite: tlc.TrafficLightBehaviorTest
Tests run: 24, Failures: 0, Errors: 0, Time elapsed: 103,345 sec
```

Example 2: sun blind control

Implementation of SunBlindContoller

See Netbeans-Project on http://swe.uni-due.de.

Phase 12: Integrate and test hardware and software

...

7. Design a software architecture for all components of the global system architecture that should be implemented in software
8. Specify the behavior of all components of all software architectures, using sequence diagrams
9. Specify the software components of all software architectures as state machines
10. Implement software components and test environment
11. Integrate and test software components
12. **Integrate and test hardware and software**

Phase 12: Integrate and test hardware and software

input:	system architecture from Phase 5	composite structure diagram
	system specifications from Phase 4	sequence diagrams with annotated states
	expression of the subproblem relationships from Phase 3	grammar
	implemented software from Phase 11	programming language
output:	integrated system	machine
	acceptance test cases	test system and/or test plans
validation:	run tests	test results

Executing Phase 12

- Load software into target (microcontroller).
- Perform manual tests.
- Build test environment for automated test.
- Implement test cases for the whole machine according to the sequence diagrams from phase 4.
- Run test cases.

Remarks

- The acceptance test should not be done by the developer.
- The test environment can be developed in parallel to the design and implementation phases.
- The test environment has to interact with the external interfaces of the machine. Hence the technical interfaces of the test system also consist of hardware.

What do we gain by defining such a process? I

- Sequence of well-defined steps helps developers to focus attention on relevant parts of the task (and fake a rational design process ;-).
- Developed models and their interrelations can be checked in each step.
- Validation is integral part of the process:
  - Validation conditions are defined for each step.
  - Systematic test case generation is part of the process.
- Certification according to safety- and security standards (IEC 61508 and Common Criteria) is supported.

What do we gain by defining such a process? II

- Various possibilities for tools support:
  - UML tools available.
  - Tool for generating sequence diagrams available.
  - Model checker for UML state machines available.
  - Other tools conceivable.
- Component-based development is supported.
- Hardware as well as software components can be part of the developed system (machine).
- Specific attention is paid to the analysis phase and the modeling of the environment. (Environment models yield test cases.)
- Non-functional (quality) characteristics can be taken into account (in particular, safety and security; by specific architectures and problem frames).

What do we gain by defining such a process? III

- Systematic evolution of existing systems is supported (traceability links between different models / documents).
- Problem decomposition is performed explicitly and systematically. Relations between subproblems are exploited to compose partial solutions of subproblems.
- Using patterns in various phases support re-use of existing knowledge and (partial) automation:
  - Problem Frames for analysis
  - Architectural patterns for software design
  - Code patterns for implementing state machines
- Process emerged from industrial practice, uses well-established languages and techniques. Hence, no ivory-tower invention.

What do we gain by defining such a process? IV

Therefore, we can hope that with DePES, we are able to develop better products with less effort.

However: DePES is not a light-weight process!