UNIVERSITÄT DUISBURG ESSEN

Faculty of Engineering
Department of Computer Science
Software Engineering

Master Thesis

A Pattern- and Component-Based Process for Embedded Systems Development

Dipl.-Inform. Denis Hatebur,
email: denis.hatebur@uni-duisburg-essen.de

Contents

List of Figures

List of Tables

1 Introduction

This thesis describes a pattern- and component-based process for embedded systems development.

Embedded systems are computer-based systems being part of a product other than a computer [Sim04]. They consist of hardware- and software-components. Embedded systems can be found in almost every area of daily life. They are used in the application domains automotive, aviation and space technology, medical technology, traffic guidance technology, industrial automation, telecommunications, business, entertainment, and household. According to Broy and Pree [BP03], about 98 % of the CPUs produced worldwide are used in embedded systems. Since embedded systems are usually produced in large numbers, incorrectly functioning systems might cause large damages.

Hence, it is crucial to develop embedded systems in such a way that the probability of errors is minimized. This aim can be achieved by following a development process with clearly defined and manageable steps. Such a process is defined in [HH05b].

Instead of starting the development from scratch each time, making use of a body of accumulated knowledge improves the quality and reduces the time to market. The use of patterns is a promising way of making use of accumulated knowledge. Patterns can be used in different phases of the software lifecycle. Problem frames are patterns for representing simple software development problems, and architectural patterns are patterns for representing the coarse-grained structure of a piece of software. Another approach to use accumulated knowledge is the reuse of components, developed in other projects.

The use of patterns and components must be integrated into the development process. In this thesis the development process defined in [HH05b] will be refined and adapted in such a way that the problem can be decomposed, the decomposed subproblems can be used to develop an appropriate system and software architecture using patterns. The software architectures for all subproblems should be combined in a systematic way.

In Chapter 2, basic concepts used in this thesis, e.g., problem frames, sequence diagrams, architectural styles, and composite structure diagrams are described. To integrate patterns and components into the development process, the process used as a basis will be described in Chapter 3. Chapter 4 identifies points for improvements within the development described in Chapter 3. In Chapter 5, the refined and adapted development process is presented. This process is then applied to two case studies: A Traffic Light Control is developed in Chapter 6 and an Automatic Teller Machine is developed in Chapter 7 using the improved process. Chapter 8 concludes with a discussion of the process and directions for future research.

2 Basic Concepts

In this chapter, the basics of this thesis are briefly described.

2.1 Agenda Concept

The *agenda* concept [Hei98] can be used to describe processes. An agenda is a list of steps or phases to be performed when carrying out some tasks in the context of systems and software engineering. Each step results in a document that is expressed in a certain language. E.g., natural language, problem diagrams (cf. 2.4), UML (Unified Modeling Language [OMG05]) diagrams, or even formal languages can be used. Agendas contain informal descriptions of the steps, which may depend on each other. Therefore they are a method to guide systems and software development activities. Additionally, agendas support quality assurance, because the steps may have validation conditions associated with them that help to detect errors as early as possible in the process. These validation conditions state necessary semantic conditions that the developed artifact must fulfill in order to serve its purpose properly.

2.2 Context Diagrams

The environment in which the machine will operate is represented by a context diagram [Jac01]. It is used for structuring of problems by structuring the description of the environment.

A context diagram consists of domains and interfaces. Domains are represented by rectangles. Plain rectangles denote *application domains* (that already exist), a rectangle with a single vertical stripe denotes a *designed domain* physically representing some information, and a rectangle with a double vertical stripe denotes the machine to be developed. The connecting lines represent interfaces that consist of *shared phenomena*. A shared phenomenon of an interface is controlled by one domain and it can be observed by the other domain. However, a context diagram does not show who is in control of the shared phenomena. An example of a context diagram is shown in Fig. 2.1. This context diagram shows a patient monitoring system. The Monitor machine is the machine domain in this context. The domain Periods & Ranges is a designed domain. All other domains (e.g., Medical staff, Nurses' station) are application domains. The interfaces in this context diagram are donated with a, b, c, d, e, and f. Period, Range, PatientName, and Factor are shared phenomena associated with the interface a.

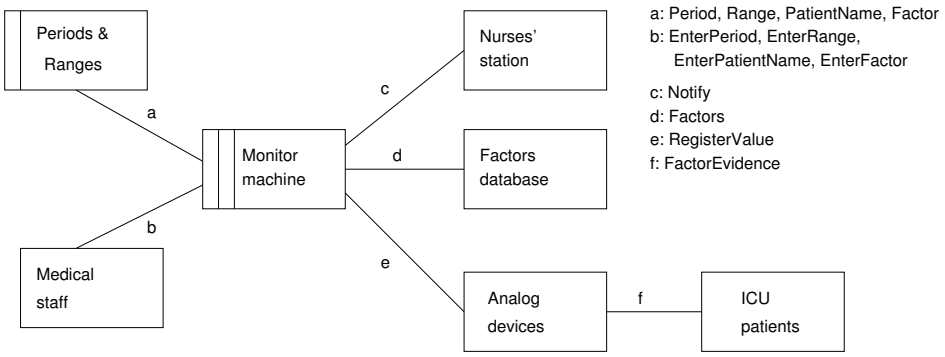


Figure 2.1: Context Diagram: Patient Monitoring System (cf. [Jac01])

The domain Analog devices is a *connection domain*. Connection domains connect two ore more other domains. They represent a communication medium or device between these domains. Connection domains have to be considered if connections are unreliable, introduce delays that are an essential part of the problem, convert phenomena, or are mentioned in the requirements. Other examples for connection domains are a network connection, a display unit, or a keyboard that is used for user input.

2.3 Problem Decomposition

The decomposition of realistic problems into simpler subproblems is necessary for solving the problems. It is also necessary for capturing, describing, and understanding realistic problems.

A *Top-Down-Decomposition*, a *Use-Case-Decomposition* or a *"Knowledge-based" decomposition* can be performed to decompose a realistic problem [CHH05b].

Top-Down-Decomposition The Top-Down-Decomposition is the oldest and worst approach. Functions are arranged in a hierarchy of several levels. At each level function are decomposed into a number of functions at the next level. This process is stopped when a level is reached where all functions are regarded as elementary. This approach takes no explicit account of the problem to be decomposed and it is unlikely to achieve a good decomposition without being familiar with the problem.

Use-Case-Decomposition The Use-Case-Decomposition is well-known through object-oriented analysis. It works well when it makes sense to think of the machine as a facility offering discrete services that are used in clearly defined episodes. But it is not suitable for continuing interaction between machine and problem domains, as often needed for embedded systems.

"Knowledge-based" decomposition The "Knowledge-based" decomposition through projection decomposes a problem into "parallel" (not hierarchical) subproblems. The

knowledge of problem classes and their solutions are used for the decomposition. The subproblems are complete, independent problems with their own problem diagrams. When a subproblem is analyzed, the other subproblems are considered as solved. This decomposition leads to a separation of concerns.

2.4 Problem Frames

Problem frames are a means to describe software development problems using "knowledge-based" problem decomposition. They were invented by Michael A. Jackson [Jac01], who describes them as follows:

> "A problem frame is a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirements."

Problem frames are described by *frame diagrams*, which basically consist of rectangles and links between them. The domains in the problem frames are denoted in the same way as in the context diagrams. The links represent interfaces that consist of *shared phenomena*. A phenomenon is controlled by one domain and can be observed by another domain. The notation "U!E3" means that the user commands E3 are controlled by the User (cf. Fig. 2.2).

Requirements are denoted with a dashed oval. A dashed line represents a requirements reference. It connects all domains that are necessary to express the requirement with the dashed oval. An arrow shows that it is a *constraining* reference.

Jackson distinguishes *causal* domains, *lexical* domains, and *biddable* domains. Causal domains, which are indicated by "C", comply with some laws. Lexical domains are data representations. The "X" on the Workpieces domain (cf. Fig. 2.2) indicates that this domain is a lexical domain. Biddable domains indicated by "B" are persons.

Jackson defines five basic problem frames (*Workpieces, Transformation, Required Behavior, Commanded Behavior*, and *Information Display*).

The following problems fit to the *Workpieces* problem frame:

> '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, analyzed or used in other ways. The problem is to build a machine that can act as this tool.' [Jac01]

The lexical domain Workpieces is an inert domain. The objects in this domain can be copied, printed, analyzed, and modified by the domain customer. The corresponding frame diagram is shown in Fig. 2.2.

The following problems fit to the *Transformation* problem frame:

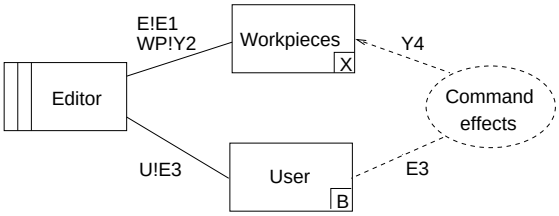


Figure 2.2: Workpieces Frame Diagram (cf. [Jac01])

> 'There are some 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.' [Jac01]

The corresponding frame diagram is shown in Fig. 2.3.

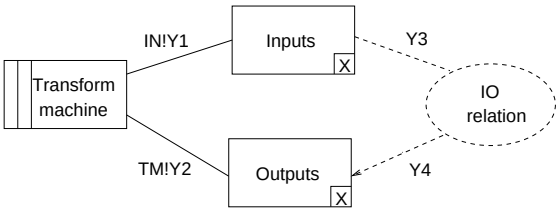


Figure 2.3: Transformation Frame Diagram (cf. [Jac01])

The following problems fit to the *Required Behavior* problem frame:

> '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.' [Jac01]

The corresponding frame diagram is shown in Fig. 2.4.

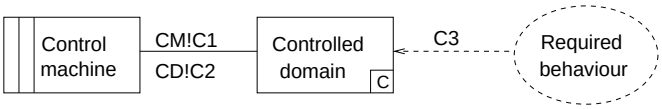


Figure 2.4: Required Behaviour Frame Diagram (cf. [Jac01])

The following problems fit to the *Commanded Behavior* problem frame:

> ‘There is some part of the physical world whose behaviour is to be controlled in accordance 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.’ [Jac01]

The corresponding frame diagram is shown in Fig. 2.5. The phenomena E4 in this figure are the operator commands.

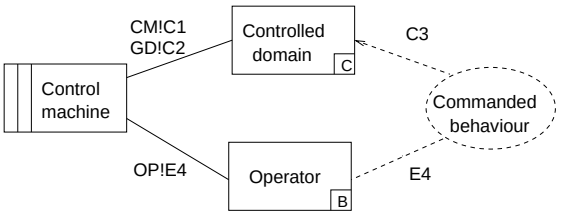


Figure 2.5: Commanded Behaviour Frame Diagram (cf. [Jac01])

The following problems fit to the *Information Display* problem frame:

> ‘There is some part of the physical world about whose states and behaviour 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.’ [Jac01]

The *Information Display* problem frame offers a structure for applications devoted to the display of real world physical data. The corresponding frame diagram is shown in Fig. 2.6. The interface between the Information machine and the Real world contains only phenomena C1 that are controlled by the real world. This means that the machine cannot influence the real world. Its purpose is only to display things that happen in the real world.

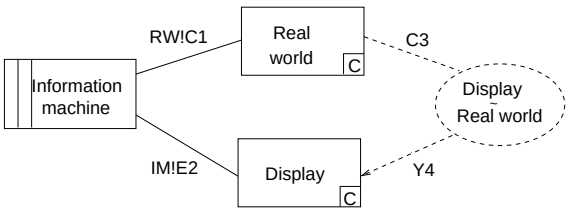


Figure 2.6: Information Display Frame Diagram (cf. [Jac01])

The *Commanded Information* problem frame (Figure 2.7) is derived from the *Simple IS* frame [Jac95]. In [Jac01], the *Commanded Information* frame is presented as a variant of the *Information Display* frame, where an operator is added. The *Commanded Information* frame is very similar to a *Database Query* frame [CH04], the only difference being that the domain to be displayed does not need to be causal, but can also be a lexical domain (cf. [Jac01]).

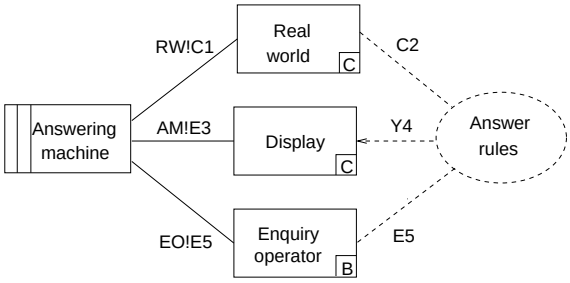


Figure 2.7: Commanded Information Frame Diagram (cf. [Jac01])

To fit a subproblem to a problem frame, one must instantiate its frame diagram, i.e., provide instances for its domains, phenomena, interfaces, and requirements. The instantiated frame diagram is called a *problem diagram*.

Additionally, the properties and the behavior of all domains should be described. This *domain knowledge* can be explained essentially in the following way [Jac01]:

> "These descriptions are *indicative* – they indicate the objective truth about the domains, what's true regardless of the machine's behaviour."

Requirements describe the environment, the way it should be after the machine is integrated. *Assumptions* are conditions that are needed, so that the requirements are accomplishable. Usually, they describe required user behavior or assumed properties of a domain. The assumptions are the basis for the user manual.

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. They are implementable requirements. The following properties can be used to identify non-implementable requirements:

1. *Controlled by the environment, not observable by the machine*: These shared phenomena are connecting problem domains, but are not directly connected to the machine domain. Shared phenomena of this category often belong to the assumptions or domain knowledge.

2. *Controlled by the environment, observable by the machine*: These shared phenomena have a relation or have been derived out of the requirements. Shared phenomena of this class are represented by a link between machine and problem domain in a context diagram.

3. *Controlled by the machine, observable by the environment*: These shared phenomena are also represented by a link between machine and problem domain in a context diagram, but with another control direction.

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

For the correctness of a specification S in relation to the domain knowledge D, the requirements R, and the assumptions A, the following implication must be shown:

$A \wedge D \wedge S \Rightarrow R$, where $A \wedge D \wedge S$ must be non-contradictory.

If the requirements are transformed into a formal notation, the implication can be proved formally. Otherwise the implication can be used as to structure an informal explanation.

2.5 Sequence Diagrams

UML *sequence diagrams* [OMG05] can be used to express interactions between *actors*, *objects*, and *processes*. In a sequence diagram, the time progresses from the top to the bottom. The interaction is expressed by *messages* between *lifelines*. Messages are represented by arrows and annotated with names, while lifelines are represented by vertical dashed lines. A sequence of messages represents a certain *scenario*, which is of concern for a security problem. Sequence diagrams are used to express sequences representing *normal case scenarios* and additionally *exceptional case scenarios*. All sequence diagrams together express the *behavior of the machine*.

Sequence diagrams can also be used to express interactions between different domains of problem diagrams. The shared phenomena are represented by messages, while the involved domains are represented by processes or objects in the sequence diagram.

A sample sequence diagram for env and machine is shown in Fig. 2.8.

This sequence diagram starts with the state invariant init_state. It ends with the state invariant end_state. For this state another valid notation is used. The state invariant is a constraint, which is assumed to be evaluated during runtime. The constraint is evaluated immediately prior to the execution of the next actions. If the constraint is true, the trace is a valid trace. If the constraint is false, the trace is an invalid trace.

The messages FoundMsg and LostMsg are lost or found messages. They are used to express messages that are lost or found during a transmission. A found message is therefore sent by a component that is not in the specification.

In Fig. 2.8 timing constraints are annotated. The message Msg3 should be sent within the given time LIMIT that is starting with the message Msg2.

In the sequence diagram, the *combined fragments* ALT to express alternatives and LOOP to express repetitions. Other operators for combined fragments in sequence diagrams are:

alt alternatives; more than two alternatives are possible.

opt option

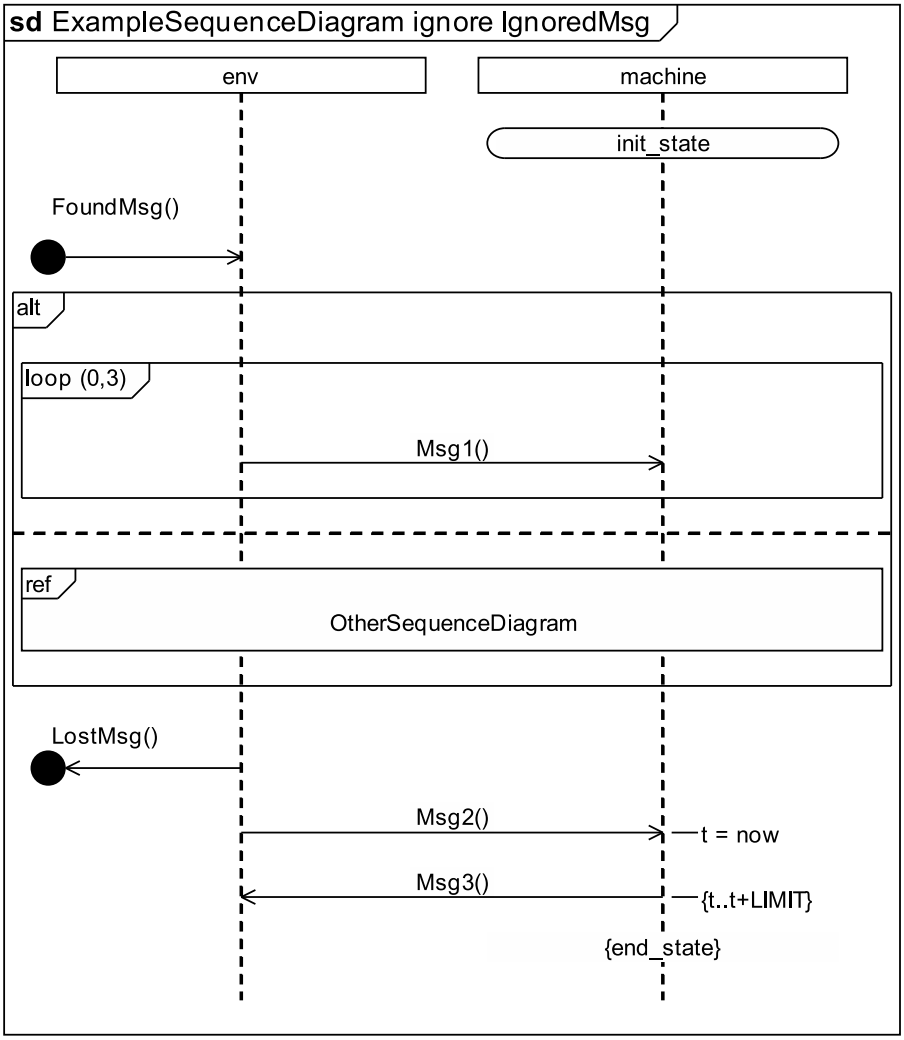


Figure 2.8: Sequence diagram example

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

2.6 Composite Structure Diagrams

Composite structure diagrams [OMG05] are a means to describe architectures. They contain named rectangles, called *parts*. These parts are components of the software or of the hardware. Each component may contain other (sub-)components. Atomic components can be described by state machines and operations for accessing internal data.

Parts have *ports*, denoted by small rectangles. Ports may have interfaces associated to them. Provided interfaces are denoted using the "lollipop" notation, and required interfaces using the "socket" notation. Figure 2.9 shows how interfaces in problem diagrams are transformed into interfaces in composite structure diagrams.

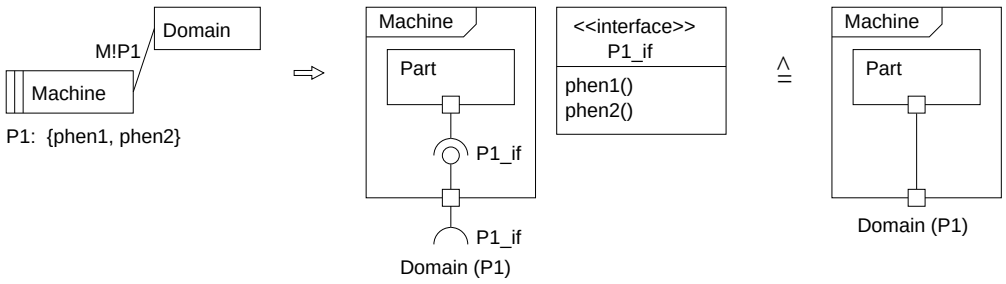


Figure 2.9: Notation for Architectures

The partial problem diagram shown on the left-hand side of Figure 2.9 states that the phenomena phen1 and phen2 shared between the machine and a domain are controlled by the machine. In the composite structure diagram (with associated interface class) shown in the middle of

Figure 2.9, this is expressed by a required interface P1_if of the component named Part, which is the same as for the whole machine.

Shared phenomena controlled by a domain correspond to provided (instead of required) interfaces of the component and the machine, respectively. Because of this direct correspondence, the socket and lollipop notation can be replaced by connectors between ports as shown on the right-hand side of Figure 2.9.

2.7 Architectures and Architectural Styles

According to Bass, Clements, and Kazman [BCK98],

> "the software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them."

Architectures can be given for hardware and software. To express hardware architectures, usually components with interfaces between the components are used. The component-based structuring is provided by hardware description languages like VHDL or Verilog and also by UML composite structure diagrams. In contrast, the architecture of software is multi-faceted: there exists a structural view, a process-oriented view, a function-oriented view, an object-oriented view with classes and relations, and a data flow view on a given software architecture. In this thesis a structural view with interfaces is used for the following reasons:

- The process-oriented view can be added by defining components as active or passive.
- The function-oriented view can be extracted from the interfaces and its description.
- Composite structure diagrams can be mapped onto a class model.
- Within a part object-oriented classes and relations can be used to express complex data type.
- The data flow is given by the parameters and return values of methods in the interface classes.

Architectural styles are patterns for software architectures. A style is characterized by Bass, Clements, and Kazman [BCK98] as

(i) a set of component types (e.g., data repository, process, procedure) that perform some function at runtime,

(ii) a topological layout of these components indicating their runtime interrelationships,

(iii) a set of semantic constraints (for example, a data repository is not allowed to change the values stored in it), and

(iv) a set of connectors (e.g., subroutine call, remote procedure call, data streams, sockets) that mediate communication, coordination, or cooperation among components.

When choosing an architecture for a system, usually several architectural styles are possible, which means that all of them could be used to implement the functional requirements. Which architectural style is the most appropriate must then be decided using *non-functional* criteria such as efficiency, scalability, or modifiability.

Some important architectural styles are the *layered architecture*, the *repository architecture*, and the *pipe and filter architecture*.

Layered architecture The layered architecture allows a hierarchical organization of software. "Lower" layers provide services for "higher" layers. A well-known example is the ISO/OSI reference model for communication protocols. The layered architectural pattern, shown in Fig. 2.10, consists of an application layer that processes the signals corresponding to those in the physical environment. The Interface Abstraction Layer (IAL) transforms the signals of the application into signals that can be understood by the Hardware Abstraction Layer (HAL). This layer provides abstract interfaces to the hardware components. The components in the layered architecture are either *Communicating Processes* (active components) or used with a *Call-and-Return* mechanism (passive components). That design decision is taken in a later step of the development.

Repository architecture The repository architecture consists of a central data storage and several client components that use the central data storage to exchange data. The repository architectural style is shown in Fig. 2.11.

Pipe and filter architecture The pipe and filter architecture consists of several sequential components. Each component performs one step of the complete task (filter) and the results are transferred to the next component using a pipe.

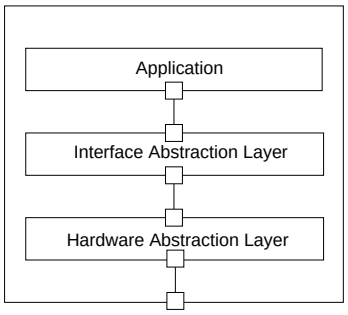


Figure 2.10: Layered Architecture

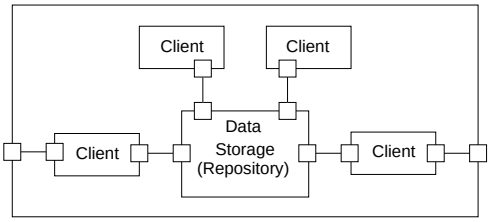


Figure 2.11: Repository Architecture

2.8 State Machines

State machines [OMG05] are a means to describe the behavior of a component. They contain *states* and *transitions*. Each state machine must contain an initial state. The initial state is indicated with a big dot and an arc pointing to the initial state. Each transition is triggered by an *input signal*. The input signal is separated from the *actions* and *output signals* by a slash. The *actions* and *output signals* are separated by commas. A transition can be guarded.

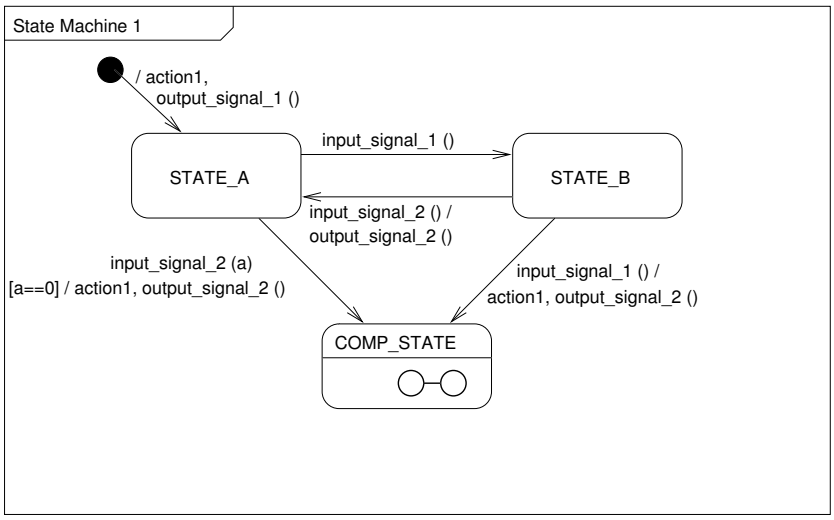


Figure 2.12: State Machine

The state machine in Fig. 2.12 consists of three states: STATE_A, STATE_B and the composite state COMP_STATE indicated by a special symbol. A composite state contains a further state machine. The initial state is STATE_A. Input signals are *input_signal_1* and *input_signal_2*. The action in this diagram in called *action1* and the output signals are called *output_signal_1* and *output_signal_2*. The signal *input_signal_2* has the parameter a. This parameter is used to guard transitions depending on the value of a. Hence, *action1* is executed and the signal *output_signal_2* is sent only if the expression *[a==0]* is true.

3 A Model Based Development Process for Embedded Systems

In [HH05b], a development process for embedded systems is presented. It was developed over time and gradually improved in an industrial context. It is based on development processes used for developing security-critical systems according to the Common Criteria [CC99] and the procedure required for developing safety-critical systems according to IEC 61508 [Int98]. The process emerged from projects dealing for example with smartcard operating systems and applets for smartcards in the area of security-critical systems, and motor control and automatic doors in the area of safety-critical systems.

In this chapter, the development process for embedded systems is presented, following the *agenda* concept (cf. Section 2.1). To apply the agenda, the following steps have to be performed:

1. Describe Problem
2. Consolidate Requirements
3. Decompose Problem
4. Derive Specifications
5. Express System Behavior
6. Design System Architecture
7. Derive Interface Behavior
8. Design Software Architecture
9. Specify Software Components
10. Implement Software Components and Test Environment
11. Integrate Hardware and Software

The following sections describe how to carry out and validate all the steps of the development process. Each step is motivated and explained in detail. It contains the input being necessary to perform the step, the generated output and validation activities. For each input entity and output entity a notation is recommended.

3.1 Step 1: Describe Problem

Step 1 of the agenda is a creative process. Table 3.1 shows the input, the output, and the validation activities for this step.

input:	mission statement (SM)	natural language
output:	requirements R	natural language
	glossary with definitions and designations	natural language
	domain knowledge D	natural language
	assumptions A	natural language
	context diagram	Jackson
validation:	in Step 2	

Table 3.1: Step 1 - Describe Problem

In contrast to other work, it is distinguished between requirements and a system mission statement (SM). This helps to classify the requirements in "need to have" and "nice to have". The system mission statement describes the purpose of the system in general terms. The requirements (R), in contrast, describe in more detail how the environment will behave after the developed system is integrated in it. The requirements are supposed to be a refinement of the system mission. Domain knowledge (D) consists of facts that are true no matter how the embedded system is built. Assumptions (A) usually are rules how users should behave, but which cannot be enforced[1]. Assumptions can also be associated with causal domains (e.g., one could assume that the display will not fail). A context diagram distinguishes between environment and machine. It structures the environment into a machine and (usually several) problem domains. The notation proposed by Jackson can be used. The requirements refer to the problem domains. The domain knowledge and the assumptions describe the domains. A glossary is important to clarify the domain-specific vocabulary which is used in the requirements, in the assumptions, and in the domain knowledge. Requirements, context diagram, glossary, domain knowledge, and assumptions are developed in parallel.

The informal way of description used here is helpful to communicate with customers.

3.2 Step 2: Consolidate Requirements

In this step, the consistency between the system mission and the requirements is checked. Table 3.2 shows the input, the output, and the validation activities for this step.

To consolidate the requirements, it must be guaranteed that the domain knowledge, the assumptions, and the requirements are not contradictory. They should suffice to accomplish the system mission. In most cases, domain knowledge, assumptions and further requirements

[1] For more details, see [ZJ97, HS99].

input:	all results of Step 1	natural language Jackson
output:	set of consolidated requirements distinguish between "need to have" and "nice to have"	natural language
validation:	$\frac{D \wedge A \wedge R \text{ are consistent}}{D \wedge A \wedge R' \Longrightarrow SM}$ to determine completeness of requirements and to determine set of most important requirements	

Table 3.2: Step 2 - Consolidate Requirements

have to be added to perform the check successfully. If there are requirements that are not necessary to show that the system mission is accomplished, then either these requirements are not mission-critical, or the system mission is incomplete. Requirements not being mission-critical can be analyzed to decide if the added value for the customer is higher than the estimated cost to develop the feature in question.

3.3 Step 3: Decompose Problem

In this step, the problem is divided into subproblems, as described by Jackson [Jac01]. Table 3.3 shows the input, the output, and the validation activities for this step.

input:	consolidated requirements R of Step 2 domain knowledge D of Step 1 assumption A of Step 1 context diagram of Step 1	natural language natural language natural language Jackson
output:	set of problem diagrams with associated set of requirements	Jackson natural language
validation:	consistent with context diagram of Step 1	

Table 3.3: Step 3 - Decompose Problem

Each requirement must belong to the requirements of some subproblem. All mission-critical requirements derived in step 2 are divided among these simple subproblems. Each requirement must therefore belong to exactly one subproblem. Otherwise the requirement must be split into two independent requirements. After a change of a requirement it is necessary to validate it again by repeating Step 2.

The subproblems are represented as *problem diagrams* (see [Jac01]). Successfully fitting a problem to a given problem frame (cf. 2.4) means, that the concrete problem indeed exhibits the properties that are characteristic for the problem class defined by the problem frame. Since all problems fitting in a problem frame share the same characteristic properties, their solutions

will have common characteristic properties, too. Therefore, it is worthwhile to look for solution structures that match the problem structures defined by problem frames.

If a problem does not fit into one of the problem frames, a new problem diagram can be developed. This should be kept as simple as possible. It can be assigned to simple requirements, i.e., the purpose of the machine is intuitively comprehensible. The problem diagram should show simple interfaces, and it is easy to characterize the way the parts interact at each interface. The domains should be clearly defined and only one problem domain should be constrained by the requirements.

The following operations can be applied to derive the subproblems as projections of the overall problem.

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

3.4 Step 4: Derive Specifications

In this step, specifications of all the subsystems to be developed (called *machines* by Jackson) are derived. Table 3.4 shows the input, the output, and the validation activities for this step.

input:	requirements R from Step 3 domain knowledge D from Step 1 assumptions A from Step 1 problem diagram from Step 3	natural language natural language natural language Jackson
output:	specification S of machine to construct	natural language
validation:	$D \wedge A \wedge S$ are consistent $D \wedge A \wedge S \Longrightarrow R$	

Table 3.4: Step 4 - Derive Specifications

Specifications are implementable requirements. Requirements that are not implementable are transformed into specifications using domain knowledge and assumptions. For an example, see [JZ95]. The specification is a description of the machine that contains all necessary information for its construction. It must be shown that, when the machine fulfills S, then the requirements are satisfied. For that proof, domain knowledge and assumptions can be used in the validation condition $D \wedge A \wedge S \Longrightarrow R$. $D \wedge A \wedge S$ must be consistent, otherwise everything can be deduced (cf. Section 2.4).

3.5 Step 5: Express System Behavior

Step 5 uses the problem diagrams from Step 3 and the specifications from Step 4. For each subproblem, the desired behavior of the corresponding machine is specified using sequence diagrams. Table 3.5 shows the input, the output, and the validation activities for this step.

input:	specifications from Step 4 problem diagrams from Step 3	natural language Jackson
output:	sequences of interactions between machine and environment	sequence diagrams
validation:	- all requirements must be captured - in the charts exactly the phenomena of the corresponding problem diagram are used - direction of signals must be consistent with control of shared phenomena as specified in problem diagram - signals must connect domains as connected in problem diagram	

Table 3.5: Step 5 - Express System Behavior

This behavior is expressed by using the UML notation for sequence diagrams. For each domain within such a problem diagram, a lifeline is drawn in the corresponding sequence diagram. The machine to be built is also represented by a lifeline in the sequence diagrams. The phenomena are represented by annotated, asynchronous signals between lifelines. This step is equipped with various validation rules that can be used to check the consistency between the problem diagrams and the sequence diagrams.

In general, the proceeding can be described as follows:

- Draw a lifeline for the machine to be built.
- Draw a lifeline for each domain (in the problem diagram) which is directly connected to the machine.
- Add asynchronous messages between the domains and the machine according to the specification.
- Add states.
- Make appropriate case distinctions according to (starting) states.
- Split the sequence diagrams at appropriate states, if necessary.
- Specify the initialization of the system.
- Refine events, make timing constraints precise.

- Add parameters to phenomena if appropriate.

Experience from many projects has shown that sequence diagrams can easily be discussed with managers and customers that do not have technical knowledge.

The specifications developed in this step can be used as a basis for manual or even automatic tests in Step 11.

3.6 Step 6: Design System Architecture

In this step, the system architecture is designed. Table 3.6 shows the input, the output, and the validation activities for this step.

For all subproblems:

input:	sequence diagrams of all subproblems from Step 5	sequence diagrams
output:	system architecture all interfaces between the components perhaps subcomponents (recursively) technical description of hardware interfaces	UML 2.0 composite structure diagrams and interface classes
validation:	all interfaces must be captured all subproblems must be captured by one or more components	

Table 3.6: Step 6 - Design System Architecture

The architecture of the embedded system is expressed as a composite structure diagram. This diagram uses parts for the components whose ports are connected as described in [OMG05]. The connections are used to transmit the signals of the annotated interfaces between the components. The interfaces with their signals are specified using interface classes. The architecture can be specified recursively, i.e., components can have their own architecture, consisting of subcomponents. The external interfaces of the components have to cover the interfaces of all problem diagrams. The architecture must cover all specifications developed in Step 5. This architecture is the starting point for the further development (hardware as well as software development).

3.7 Step 7: Derive Interface Behavior

This step refines the sequence diagrams from Step 5 for all complex components of the system architecture. Table 3.7 shows the input, the output, and the validation activities for this step.

For this sequence diagram, the signals specified in the interfaces of the architecture are used to annotate the sequence diagrams. These sequence diagrams are a concrete basis for the test implementation for all software components.

For all subproblems and for all components:

input:	architecture from Step 6 interfaces from Step 6 subcomponents (if defined) from Step 6 sequences of interactions from Step 5	UML 2.0 composite structure diagrams interface classes UML 2.0 composite structure diagrams sequences diagrams
output:	interface behavior of all complex components (test specification)	UML 2.0 sequence diagrams
validation:	consistent with input	

Table 3.7: Step 7 - Derive Interface Behavior

In general, the proceeding can be described as follows:

- Draw a lifeline for all components of the architecture that are necessary to describe the interface behavior of the subproblem and one lifeline for the environment.
- Describe the interface behavior of all components using the signals from the system architecture (Step 6). The behavior must refine the behavior described in Step 5.
- Add missing sequence diagrams to describe the behavior for all relevant states.
- Describe the behavior of all components/parts at their interfaces.
- As for Step 5: Each diagram represents one concrete interaction sequence. Do not try to make the diagrams too general. Draw further diagrams instead.

3.8 Step 8: Design Software Architecture

In this step, the software architecture for all components containing software is designed. Table 3.8 shows the input, the output, and the validation activities for this step.

The architecture of embedded software should be a layered architecture (cf. 2.7).

To perform this step, the software can be divided into device-dependent and device-independent parts according to the (extended) *four variable model* developed by David Parnas and extended by Connie Heitmeyer [BH99].

The four variables in the model are the *monitored variables*, the *controlled variables*, the *input data*, and the *output data*.

Monitored variables The monitored variables are measured quantities (i.e., physical values, monitored by sensors).

Controlled variables The controlled variables are affected quantities (i.e., physical values, controlled by actuators).

input:	architecture from Step 6 interfaces from Step 6 phenomena in sequence diagrams from Step 5 perhaps reusable classes from other projects	UML 2.0 composite structure diagrams interface classes sequence diagrams
output:	layered software architecture interfaces between software components	UML 2.0 composite structure diagrams interface classes
validation:	phenomena of sequence diagrams are interfaces of the application layer hardware abstraction layer is included direction of all signals consistent to each other and input	

Table 3.8: Step 8 - Design Software Architecture

Input data The input data are resources from which the values of monitored variables must be determined. These are submitted via a technical interface (electrical signals corresponding to digital values).

Output data The output data are resources available to affect controlled variables. They are submitted via a technical interface by the machine (electrical signals corresponding to digital values).

The basic idea of the extended four variable model is, that the application layer software should have the same interfaces as the system, i.e., monitored and controlled variables. Thus, the application layer becomes device-independent, which is factored out in IALs and HALs. (cf. 3.1)

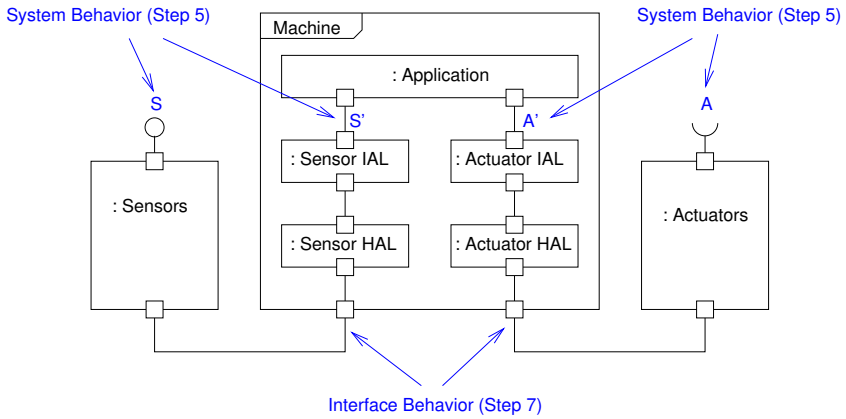


Figure 3.1: Extended Four Variable Model and Layered Architecture

The differences between the interface behavior and the system (machine) behavior are shown in Figures 3.2 and 3.3. On the left-hand side of both figures, the interface behavior of the

software is shown (corresponds to *S'*) and on the right-hand side the behavior of the machine at its external interface is shown.

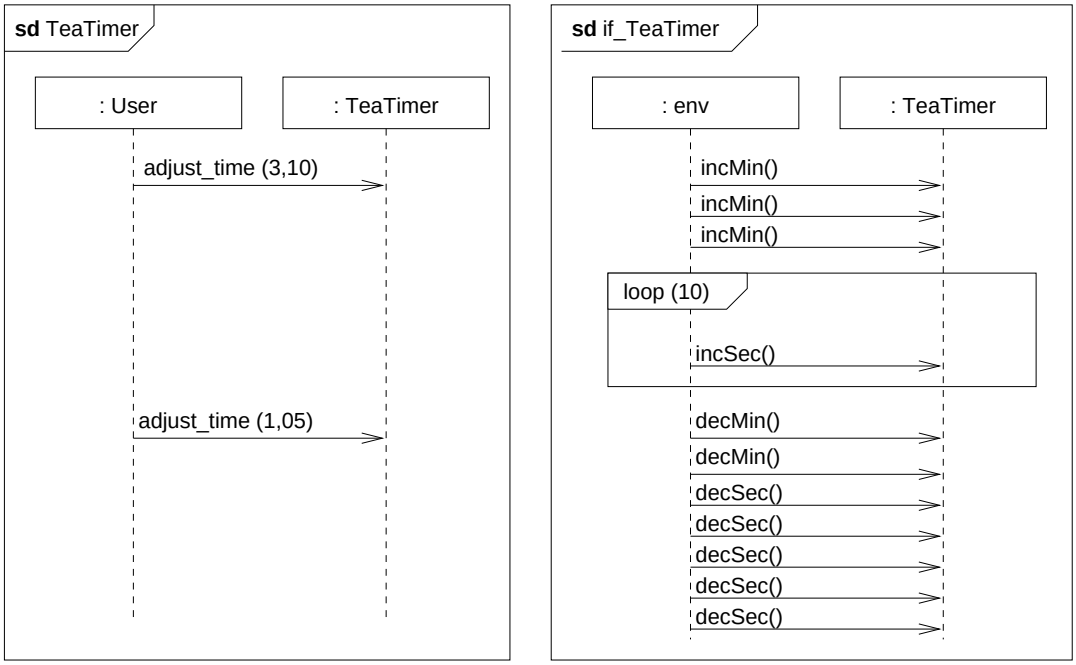


Figure 3.2: Interface vs. System Behavior 1

In Fig 3.3 an analog-digital converter (ADC) is used to transform measured weight of vessel. (5V $\widehat{=}$ value of 255)

The software architecture is expressed as a UML 2.0 composite structure diagram (cf. Section 2.6). To perform this step, already specified components of other projects can be reused.

The components in this layered architecture are either *Communicating Processes* (active components) or used with a *Call-and-Return* mechanism (passive components). That design decision is taken in a later step of the development.

In general, the proceeding for all software components can be described as follows:

- Draw the component to build with all its external ports.
- Add a driver component for each external port and connect it with the already specified interfaces. These driver components (which constitute the HAL) will abstract the access to the hardware.
- Add an application component whose interfaces contain the phenomena from Step 5 (extended four variable model).
- Add interface abstraction components that convert the signals from the driver components into signals that can be processed by the application component, if necessary.
- Refine the interface components by splitting them into subcomponents if appropriate.
- Refine the application component by splitting it into subcomponents if appropriate.

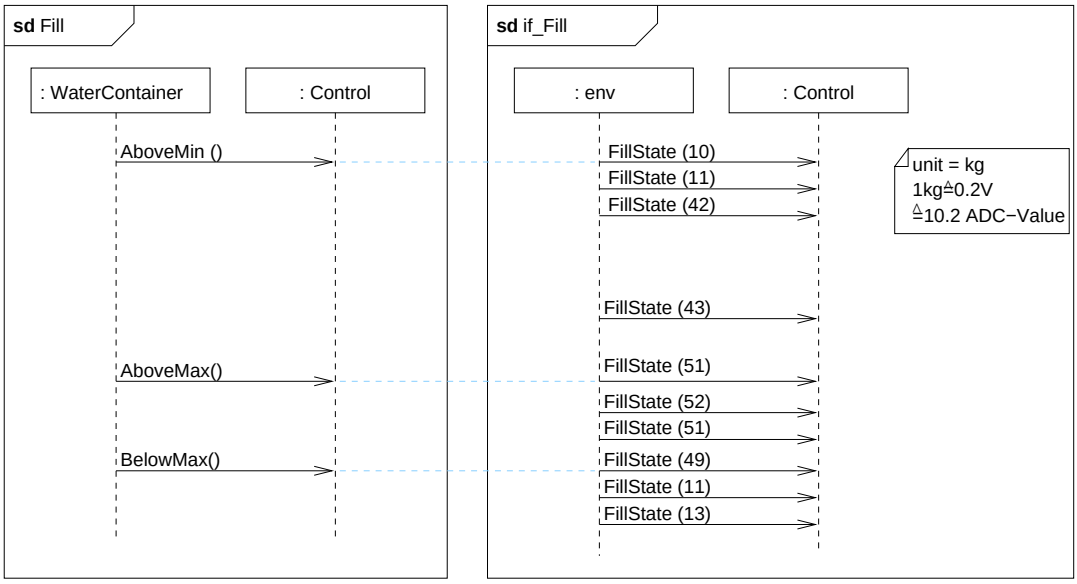


Figure 3.3: Interface vs. System Behavior 2

3.9 Step 9: Specify Software Components

In this step, the software components are specified as classes, taking a white-box view. Table 3.9 shows the input, the output, and the validation activities for this step.

input:	output of Step 8 behavior from Steps 5 and 7	
output:	component description consisting of: component overview description	UML 2.0 class diagram with ports and lollipops
	data types	UML class diagrams
	for all operations: pre- and postconditions	formulas or natural language
	invariants	formulas or natural language
	state machine	UML 2.0 state machine diagram
validation:	consistent with interface behavior completeness of state machines (implies error-cases for user-interaction)	

Table 3.9: Step 9 - Specify Software Components

The specifications in this step have to be consistent with Step 7 with respect to the behavior of data types and state machines. The state machines must be complete, i.e., there must be a specified reaction to each possible input signal. The specifications must have the same interfaces as in the component diagram designed in Step 8. In this step, we also have to decide

if the component is an active (e.g., behaves like hardware) or passive (e.g., calculation-routine) component. The result of this step forms the basis for the implementation step.

In general, the proceeding for all components in the software architecture can be described as follows:

- Draw an active or passive class with its interfaces as a component overview description.
- Add necessary data to this class.
- In case of complex data or complex operations on data types: add classes for data types.
- Specify pre- and postconditions for all operations.
- Add invariants if possible.
- Add state machines.

3.10 Step 10: Implement Software Components and Test Environment

In this step, the test environment for all software components is implemented, using the test specification from Step 7. In addition, time frames must be added, specifying when an event is expected to occur. Table 3.10 shows the input, the output, and the validation activities for this step.

input:	output of Step 7 and 5 for test environment output of Step 9 for machine output of Step 8 for machine	sequence diagrams different UML notations
output:	software (machine and test software)	
validation:	run tests (perhaps in emulator)	test results

Table 3.10: Step 10 - Implement Software Components and Test Environment

The system components are implemented using the results of Step 9, applying some simple heuristics. The components have to be connected as specified in Step 8. 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, function calls, asynchronous messages, or hardware access.

This development process allows developing statically linked software components with the capability of reuse. To validate the results of this step, tests may be run in an emulation environment.

In general, the proceeding for object-oriented programming languages can be described as follows:

1. Create interface classes for all internal interfaces (also for subcomponents).
2. Create classes for all (sub-)components and implement them.
   a) Implement actions as private methods according to the pre- and post-conditions.
   b) Implement the state machine.
   c) Implement the active classes with threads or timer libraries.
   d) Check all classes if there is concurrent access to variables and resolve this problem with synchronization statements.
3. Implement test cases for all components (except HAL) according to the sequence diagrams from Steps 5 and 7.
4. Create a method to initialize all objects according to the architecture from Step 8 and run your application.
5. Run test cases.

3.11 Step 11: Integrate Hardware and Software

In this step, hardware and software components are integrated. Table 3.11 shows the input, the output, and the validation activities for this step.

input:	hardware and output of Step 10	
output:	system and test environment	
validation:	run test with hardware and software	test results

Table 3.11: Step 11 - Integrate Hardware and Software

The test of the whole embedded system, consisting of hardware as well as software, is performed. In general, the proceeding can be described as follows:

- 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 Step 5.
- Run test cases.

The acceptance test should not be done by the developer. Therefore, the test environment can be developed in parallel to the last steps. The test environment has to interact with the external interfaces of the machine. Hence, the technical interfaces also consist of hardware.

4 Points for Improvement

In this chapter, points for improvement for the development process described in Chapter 3 are presented. The following points for improvement have been identified while applying the development process on several machines to be developed.

4.1 Existing Machine Problem

Today, machines to be developed often replace or extend machines that are in use. They are replaced or extended because the new machine brings additional benefit to the customer. This benefit must be identified to develop the requirements for the machine. In Step 1, existing machines that should be replaced are not considered. In this case the development process for embedded systems gives little help for developing the requirements systematically.

Possible Solution for the Existing Machine Problem

To solve this problem, I suggest to draw two context diagrams in Step 1. One should be used to show the actual context without the machine to be built. Another context diagram should show the environment after the machine is integrated. Requirements can be found systematically by comparing the system in use and the system at that time the machine is integrated. For example, the domain knowledge and the assumptions of removed domains give hints for requirements of the machine to be built.

4.2 System Architecture Design Problem

The development process gives little help for developing the system architecture. In the process described in Chapter 3, the components inside the machine to be built must be identified by experienced developers. An appropriate system architecture reduces costs to build the system and it also minimizes the costs to extend the system with additional components.

Possible Solution for the System Architecture Design Problem

Usually, the developer knows when more than one new machine is necessary to solve the problem by comparing the existing system with the system to be built. Systems with more

than one machine are necessary, when the machines are physically distributed (e.g., Client-Server Systems) or when two machines solving different problems are connected with a given connection domain.

This knowledge should be described in the context diagram in Step 1. But the context diagram defined by [Jac01] allows only one machine domain. I suggest to allow more than one machine domain in a context diagram. Otherwise for each machine a context diagram must be drawn, which is very redundant.

Further knowledge which is necessary for developing the system architecture can be acquired, when the problem is structured with problem frames. In the problem diagram, additional connection domains can be included. Devices to access external domains are typical connection domains that are helpful for the system architecture design. These connection domains belong to the machine or to the environment. The dot-notation, introduced by Jackson in [Jac95], can be used to express if a domain belongs to the machine or not. In the workpieces-frame, the domain Workpieces is a part of the machine, which can be indicated with a big dot as shown in Fig. 4.1.

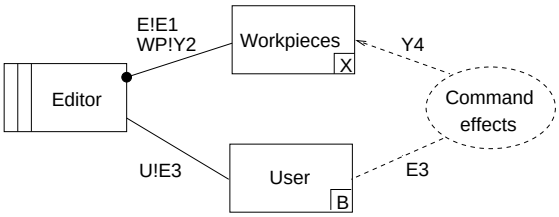


Figure 4.1: Workpieces Frame Diagram with Dot-Notation

According to Jackson, the big dot at the connecting line is only possible for lexical domains. Since this development process is used to describe machines consisting of hardware and software, also causal domains can belong to the machine (e.g., an analog-digital converter). If the connection domain belongs to the machine, another problem diagram should be used to describe the requirements for this domain. Usually a transformation is done by these domains, and the transformation frame might be instantiated.

The machine domains can be used to identify components in our system architecture. For each component in the system architecture several problem diagrams can be assigned. Hardware components usually work in parallel. Therefore, only parallel problems can be distributed to different components in the system architecture. If sequential or alternative problems should be assigned to different hardware components, there must be one component coordinating the sequential or alternative processing.

4.3 Subproblem Composition Problem

Within the process, the problem to be solved is decomposed into smaller problems and fitted to a problem frame. Once a problem is successfully fitted to a problem frame, its most important characteristics are known and it is possible to reuse software development knowledge. But the development process gives only little help for the composition of the subproblems. The composition of the subproblems is performed on the level of machine behavior description. Therefore, the knowledge about appropriate software architectures for a problem can not be reused.

Possible Solution for the Subproblem Composition Problem

To solve the problem, the dependencies (or synchronization information) between the subproblems should be described, so that this information can be used to combine the state machines as suggested in [CHH05b].

This paper only covers the software development. For the development of embedded systems an additional step is necessary: When specifying the system architecture, for each component the dependencies of the subproblems assigned to this component should be described. This description must be consistent with the dependencies of all subproblems.

4.4 Readability of Sequence Diagrams Problem

Most developers try to describe the whole behavior in only few sequence diagrams. These diagrams are very difficult to read and to explain. The reader has to extract all possible traces without getting lost in all possibilities.

Possible Solution for the Readability of Sequence Diagrams Problem

In the early steps only examples should be described. Loops, states, references and, co-regions do not cause any problems, while other new constructs of UML 2.0 such as parallelism, and continuation should be used with care. The operator *alt* and the statements *ignore* and *consider* should be used very carefully or even better left out. Each diagram should represent one concrete interaction sequence. One should not try to make the diagrams too general. It is better to draw further diagrams.

4.5 Interoperability with Other Methods Problem

In the development process, no decision point is included for the question if another method is more appropriate for the problem. The development process is tailored for Embedded System development. Usually, the behavior of Embedded Systems depends on their internal states and states machines are used to described the components. Machines dealing with complex and dynamic data can not be described adequately using extended state machines.

Possible Solution for the Interoperability with Other Methods Problem

If the architecture consists of only one software component or there are no causal domains in the environment, we should continue with the FUSION method [DCJ94], because we do not have an embedded system to be developed.

If one component of the architecture contains complex or dynamic data types or operations, this component should be developed with the FUSION method, which is more suitable for these problems.

4.6 Software Architecture Design Problem

The development process gives little help for developing the software architecture. An appropriate software architecture is important to develop reliable, maintainable, and portable software. The architecture has to be adequate for the given problem.

Possible Solution for the Software Architecture Design Problem

Architectural patterns can be provided for each problem frame as suggested in [CHH05a]. Fitting a problem to an appropriate problem frame should not only help to understand it, but also to solve the problem. The proposed software architectural patterns correspond to the different problem frames and are designed to be a starting point for the construction of the software solving the given problem. These architectural patterns exactly reflect the properties of the problems fitting to a given frame, and that they can be combined in a modular way to solve multi-frame problems. Additional alternative architectures to cope with specific system characteristics (e.g. distribution) are proposed.

4.7 Interface Control Problem

For hardware components, from the control of the phenomena can be deduced, which component provides an interface and which component requires an interface (cf. Section 2.6). For

software components this is not always true: A simple database component is in control of the requested information, but the requested information is a return value of an operation in the provided interface.

Possible Solution for the Interface Control Problem

One solution would be an abstract description with a provided interface for the query and a requested interface for the answer of the query as suggested implicitly in the development process described in Chapter 3. But this might lead to inefficient solutions. Another solution could be to allow the specification of operations with return values in the interfaces.

4.8 Notation Problem

In Step 4 (Derive Specification) and Step 5 (Describe System Behavior), the same aspects are expressed with different notations.

Possible Solution for the Notation Problem

Both steps can be merged. The sequence diagrams of Step 5 can be annotated with informal explanatory text in natural languge, which was the result of Step 4. By merging the steps the process will be more independent from the used notations.

4.9 Interface Specification Problem

In many cases, it is very difficult to describe the whole behavior at the interface of a machine. The specification of the control of a LCD-Display for example must include the control of each segment or dot of the display. It is not possible to describe the behavior of the machine at this level of detail. The same problem exists for the description of the software interface. For this reason, it is difficult to test the machine directly at the software interface.

The solution suggested in the embedded systems development process was to use an abstract notation, but there was no step where this abstract notation was put in concrete terms.

Possible Solution for the Interface Specification Problem

Instead of specifying the specification directly, the requirements and the corresponding domain knowledge can be described. For the LCD-Display a requirement may be that a certain number should be visible on the display. The necessary domain knowledge includes information about which segments or dots must be switched on and off to display this number. Then, a developer

can easily derive the specification by replacing the signal for showing the number with signals switching the segments or dots on or off.

This specification can be checked using the implication $D \wedge S \Rightarrow R$, where $D \wedge S$ must be non-contradictory.

The domain knowledge can be used to develop test interfaces and the requirements can be used to develop the test cases.

4.10 IAL and HAL Specification Problem

In Step 5, the system behavior is described, and this description is also used to specify the behavior of the application component. In Step 7, the interface of the software is described. The description in Step 5 and Step 7 only implicitly specify the IAL and HAL behavior. An explicit description of the IAL and HAL behavior is necessary for a test of these components.

Possible Solution for the IAL and HAL Specification Problem

In a further step, the IAL and the HAL can be described using sequence diagrams. The sequence diagrams describe the transformation rules from the application layer to the hardware control. The domain knowledge used to describe the specification using requirements can be reused for the specification of the IAL and HAL.

4.11 Jackson – Four Variable Model Integration Problem

In Step 4 of the development process, the specifications of the machines are derived. These specifications are the basics for the following steps of the development process. It describes the external interface of the machine and therefore also the interface of the application component of the software (cf. Chapter 3, Step 8). If the machine is only one part of the system, the machine specifications do not correspond to the monitored or controlled variables. In this case, the application component becomes more complex than necessary, and the extended four variable model will not be applied.

Possible Solutions for the Jackson – Four Variable Model Integration Problem

Two possibilities exist to solve the problem: Either the system architecture must also include the existing components that transform the monitored variables into the input data of the components to be built, or the specification for these cases is expressed as suggested in Section

4.9. I suggest not to include existing components in the system architecture which will be developed. In this case, the system architecture only consists of components which will be bought and must be selected. In this case, the specifications should be expressed using the requirements and the domain knowledge.

4.12 Contradicting Requirements Problem

In the development process, defined in [CHH05b], contradicting requirements would be found in a very late step of the development process. In case of contradicting requirements the system can not be designed as required. One requirement must be removed, changed, or priorized. Changes of the requirements must be detected and discussed with the customer as early as possible.

Possible Solution for the Contradicting Requirements Problem

Contradictions between requirements usually occur in the application component. Therefore, the IAL and HAL need not to be considered. To check if contradicting requirements exist, the sequence diagrams can be transformed into a state machine. If a state machine which is consistent to all sequence diagrams can be constructed, no contradicting requirements exist. This step can be performed, when the behavior of the machine is specified, because its behavior is the same for the application component.

4.13 Completeness of Specification Problem

For many safe systems, it is necessary to check the completeness of the specification. This can only be done at the level of state machines and not at the level of sequence diagrams. Within the process, the state machines are developed in Step 9 and an incomplete specification would be detected not until the design phase is performed. When the design of the machine should not be started before the specification is checked to be complete, this check has to be performed before.

Possible Solution for the Completeness of Specification Problem

It is not possible to check if a requirement is missing or important domain knowledge is not considered. But the methods and notations provided in the first steps of the development process help to understand the problem in detail. Completeness also includes that in all states of the machine all phenomena or signals that can occur are handled. This aspect of completeness can be checked by constructing the state machine in the same way as for the contradicting requirements problem.

5 Refined and Adapted Development Process for Embedded Systems

In this chapter, a pattern- and component-based development process for embedded systems is presented. It is emerged from the development process illustrated in Chapter 3, and it considers the points for improvement identified in Chapter 4. It includes the problem frames [Jac01] and the corresponding architectural patterns proposed in [CHH05a]. The relationships between the subproblems are expressed explicitly, and the fact these relationships are exploited when generating a global software architecture for the overall problem, as proposed in [CHH05b]. The development process also treats the design of the system architecture and the completeness and the consistency of the specification.

The process consists of the following fourteen steps explained one by one using the *agenda* concept (cf. Section 2.1):

1. Describe Problem
2. Consolidate Requirements
3. Decompose Problem
4. Derive Machine Behavior Specifications for each Subproblem
5. Design Global System Architecture
6. Derive Specifications for all Components being relevant for the Subproblem
7. Design Software Architecture for each Software Component and each Subproblem
8. Specify Behavior of Software Architecture Components for each Subproblem
9. Specify Software Components of Software Architectures for each Subproblem
10. Develop Global Software Architectures
11. Specify Composed Software Components
12. Implement Software Components and Test Environment
13. Integrate Software Components
14. Integrate Hardware and Software

In the following sections is described how to carry out and validate all the steps of the development process. Each step is motivated and explained in detail. It contains the input being necessary to perform the step, the generated output and validation activities. For each input entity and output entity a notation is recommended.

5.1 Step 1: Describe Problem

Step 1 of the agenda is a creative process. Table 5.1 shows the input, the output, and the validation activities for this step.

input:	system mission statement SM, informal description of the task	natural language
output:	context diagram of system in use context diagram of system to be built requirements R domain knowledge D glossary with definitions and designations assumptions A	Jackson Jackson natural language natural language natural language natural language
validation:	domains and phenomena in the context diagram and in A, R or D must be consistent *validation continued in Step 2*	

Table 5.1: Step 1 - Describe Problem

An informal description of the task and a system mission statement is used as an input for this step.

All domains that are relevant to the problem at hand and the phenomena that are shared by different domains must be identified. When there is already a machine in the system that should be replaced, the system (consisting of the machine and the environment) should be expressed with a context diagram. An additional context diagram should be created, which shows the system when the machine to build is integrated into the environment. In contrast to the context diagrams defined by Jackson [Jac01], more than one domain can be a machine domain. This should be only done if the machines are physically distributed or if there are two machines solving different problems being connected with a given connection domain. If there is more than one machine domain, the following steps of this development process must be applied for all machines.

The requirements R (optative statements) have to be expressed, as well as domain knowledge D and the assumptions A about the environment in which the machine (the system to be developed) has to operate (indicative statements). These can be expressed in natural language, in semi-formal, or in formal notations. In a glossary with definitions and designations the vocabulary is described.

All items of the output can be developed in parallel. When e.g., a new requirement is identified,

it is possible that an additional phenomenon must be added to the context diagram or the domain knowledge must be described more in detail. Additionally, the context diagram should be consistent with the phenomena and domains in the statements.

All domains and phenomena mentioned in A, R, and D, must be contained in the context diagram. All domains of the context diagram (except the machine domains) must be related to some elements of A, R, or D. If two context diagrams are developed, the domain knowledge and the assumptions for a domain included in both must be the same.

The step described in this section is derived from Step 1 of the process defined in [CHH05b] and Step 1 of the process described in Chapter 3. In this process two context diagrams should be created if there is a machine to be replaced. In the context diagram of this development process, more than one machine domain is allowed. The differences in "system in use" and "system to be developed" help to acquire requirements. Embedded systems often solve a problem together with other machines. To allow more than one machine domain helps to reduce the effort for describing the context, and it gives a better overview about the whole system. This covers the point of improvement described in Section 4.1.

5.2 Step 2: Consolidate Requirements

In Step 2, the consistency between the system mission and the requirements is checked. Table 5.2 shows the input, the output, and the validation activities for this step.

input:	all results of Step 1	natural language
output:	set of consolidated requirements R' distinguish between "need to have" and "nice to have"	natural language
validation:	$D \wedge A \wedge R$ are non-contradictory $D \wedge A \wedge R' \Longrightarrow SM$ to determine completeness of requirements and to determine set of most important requirements	

Table 5.2: Step 2 - Consolidate Requirements

The inputs for this step are all results of Step 1.

In this step, we distinguish between requirements being "need to have" or being "nice to have". "Need to have" requirements are necessary to fulfill the system mission. "Nice to have" can be analyzed to decide if the added value for the customer is higher than the estimated cost to develop the feature in question. The result is a set of consolidated requirements.

To validate this step it must be ensured, that the statements contained in the requirements R, the domain knowledge D, and the assumptions A are non-contradictory. They should suffice to accomplish the system mission. In most cases, domain knowledge, assumptions and further requirements have to be added in Step 1 to successfully perform the check. If there are

requirements that are not needed to show that the system mission is accomplished, then either these requirements are not mission-critical, or the system mission is incomplete.

This step is the same as Step 2 of the process described in Chapter 3. In the process defined in [CHH05b] such a step does not exists. Performing such a validation helps to reduce the development risks by prioritizing the requirements.

5.3 Step 3: Decompose Problem

In Step 3, the problem is divided into subproblems, as described by Jackson [Jac01] and in [CHH05b]. Table 5.3 shows the input, the output, and the validation activities for this step.

input:	consolidated requirements R' of Step 2 domain knowledge D of Step 1 assumption A of Step 1 context diagram of Step 1	natural language natural language natural language natural language
output:	set of problem diagrams with associated set of requirements expression of the subproblem relationships	Jackson with dot-notation natural language process algebra-like notations, grammars, high-level sequence charts, or sequence charts using combined fragments
validation:	consistent with context diagram of Step 1 all requirements of Step 2 must be captured	

Table 5.3: Step 3 - Decompose Problem

The inputs for this step are consolidated requirements R', domain knowledge D, assumption A, and the context diagram of the machine to be built.

A set of problem diagrams is developed in this step, and all mission-critical requirements R' are assigned among these simple subproblems.

There are different possibilities to decompose a complex problem into subproblems. Jackson [Jac01] proposes a parallel decomposition using projection, but a decomposition by use-cases (for an example, see [CH04]) or a top-down decomposition are also possible. The following relationships between subproblems can be identified and help to compose the solution:

- *Parallel* subproblems are largely independent of each other, and the composed machine will have to treat the problems in parallel.
- *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.

To express subproblem relationships, different means of expression are appropriate, for example process algebra-like notations, grammars, high-level sequence charts, or sequence charts using combined fragments (the latter two introduced in UML 2.0).

However, composing the solution of the overall problem from the solutions of the subproblems does *not* mean to develop an independent program for each subproblem and then compose these programs. Instead, the solutions to the subproblems will contain common components that have to be identified and then merged accordingly (cf. Steps 10 and 11). This is the challenge of the composition problem.

The subproblems are represented as *problem diagrams* (see [Jac01]). Successfully fitting a problem to a given problem frame (cf. 2.4) means that the concrete problem indeed exhibits the properties that are characteristic for the problem class defined by the problem frame. Since all problems fitting in a problem frame share the same characteristic properties, their solutions will have common characteristic properties, too. Therefore, it is worthwhile to look for solution structures that match the problem structures defined by problem frames. To fit a subproblem into a problem frame, the following operations can be applied:

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

If a problem does not fit into one of the problem frames, a new problem diagram can be developed. This diagram should be kept as simple as possible. It can be assigned to simple requirements, i.e., the purpose of the machine is intuitively comprehensible. The problem diagram should show simple interfaces, and it is easy to characterize the way the parts interact at each interface. The domains should be clearly defined, and only one problem domain should be constrained by the requirements. Otherwise, the subproblem is not simple but needs further decomposition.

In the problem diagrams, the domains belonging to the machine should be identified by a big dot at the interfaces of the machine.

To validate this step, it must be checked, that the problem diagrams are consistent with the context diagram of Step 1, i.e. only the operations described above are applied. All requirements R' of Step 2 have to be captured. All domains and all phenomena of the context diagram must be captured.

This step is derived from Step 3 of the process defined in [CHH05b] and Step 1 of the process described in Chapter 3. In contrast to these development processes, it is expressed in the problem diagrams, which domains belong to the machine to build. This information can be

used to develop the system architecture (Step 5), as explained in Section 4.2. The description of the dependencies in [CHH05b] is integrated in the process described in Chapter 3. This information can be used to combine the software architectures in Step 10 and components in Step 11 (cf. Section 4.3).

5.4 Step 4: Derive Machine Behavior Specifications for each Subproblem

In this step, the machine behavior specifications for each subproblem are derived. Table 5.4 shows the input, the output, and the validation activities for this step.

For all subproblems:

input:	requirements R' from Step 2 domain knowledge D from Step 1 assumptions A from Step 1 problem diagram from Step 3	natural language natural language natural language Jackson
output:	specification S of machine to construct sequences of interactions expressing S between machine and environment with annotated state invariants for the domains in the environment	natural language sequence diagrams
validation:	$D \wedge A \wedge S$ are non-contradictory $D \wedge A \wedge S \Longrightarrow R'$ all requirements must be captured in the sequence diagrams exactly the phenomena of the problem diagrams are used direction of signals must be consistent with control of shared phenomena signals must connect domains as connected in problem diagram the relationships of Step 3 must be consistent with the state invariants	

Table 5.4: Step 4 - Derive Machine Behavior Specifications for each Subproblem

The inputs for this step are the problem diagrams, the requirements R', the domain knowledge D, and the assumptions A.

Specifications are derived and additionally expressed as a set of UML sequence diagrams assigned to each problem diagram.

Whereas requirements describe how the environment should behave once the machine is integrated in it, the specification describes the machine and forms the basis for its construction. Specifications are implementable requirements, and they are derived from the requirements

using the domain knowledge and the assumptions. For the specification, natural language and UML sequence diagrams can be used.

To create the sequence diagrams for each domain which is directly connected to the machine in a problem diagram, a lifeline is drawn in the corresponding sequence diagram. Domains can be merged in the sequence diagram to simplify the description. The machine to be built is also represented by a lifeline in the sequence diagrams. The phenomena are represented by annotated, asynchronous signals between lifelines. For states in the environment of the machine, it should be assumed that an asynchronous signal occurs when the state in the environment changes. To express the coherence between the sequences *state invariants* for the domains in the environment should be included. Appropriate case distinctions according to these state invariants 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. It is important to specify the initialization of the machine. For this diagram a *found signal* (cf. Section 2.5) or a signal from a gate can be used to specify a power on signal. In this step, the events can be refined by adding parameters to phenomena. For most embedded systems precise timing constraints are necessary.

The sequence diagrams should express typical cases with example values. Loops, states, references, and co-regions do not cause any problems, while the other new constructs of UML 2.0 such as parallelism, continuation and considered signals should be used with care. Each diagram represents one concrete interaction sequence. One should not try to make the diagrams too general. It is better to draw further diagrams.

It is difficult to express the specification directly if the machine has a low-level interface to other domains. A display e.g., is controlled by commands that change the color of a dot, or an incremental encoder generates pulses representing a speed. Using the pulses or the commands, the behavior of the machine can not be expressed adequately. In this case, the requirements and the domain knowledge can be expressed as separate sequence diagrams instead of expressing the specification as more complex sequence diagrams.

To check the consistency of the specification and to design a complete specification of the machine, one can continue with the application component for the subproblems of Step 9 and the merged application component of Step 11. This can be done, because the (extended) *four variable model* is applied, and therefore the machine specification derived in this step is already the specification of the application component. Within Step 9, the state machines have to consider all signals in all states, and we can derive a complete specification. When the state machines for subproblems can be merged in Step 11, we can be quite sure that there are no contradicting requirements.

It must be shown that, when the machine fulfills the specifications, then the requirements are satisfied. For that proof, domain knowledge and assumptions can be used in the validation condition $D \wedge A \wedge S \implies R$. $D \wedge A \wedge S$ must be consistent, otherwise everything can be deduced. Additionally, it should be checked that all requirements are captured, and in the sequence diagrams exactly the phenomena of the problem diagram are used. Also the direction of signals must be consistent with the control of the shared phenomena, When a shared phenomena is controlled by one domain, this domain sends the signal or it defines the

return value. The signals must connect the domains as connected in the problem diagram. Each phenomenon at the interfaces of the machine must be used in at least one sequence diagram. The annotated state invariants must allow to combine the sequence diagrams in the same way as the relationships in Step 3 describe.

This step is derived from Step 4 of the process defined in [CHH05b] and Steps 4 and 5 of the process described in Chapter 3. As proposed in Section 4.8, both steps are merged. In contrast to this process, typical cases with example values are described, as suggested in Section 4.4. If appropriate, the specification is expressed using the requirements and the domain knowledge. Using this representation, the problems discussed in Sections 4.9 and 4.11 are treated. In contrast to the process in [CHH05b], assumptions are considered. Additionally, the problem of contradicting requirements and the completeness of the specification described in Sections 4.12 and 4.13 is considered in this step.

Experience from many projects has shown that sequence diagrams can easily be discussed with managers and customers that do not have technical knowledge. The specifications developed in this step can be used as a basis for manual or even automatic tests in Step 11. If requirements and domain knowledge are used to specify the requirements, the requirements will be the test cases, and the domain knowledge describes how to build the test interfaces.

5.5 Step 5: Design Global System Architecture

In Step 5, the system architecture is designed. Table 5.5 shows the input, the output, and the validation activities for this step.

The context diagram, the problem diagrams, the sequences of interactions between machine and environment of all subproblems, and the expression of the subproblem relationships are required for this step.

To design the architecture, the problem diagrams are associated with components. Parallel problems 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. Connection domains being part of a machine will become separate components. The architecture can be specified recursively, i.e., components can have their own architecture, consisting of sub-components.

The system architecture of the embedded system is expressed as a composite structure diagram. This diagram uses parts for the components. When the components are identified, the ports of the components must be connected with interfaces as described in Section 2.6. The connections are used to transmit the signals of the annotated interfaces between the components. The operations in the interfaces can be derived from the phenomena of the problem diagrams as described in Section 2.6. The parameters of the operations for the external interfaces can be extracted from the sequence diagrams. The interfaces between the components that are directly derived from domains must be designed according to the desired functionality of the connected components.

For all subproblems:

input:	context diagram from Step 1 problem diagrams from Step 3 sequences of interactions between machine and environment of all subproblems from Step 4 expression of the subproblem relationships	Jackson Jackson sequence diagrams e.g. grammars
output:	system architecture perhaps subcomponents (recursively) external interfaces interfaces between the components technical description of hardware interfaces expression of the subproblem relationships for all components	composite structure diagrams composite structure diagrams interface classes interface classes natural language, figures process algebra-like notations, grammars, high-level sequence charts, or sequence charts using combined fragments
validation:	all machine interfaces of the problem diagram must be captured the signals in the sequence diagrams must be consistent with the external interfaces for each complex component at least one problem diagram must be associated each problem diagram must be associated to 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 be a separate system or a component	

Table 5.5: Step 5 - Design Global System Architecture

Additional 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 the dependencies between the associated subproblems must be derived from the expression of the subproblem relationships for the whole machine. The developed architecture is the starting point for the further development (hardware- as well as software development).

To validate this step, several activities have to be performed: The external interfaces of the components have to cover the interfaces of all problem diagrams. The architecture must cover all specifications developed in Step 4. Each machine domain in the context diagram must be a separate system or a component. All domains in the problem diagrams being part of the machine must be associated to a component. The description of the relationships between the subproblems for the machine and the components must be consistent.

If only one component exists in the system and no or few causal domains exist, one should continue with the FUSION method [DCJ94].

This step is not defined in the process in [CHH05b], since this process covers no hardware components. This step extends Step 6 described in Chapter 3 by the description of the subproblem relationships as mentioned in Section 4.3. Using the connection domain and the notation for domains being part of the machine introduced in Step 3 of this agenda, the system specification can be developed systematically (cf. Section 4.2).

5.6 Step 6: Derive Specifications for all Components being Relevant for the Subproblem

In this step, the specification of all components of the system architecture is derived. Table 5.6 shows the input, the output, and the validation activities for this step.

To derive the specification of all components, the system architecture with its interfaces from Step 5 and the sequences diagrams from Step 4 are necessary.

The specifications of all components are expressed as sequence diagrams.

The signals specified in the interfaces of the architecture are used to annotate the sequence diagrams. These sequence diagrams are a concrete basis for the test implementation for all components. The sequence diagrams describe the behavior of all components and the interaction between them.

In general, the proceeding can be described as follows:

- Draw a lifeline for all components of the architecture that are necessary to describe the interface behavior of the subproblem and one lifeline for the environment. If the behavior between two components can not be described directly, the components can be merged in the sequence diagram and this behavior can be described separately.

For all subproblems and for all components:

input:	architecture from Step 5	composite structure diagrams
	interfaces from Step 5	interface classes
	subcomponents (if defined) from Step 5	composite structure diagrams
	sequences of interactions from Step 4	sequence diagrams
output:	interface behavior of all complex components (test specification)	sequence diagrams
validation:	sequence diagrams together must describe the same behavior as in Step 4 all signals in the interface classes of Step 5 must be captured in at least one sequence diagram direction of signals must be consistent with the required or provided interfaces of Step 5 signals must connect components as connected in the software architecture of Step 5 it must be possible to map the new state invariants to the state invariants in Step 4	

Table 5.6: Step 6 - Derive Specifications for all Components being Relevant for the Subproblem

- Describe the interface behavior of all components using the signals from the system architecture (Step 5). The behavior must refine the behavior described in Step 4.
- Add state invariants 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 Step 5: Each diagram represents one concrete interaction sequence. One should not try to make the diagrams too general. It is better to draw further diagrams.

The sequence diagrams together must describe the same behavior as in Step 4. The signals at the external interfaces of this step must be the same, have the same direction and the same order. All signals in the interface classes specified in Step 5 must be captured in at least one sequence diagram, and the direction of signals must be consistent with the required or provided interfaces. That means if a component provides an interface, the signals of this interface must be sent to this component in the sequence diagram. This implies also that the signals must connect components as connected in the system architecture. It must be possible to map the new state invariants to the state invariants in Step 4.

If a component needs complex or dynamic data types or operations, this component should be developed with the FUSION method [DCJ94].

This step is not defined in the process in [CHH05b], since this process covers no hardware components. The validation condition of Step 7 of the process described in Chapter 3 is detailed to give concrete guidance for this important activity. The problem identified in Sections 4.5 and 4.4 are taken into account for this step.

5.7 Step 7: Design Software Architecture for each Software Component and each Subproblem

In this step the software architecture for all complex components and all subproblems are derived. Table 5.7 shows the input, the output, and the validation activities for this step.

The system architecture from Step 5 and associated problem diagrams from Step 3 are necessary to design the software architecture. To define the interfaces in the architecture, the signals of the machine behavior sequence diagrams from Step 4 and the interfaces from Step 5 are required. If reusable components from other projects are used, their interfaces must be integrated into the software architecture.

For all subproblems and for all components:

input:	system architecture from Step 5 associated problem diagrams from Step 3 interfaces from Step 5 perhaps reusable components from other projects (Step 11) signals of the machine behavior specifications from Step 4	composite structure diagram Jackson interfaces classes active or passive classes with interface classes sequence diagrams
output:	layered software architecture interfaces between software components	composite structure diagrams interface classes
validation:	if no instantiation: consistent with problem diagram phenomena of sequence diagrams are interfaces of the application layer direction of all signals consistent to each other and input	

Table 5.7: Step 7 - Design Software Architecture for each Component and each Subproblem

The output of this step is a layered architecture. These architectural patterns are not the only possible way to structure the machine domain solving the problem that fits to a given problem frame. However, the layered architecture has proven useful in practice (see for example [CD01, HH05b, Tan92]) and allows for combining solutions to different subproblems of complex problems in a systematic way. It is also flexible enough to be combined with other architectural styles (cf. Section 2.7).

The lowest layer is the *hardware abstraction layer* (HAL). This layer covers all interfaces to the external components in the system architecture and provides access to these components independently of the used controller or processor. For porting the software to another hardware platform, only this part of the software needs to be replaced.

The hardware abstraction layer is used by the *interface abstraction layer* (IAL). This layer provides an abstraction of the (low-level) values yielded by the sensors and actuators. For example, a frequency of wheel pulses could be transformed into a speed value. Thus, in the interface abstraction layer, values for the monitored and controlled variables (see [DLP95]) of the system are computed. It is possible that these variables have to be computed from the values of several hardware interfaces. For safety-critical software components, the interface abstraction layer will usually make use of redundant arrangements of sensors and actuators.

The highest layer of the architecture is the *Application layer*. This layer only has to deal with variables from the problem diagram. Therefore, the system requirements can be directly mapped to the software requirements of the application layer, as described by Bharadwaj and Heitmeyer [BH99].

In the following sections, for the six most important problem frames (cf. 2.4) the corresponding architectural patterns proposed in [HH05a] are presented. 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 the further development.

The architectural pattern shown on the right-hand side of Figure 5.1 represents an adequate structure for the *Control machine* in the ***Required Behavior*** problem frame showed on left-hand side of Figure 5.1. For special kinds of embedded systems, that architecture could be refined. However, a refinement of the architecture would also correspond to a refinement of the corresponding problem frame. The architectural pattern shown here has the same degree of generality as the problem frame.

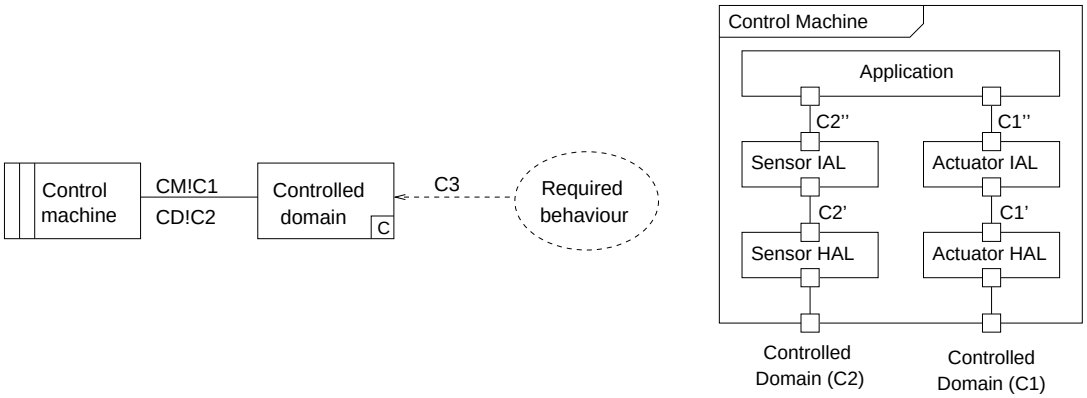


Figure 5.1: Required Behaviour Frame Diagram [Jac01] and Architectural Pattern

For problems fitting to the ***Commanded Behavior*** problem frame (see left-hand side of Fig. 5.2), another architectural pattern can be instantiated.

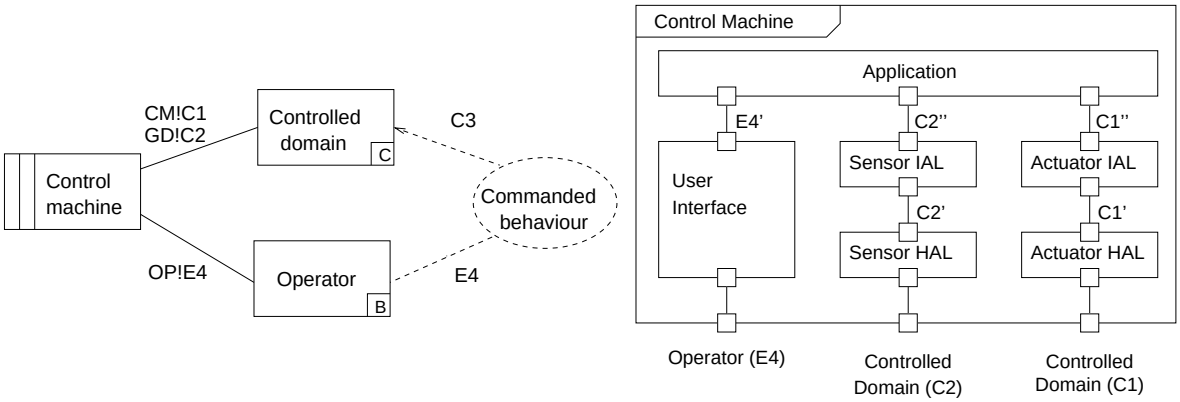


Figure 5.2: Commanded Behaviour Frame Diagram [Jac01] and Architectural Pattern

As can be seen from the frame diagram, the distinguishing feature of the *Commanded Behaviour* frame as compared to the *Required Behavior* frame is the presence of an operator. That distinction is reflected in the corresponding architectural pattern shown on the right-hand side of Figure 5.2. The operator commands and the corresponding feedback are handled by

a dedicated component User Interface. The user interface follows the MVC design pattern [GHJV95]. In contrast to the solution discussed in [RHJN04], the interface to the Application of this component should be the interface to the model, i.e., the User Interface comprises the View and Controller parts of the MVC pattern. With this variation, it can be used in architectural patterns associated with different problem frames.

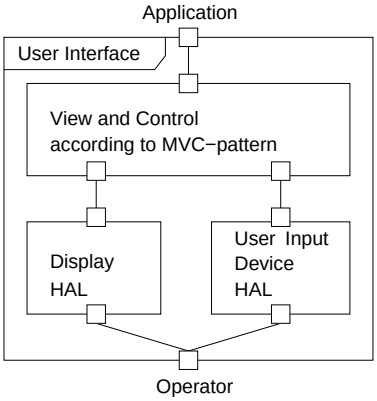


Figure 5.3: Detailed Architectural Pattern for User Interface

Since each architectural pattern corresponding to a problem frame containing an operator domain will contain a user interface component, we give the structure of such a component in more detail (Figure 5.3). For reasons of practicality, the user interface component contains not only a sub-component that serves to read user input via some device. In most cases, a sub-component will also be needed that provides some kind of feedback to the user via a display. The physical input arriving at the port at the bottom of the component is transformed into the more abstract phenomena E4 by the sub-component View and Control.

The ***Information Display*** problem frame offers a structure for applications devoted to the display of real world physical data. The corresponding frame diagram is shown on the left-hand side of Figure 5.4. The interface between the Information machine and the Real world contains only phenomena C1 that are controlled by the real world. This means that the machine cannot influence the real world. Its purpose is only to display things that happen in the real world. Accordingly, the architectural pattern given on the right-hand side of Figure 5.4 does not contain any components for handling actuators, but only components for handling sensors. There is no operator, but a display is needed. Hence, the architectural pattern contains a display interface.

The ***Commanded Information*** frame is presented as a variant of the *Information Display* frame where an operator is added. The architectural pattern proposed for Answering machines that solve a *Commanded Information* problem is shown on the right-hand side of Figure 5.5. To take the presence of an operator into account, the Display Interface component of the architectural pattern for the *Information Display* frame is replaced by a User Interface component.

Moreover, to cover database applications in addition to the operator-controlled display of physical data, the architectural pattern we propose contains a Data Storage component. Of course, this component can be left out if it is not needed to solve the problem. In that case,

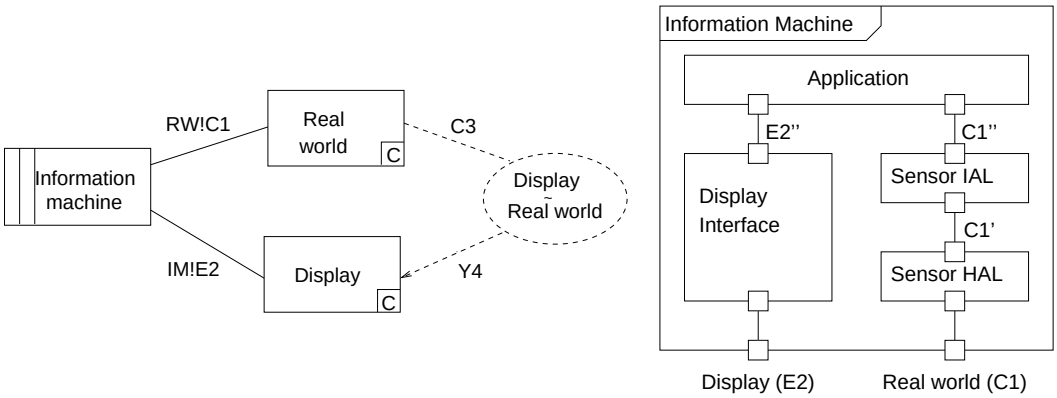


Figure 5.4: Information Display Frame Diagram [Jac01] and Architectural Pattern

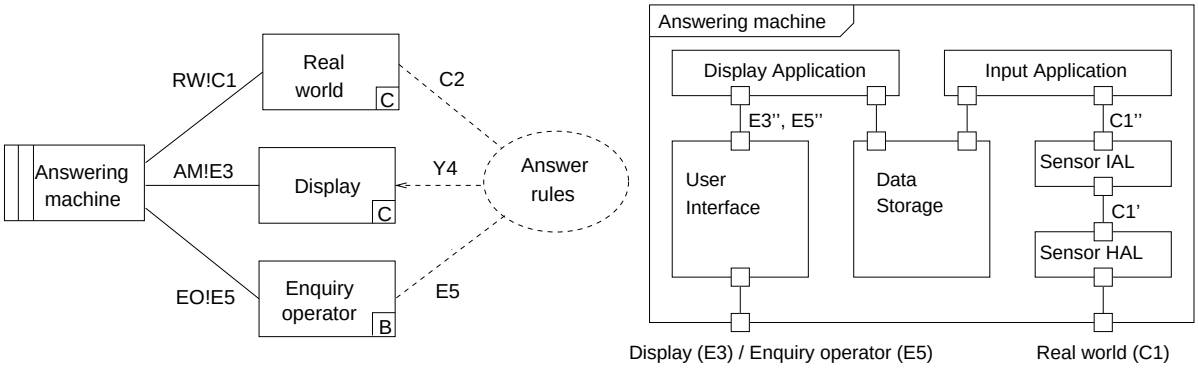


Figure 5.5: Commanded Information Frame Diagram [Jac01] and Architectural Pattern

there would only be one (or even no) application component. Alternatively, for pure database applications, the sensor-handling components of the architectural pattern will not be needed.

For problems fitting to the ***Workpieces*** problem frame, the architectural pattern shown on the right-hand side of Fig. 5.6 can be instantiated. This figure contains a user interface component,

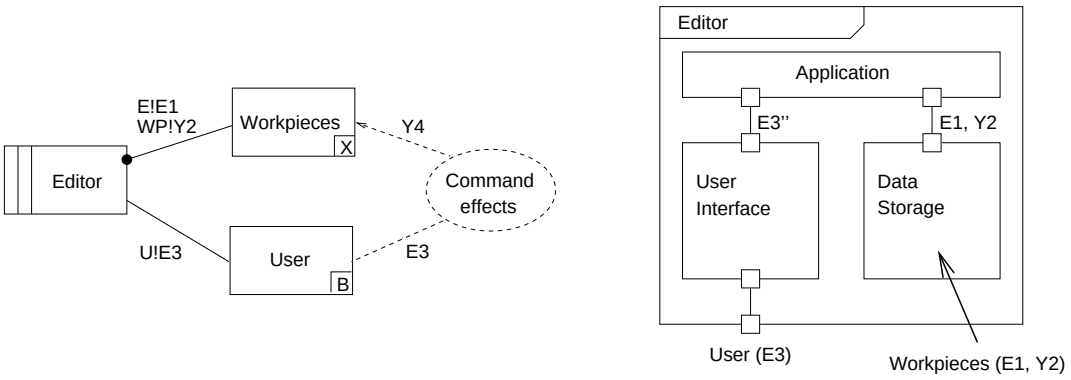


Figure 5.6: Workpieces Frame Diagram [Jac01] and Architectural Pattern

because the problem frame diagram contains a user. The data storage component of the architectural pattern corresponds to the Workpieces domain of the frame diagram. The Application component is responsible for manipulating the data storage according to the user commands. The *Workpieces* problem frame is very similar to a *Database Update* frame [CH04].

Note that there is only one interface with the environment – namely the interface with the user – because the lexical Workpieces domain is part of the machine. This holds true also for the input and output domains of the architectural pattern for transformation problems (see below). Because our user interface component (see Figure 5.3) contains not only input but also output facilities, no change in the architectural pattern is necessary if the problem frame is extended with a feedback for changes on workpieces. Such an extension is necessary for realistic problems and user-friendly applications.

Non-functional requirements might state that distributed access to the workpieces must be provided. Such requirements cannot be expressed in problem diagrams. Nevertheless, they may have an influence on the architectural pattern. For this case, we propose a repository architectural pattern (see left-hand side of Figure 5.7). The repository architectural pattern can be mapped to the layered architectural pattern as shown on the right-hand side of Figure 5.7 (for one client). Here, remote access to the data storage is possible via a network.

If a problem fits to the ***Transformation*** problem frame, the architectural patterns shown in Fig. 5.8 can be instantiated.

Again, this architectural pattern exactly reflects the domains of the frame diagram. Inputs and outputs are stored in data storage components, and the application component is responsible for transforming inputs into outputs. In this architectural pattern, there are two data storages. They represent persistent data of perhaps different structure. One is for the inputs (e.g., source code), the other for outputs (e.g., an executable file).

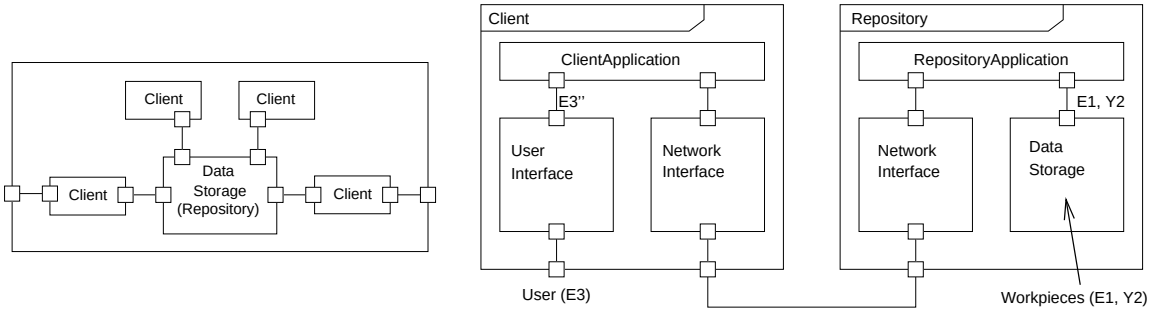


Figure 5.7: Architectural Pattern for Remote Access to Data Storage

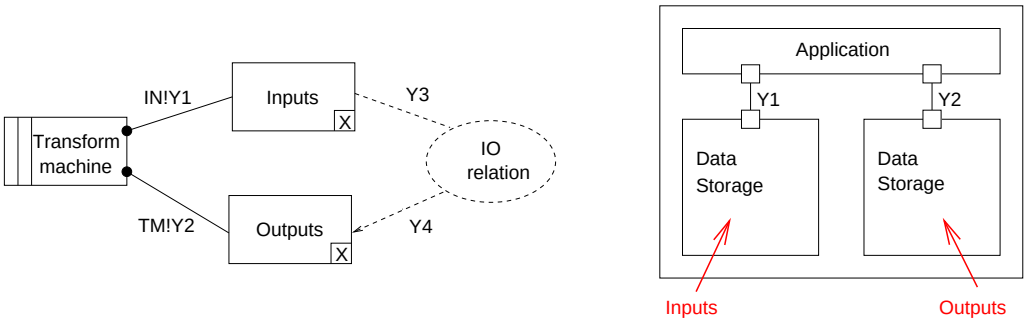


Figure 5.8: Transformation Frame Diagram [Jac01] and Architectural Pattern

If a subproblem is unrelated to any problem frame, then a corresponding architecture has to be developed from scratch. The following proceeding can be applied to develop a layered architecture:

- The interfaces of the architectural patterns correspond exactly to the interfaces of the machine domains as defined in the different frame diagrams. Hence, the architecture refines exactly the machine to build; it neither adds nor leaves out any shared phenomena as compared to the problem description.
- If the machine has interfaces with causal domains, the corresponding architectural pattern contains 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 architectural pattern contains a user interface component.
- If the machine has interfaces with lexical domains, these domains are reflected as parts of the corresponding architectural pattern. This must be the case, because lexical domains can only exist inside the machine.
- Components for data storage should only by included if the data is stored persistently. Otherwise they can be assumed to be part of some other component.

When the architecture is designed, the interface classes must be specified. The interfaces to the hardware are the same as in the system architectures between the components.

Since we use descriptions from the software technology to describe the hardware interfaces, the interfaces between IAL and HAL are the same in most cases. Only if a detailed technical description of the external interfaces is provided, the interface between IAL and HAL is the same as the external interface. 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.

If the interface of the application layer is the same as the interface of the HAL, the IAL can be removed from the architecture. If a component is too complex, the component should be split into subcomponents.

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.

If the architectural diagrams are instantiations of the given patterns, no validation of the architecture is necessary. Otherwise, it must be checked that all domains of the problem diagram of Step 3 are captured in the architecture and that the external interface of the architecture coincides with the machine interface of the problem diagram.

All interfaces must be covered. The signals or operations in the external interface classes of the software architecture must be the same as the signals in the sequence diagrams of Step 6. The interface of the application components must contain the signals used in the sequence diagrams of Step 4. Additionally, the directions of all signals are consistent to each other and to the input.

In the process described in Chapter 3, the architecture for all subproblems is developed together in Step 8. To support a structured development of the software architecture, for each subproblem one architecture is developed as suggested in Step 4.6 and described in [CHH05b], Steps 5 and 6. These steps are combined here, because the interfaces between the components are an elementary part of architectures. Also the problem in Section 4.7 is covered in this process.

5.8 Step 8: Specify Behavior of Software Architecture Components for each Subproblem

In this step, the specifications of all components of the software architecture are derived. Table 5.8 shows the input, the output, and the validation activities for this step.

To derive the specifications of all software components, the software architectures with its

For all subproblems and for all software components:

input:	software architectures from Step 7	composite structure diagrams
	interfaces from Step 7	interface classes
	system behavior from Step 4	sequence diagrams
	interface behavior of all complex components from Step 8	sequence diagrams
output:	interface behavior of all components (test specification)	sequence diagrams
validation:	all sequence diagrams together must describe the same behavior as in Step 6 all signals in the interfaces classes of Step 7 must be captured in at least one sequence diagram direction of signals must be consistent with the required or provided interfaces of Step 7 signals must connect components as connected in the software architecture of Step 7 it must be possible to map the new state invariants to the state invariants in Step 6	

Table 5.8: Step 8 - Specify Behavior of Software Architecture Components for each Subproblem

interfaces from Step 7, the sequences diagrams from Step 4 for the application components, and the components interface behavior from Step 8 are necessary.

Outputs of this step are specifications of all software components, expressed as sequence diagrams. A specification expressed somewhere else should be referenced and does not have to be translated into sequence diagrams.

The signals specified in the interfaces of the software architectures are used to annotate the sequence diagrams. The sequence diagrams developed in this step are a concrete basis for the test implementation for all software components. In this step, each component is described separately. In general, the proceeding can be described as follows:

- Draw a lifeline for the software component that should be specified. Either a lifeline for the environment should be introduced or the messages from the environment should be drawn to 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 the same as in Step 4.
- The specification for the interfaces abstraction layer can be derived from the domain knowledge expressed as sequence diagrams.
- The specification for the hardware abstraction layer should show the mapping from the software to the interfaces of the machine. Since this specification is usually described in the data book of the target system, only a reference must be given.
- If possible refer, to other sequence diagrams and do not draw diagrams for the same aspect several times.
- Add state invariants where they are relevant to describe the behavior.
- Add missing sequence diagrams to describe the behavior for all relevant states.
- Add timing constrains if necessary.

The sequence diagrams together must describe the same behavior as in Step 8. The signals at the external interfaces of this step must be the same, have the same direction, and have the same order as in Step 8. All signals in the interface classes, specified in Step 7 must be captured in at least one sequence diagram. The direction of signals must be consistent with the required or provided interfaces (cf. Step 6). It must be possible to map the new state invariants to the state invariants in Step 6.

As stated in Section 4.10, this step is missing in the process described in Chapter 3. This step is derived from Step 7 in [CHH05b] and is extended with more detailed guidance how to perform this step.

5.9 Step 9: Specify Software Components of Software Architectures for each Subproblem

In this step, the specifications of all components of software architecture are derived. Table 5.9 shows the input, the output, and the validation activities for this step.

For all subproblems and for all software components:

input:	interface behavior from Step 8	sequence diagrams
output:	component overview description with ports and lollipops data types and operations defined using pre- and postconditions state machine invariants	class diagram reference to interface classes class diagrams formulas or natural language state machine diagram formulas or natural language
validation:	consistent with interface behavior from Step 8 completeness of state machines (implies error-cases for user-interaction) interface classes must be the same as in Step 7	

Table 5.9: Step 9 - Specify Software Components of Software Architectures for each Subproblem

The sequence diagrams from Step 8 (describing the interface behavior of the software components) are necessary to design the software component for each subproblem.

Outputs of this step are specifications of all software components. In contrast to Step 8 the behavior must be described completely for each subproblem. It is not enough to express typical interactions as sequence diagrams.

Instead, a component overview description should be developed and expressed using a class diagram. With this class diagram is expressed if the component is an active or a passive component. An active class behaves like hardware, it may contain timers, it work in parallel to its environment, and it may contain other passive or active classes (see composite structure diagram). A passive class cannot contain timers, its functionality is executed in the time context of an active class, and it can only contain other passive classes. The ports of this class are associated with interface classes. An interface can be provided or required. The reference to the interfaces can expressed using the lollipop-notation (cf. Section 2.6).

To each class several data types can be assigned. Simple data types can be used directly. More complex data type should be defined with class diagrams. All operations of these classes must be specified using pre- and postconditions.

For each class a state machine should be designed. This state machine can make use of the data types specified in this step. It uses the signals in the provided interfaces as trigger for the transitions and the signals in the required interfaces as outputs.

If possible, invariants about states and the data should be added.

Usually the overview description of a component is the same for all subproblems and therefore only one overview description should be drawn. If also the behavior of one component is the same for several subproblems, only one state machine should be created.

In general, the proceeding for all components in the software architecture can be described as follows:

- Draw an active (e.g., behaves like hardware, includes a clock) or passive (e.g., calculation-routine) class with its interfaces as a component overview description.
- Add necessary data to this class.
- In case of complex data or complex operations on data types: add classes for data types.
- Specify pre- and postconditions for all operations.
- Design a state machines that implements the behavior of all sequence diagrams specified in Step 8.
- Complete the state machines, i.e. there must be a specified reaction to each possible input signal.
- Add invariants if possible.

Each architectural component must be covered, and each state machine is *complete*, i.e., each possible input signal (as specified in Step 7) is taken into account. Each state machine must behave as described in its corresponding sequence diagrams. Moreover, all referenced interface classes must be the same as the interface classes of the subproblem architecture of the respective component (Step 7).

This step contains the same activities as in Step 9 of the process described in Chapter 3, but here they are performed only for one subproblem. It is derived from Step 8 in [CHH05b]. The advantage of introducing this step is that the solution for subproblems can be reused and that this step simplifies the composition of the subproblems within the next step.

5.10 Step 10: Develop Global Software Architectures

In this step, the composed software architectures for all complex components are developed. Table 5.10 shows the input, the output, and the validation activities for this step.

To develop the global architecture for one hardware component, all software architectures for this hardware component and their interfaces are necessary. The relationships between subproblems help to perform this step systematically.

Within this step, the architectures and interfaces of all subproblems for one component are composed and expressed as composite structure diagrams.

For all components:

input:	layered software architectures from Step 5	composite structure diagrams
	interfaces between software components from Step 5	interface classes
	relationships between subproblems specified in Step 5	e.g. grammars
output:	layered software architecture	composite structure diagrams
	interfaces between software components	interface classes
validation:	architecture must contain all components and interfaces of all subproblem architectures from Step 5 external interfaces must be consistent with the interfaces of the context diagram developed in Step 1	

Table 5.10: Step 10 - Develop Global Software Architecture for each Component

The crucial point of this step is to decide if two components contained in different subproblem architectures should occur only once in the global architecture, i.e., they should be merged. To decide this question, we make use of the information gathered when decomposing the overall problem into subproblems. We distinguish the following cases, where all cases but the first one concern application components:

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.

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.

The global architecture must contain all components and interfaces of all subproblem architectures. Its external interfaces must be the same as in the system architecture developed in Step 5.

This step has the same output as Step 8 of the process described in Chapter 3. But the procedure is taken from Step 9 in [CHH05b], with the difference that this architecture is embedded in the system architecture and not in the context diagram. This step helps to solve the subproblem composition problem, described in Section 4.3. It improves the reuse of architectures for subproblems.

5.11 Step 11: Specify Composed Software Components

In this step, the components of the software architecture for the subproblems are composed. Table 5.9 shows the input, the output, and the validation activities for this step.

For all software components:

input:	component overview description with ports and lollipops from Step 9 data types with operations from Step 9 defined using pre- and postconditions state machine from Step 9 invariants from Step 9 relationships between subproblems specified in Step 5	class diagram reference to interface classes class diagrams formulas or natural language state machine diagram formulas or natural language e.g. grammars
output:	component overview description with ports and lollipops data types with operations defined using pre- and postconditions state machine invariants	class diagram reference to interface classes class diagrams formulas or natural language state machine diagram formulas or natural language
validation:	consistent with interface behavior from Step 8 completeness of state machines (implies error-cases for user-interaction) interface classes must be the same as in Step 7	

Table 5.11: Step 11 - Specify Composed Software Components

The complete output of Step 9 and the relationships between subproblems specified in Step 5 are necessary to perform this step.

The output of this step is the same as for Step 9 with the difference that all items refer to the composed component and not only to a component for one subproblem.

The component overview description, expressed as a class diagram with ports and lollipops, can be merged with the same procedure as described in Step 10. For data types the developer has to decide if the same data is described in two components of different subproblems of if different data is described. If different data is described, both class diagrams for the data will be in the merged component. If the same data is described, a mapping must be created to transform the operations and attributes to a common representation. The merged operations must have the weakest precondition and the strongest postcondition. It refines both operations of the subproblem components. The state machine diagrams can be merged according to the case distinction we made in Step 10:

- Case 1. 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. The composition can be achieved by using composite states. The connecting arcs between the sub-automata depend on the problem.
- Case 3. Here, the merge depends on the problem to be solved. Often there will be a priority between the different subproblems that has to be taken into account when defining the common state machines. As a heuristic, we can note that priorities between subproblems will be necessary when the two subproblems constrain the same domain.
- Case 4. The merge has to be performed manually.

The invariants of all subproblems must also hold for the composed component. Possibly, additional invariants can be found in this step.

To validate this step, 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.

This step has the same output as Step 9 of the process described in Chapter 3. To solve the composition problem described in Section 4.3, the procedure in this step is completely different: The software components are developed from software components of the subproblems and not directly from the sequence diagrams. This approach was taken from Step 10 in [CHH05b]. Additionally, the operations and private data types are merged in this step. This merge was suggested to be Step 11 in [CHH05b]. The merge of Steps 10 and 11 was done to be symmetric with Step 9. Within this step, inconsistencies in the different specifications can be detected, as introduced in 4.12.

5.12 Step 12: Implement Software Components and Test Environment

In this step, the software components and the corresponding test cases are implemented. Table 5.12 shows the input, the output, and the validation activities for this step.

input:	output of Step 11 software component behavior from Step 8 expression of the subproblem relationships from Step 5	*different notations* sequence diagrams e.g. grammars
output:	implemented software components test software for software component	programming language programming language or test language
validation:	run tests	test results

Table 5.12: Step 12 - Implement Software Components and Test Environment

To implement the software components, the output of Step 11 is necessary. The test environment can be created with the software component behavior from Step 8 and the expression of the subproblem relationships from Step 5.

The system 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, function calls, asynchronous messages, or hardware access.

This development process allows developing statically linked software components with the capability of reuse.

In general, the proceeding for object oriented programming languages can be described as follows:

1. Create interface classes for all internal interfaces (also for subcomponents).
2. Create classes for all (sub-)components and implement them.
   a) Implement actions as private methods according to the pre- and post-conditions.
   b) Implement the state machine.
   c) Implement the active classes with threads or timer libraries.
   d) Check all classes if there is a concurrent access to variables and resolve this problem with synchronization statement.
3. Implement test cases for all components (except HAL) according to the sequence diagrams from Steps 5 and 7. Time frames must be added, specifying when a signal is expected to occur.
4. An independent person should develop and implement further test cases.
5. Run test cases.

The validation of this step is performed by running the test cases.

This step is derived from Step 10 of the process described in Chapter 3 and Step 11 in [CHH05b]. Since in this process a specification of the software components is explicitly developed, this specification can be used to check the implemented software components. The integration of these software components is performed in the next step.

5.13 Step 13: Integrate Software Components

In this step, the software components are integrated and the test cases are implemented. Table 5.13 shows the input, the output, and the validation activities for this step.

input:	software architecture from Step 10 software behavior from Step 6 expression of the subproblem relationships from Step 5	composite structure diagram sequence diagrams e.g. grammars
output:	implemented software test software for software	programming language programming language or test language
validation:	run tests	test results

Table 5.13: Step 13 - Integrate Software Components

To integrate the software components, the output of Step 11 is necessary. The test environment can be created with the software component behavior description from Step 6 and the expression of the subproblem relationships from Step 5.

Within this step, the components have to be connected as specified in Step 8. For an object oriented implementation, a method to initialize all objects according to the architecture from Step 8 must be created. Test interfaces have to be implemented, and the test cases have to be implemented as specified in Step 6. The expression of the subproblem relationships should be used to create concrete test scenarios. An independent person should develop and implement further test cases.

To validate the results of this step, tests may be run in an emulation environment.

This step is derived from Step 10 of the process described in Chapter 3 and Step 11 in [CHH05b]. Within this step only, the integration of the software components is performed.

5.14 Step 14: Integrate Hardware and Software

In this step, all components are integrated and the acceptance test cases are performed. Table 5.14 shows the input, the output, and the validation activities for this step.

The system is built according to the system architecture from Step 5. The test cases for the acceptance test are derived from system specification from Step 4 and the expression of the

input:	system architecture from Step 5 system specification from Step 4 expression of the subproblem relationships from Step 3	composite structure diagram sequence diagrams e.g. grammars
output:	integrate system acceptance test cases	machine test system and/or test plans
validation:	run tests	test results

Table 5.14: Step 14 - Integrate Hardware and Software

subproblem relationships from Step 3.

Within this step, the hardware and software components have to be integrated. The test of the whole embedded system, consisting of hardware as well as software, is performed. In general, the proceeding can be described as follows:

- Connect the components as specified in Step 5.
- Load software into targets (microcontroller).
- Perform manual tests.
- Build test environment for automated test. The specification of the test interfaces can be derived from the domain knowledge used to derive the specification in step 4.
- Implement test cases for the whole machine according to the sequence diagrams from Step 4.
- Connect test cases as specified in the expression of the subproblem relationships.

The acceptance test should not be done by the developer. Therefore, the test environment can be developed in parallel to the last Steps. The test environment has to interact with the external interfaces of the machine. Hence, the technical interfaces also consist of hardware.

To validate the results of this step, the tests have to be executed.

This step is derived from Step 11 of the process described in Chapter 3 and Step 11 in [CHH05b]. Within this step, only the integration of hardware and software is performed.

6 Case Study: Traffic Light Control

In this chapter, the development process is applied to a *Traffic Light Control* case study. The case study is taken from [HH05b], extended with additional safety requirements and adopted on the development process presented in Chapter 5.

The system mission of *Traffic Light Control* can be stated as follows:

SM1: The traffic lights should prevent accidents on the crossing.

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

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

6.1 Step 1: Describe Problem

6.1.1 Context Diagram of System in Use

Since no machine domain to be replaced exists for the crossing, a context diagram of the system in use is not created. All relevant existing domains are included in the context diagram of the system to be built.

6.1.2 Context Diagram of System to be Built

Figure 6.1 shows the context diagram for the traffic light control.

6.1.3 Requirements

The traffic light control has to fulfill the following requirements:

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.

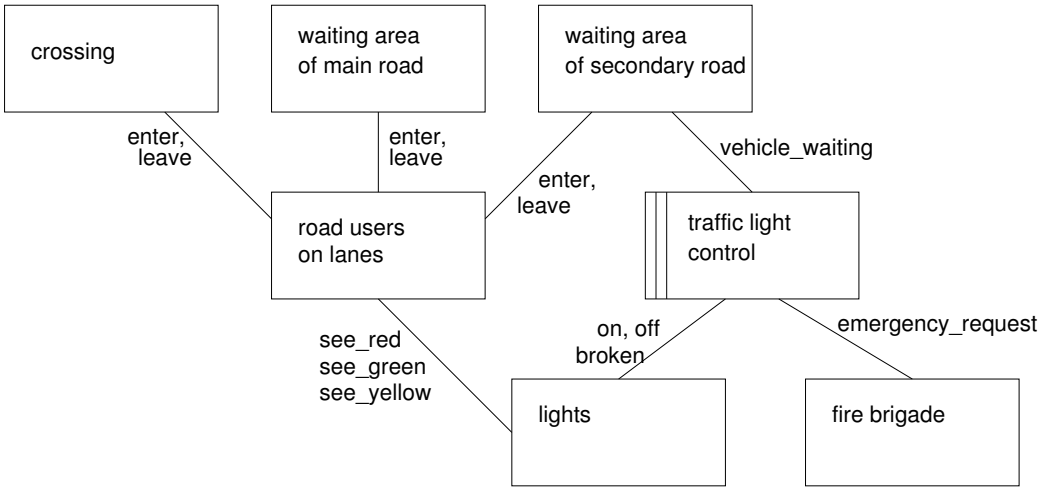


Figure 6.1: Context Diagram for the Traffic Light Control

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]).

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" and yellow blinking[2]) 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 *[3].

6.1.4 Domain Knowledge

The following domain knowledge can be stated:

D1: Traffic rule: stop if red.

D2: Traffic rule: cross if green.

D3: Traffic rule: leave critical section as fast as possible. *

D4: 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.*

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

D5: Traffic rule: if yellow: stop if possible.

D6: Vehicles can not stop immediately without entering the critical section.

D7: A broken light bulb can be detected by measurement of the electric current (no current = no light).

D8: There is a bridge for pedestrians.

D9: Induction loops are used for observing the secondary road request.

6.1.5 Glossary

To define some vocabulary of the traffic light context, references to Fig. 6.2 are used in the following glossary:

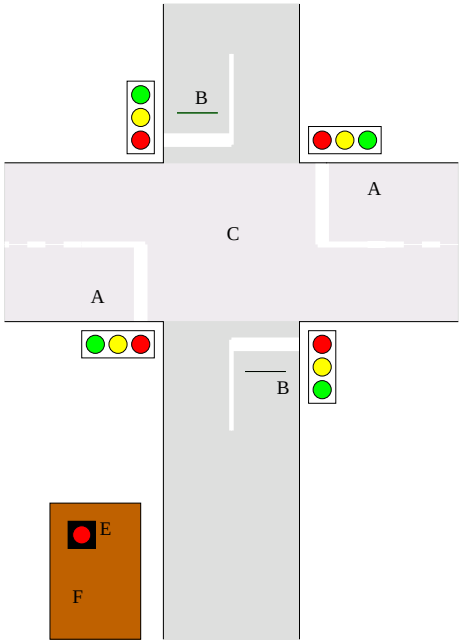


Figure 6.2: Glossary Extension for Traffic Light

- 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

- secondary road request: sensor detecting if a vehicle is in the waiting area of the secondary road
- fire brigade emergency request: button that will be pressed in case of emergency and will be released when all cars of the fire brigade passed the crossing: E
- accident: 2 vehicles at the same time at the same place
- read users on lanes = vehicles on one road = flow of traffic
- vehicles on the main road pass the waiting area of main road

6.1.6 Assumptions

A1: All vehicles follow the traffic rules.

A2: Pedestrians use the bridge.

A3: In case of emergency the button is pressed and released after crossing. *

A4: The sensors do not miss any vehicle waiting on the secondary road. *

6.1.7 Validation

The crossing is referenced in $D4$ and $D6$. The waiting area of main road is referenced in $R3$. The waiting area of secondary road is referenced in $R1$, $D9$, and $A4$. The road users on lanes is referenced in $R1$, $R2$, $R3$, $R4$, $R7$, $D1$, $D2$, $D3$, $D5$, $D6$, $A1$. The fire brigade with its emergency button is referenced in $R2$, $R3$, $A3$. The lights are referenced in $R5$, $R6$, $D7$.

The traffic light control is not referenced in any requirement, domain knowledge, or assumption because the environment, not the machine is describe in this step.

Additionally, pedestrians are referenced in $D8$ 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 all requirements, domain knowledge and assumptions only elements of the context diagram are referenced.

6.2 Step 2: Consolidate Requirements

To consolidate the requirements, for each system mission statements it is investigated, which requirements are necessary for this system mission.

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 are:

- R4 (at most one road may pass)
- R5 (yellow before red, red/yellow before green)
- R7 (both roads get "stop" signal for some time)

Sufficient are: $(R4 \wedge D1 \wedge A1) \wedge (D6 \wedge R5 \wedge R7 \wedge D3 \wedge D5 \wedge A1) \Longrightarrow SM1$.

That means that only one road may pass, the vehicles have to stop if red and they follow rules. Because vehicles cannot stop immediately, a correct order of signaling and a period with red for all is necessary together with the rules to leave the critical section as fast as possible and stop on yellow if possible. If the vehicles follow the rules $SM1$, is fulfilled.

SM2: priority for fire brigade

This system mission statement is achieved by the emergency button. The requirement $R2$ (emergency button yields "go" signal for secondary road) is necessary.

Sufficient are: $R2 \wedge D1 \wedge D2 \wedge A3 \wedge A1 \Longrightarrow SM2$.

That means that the emergency button stops the traffic on the main road, the vehicles have to stop if red and drive if green, and the button is pressed on emergency. If the vehicles follow the rules, $SM1$ is fulfilled.

SM3: fair traffic flow

Requests from the secondary road must be taken into account, but the main road should be allowed to pass twice as long as the secondary road. Necessary are $R1$ and $R3$.

Sufficient are: $R1 \wedge R3 \wedge D4 \Longrightarrow SM3$.

The requirements that the secondary road request leads to "go" and a longer period for main road should be achieved, together with the definition of fairness in $D4$ is sufficient to fulfill $SM3$.

Determine Set of Most Important Requirements

The validation results of this step can be summarized in a set (R') of mission critical requirements:

$R' = \{R1, R2, R3, R4, R5, R7\}$
$R \setminus R' = \{R6\}$

$R6$ is required for safety, and the system mission has to be extended by a corresponding system mission $SM4 \iff R6$. Hence, all requirements are necessary and will be implemented ($R' = R$).

6.3 Step 3: Decompose Problem

The traffic light control consists of the subproblems *MainRoadPassing*, *SecondaryRoadPassing*, *EmergencyRequestSecondaryRoadPassing*, and *BrokenLightSafeState*. The following figures provide the problem frame instances for these subproblems.

6.3.1 Subproblem: SecondaryRoadPassing

Fig. 6.3 shows the subproblem diagram for the secondary road passing phase of the traffic light.[4] The problem diagram presented here only covers the secondary road passing phase. It is an instantiation of the required behavior problem frame. All requirements assigned to this subproblem are relevant for the secondary road passing phase and the main road passing phase.

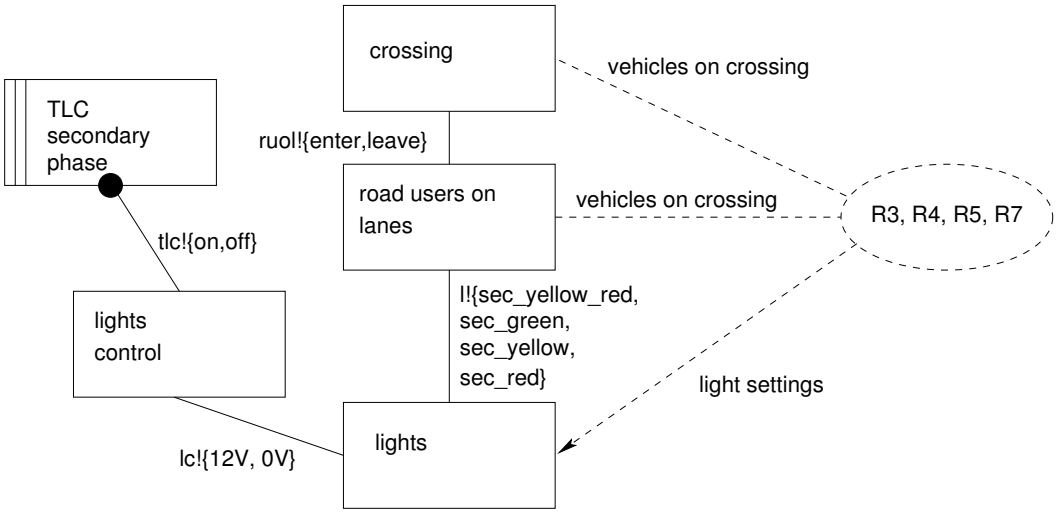


Figure 6.3: Problem Diagram for SecondaryRoadPassing

The following projection operators have been applied:

[4]The subproblems *prevent accidents* and *fair and adapted flow of traffic* in [HH05b] are decomposed differently since a sequential decomposition is possible.

- The domains waiting area of main road, waiting area of secondary road, fire brigade, and the corresponding interfaces are left out.
- The domain lights is divided into light and light control.
- 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 and reduced. It only contains the phenomena relevant for the secondary road passing phase.

6.3.2 Subproblem: MainRoadPassing

Fig. 6.4 shows the subproblem diagram for the main road passing phase of the traffic light. It is an instantiation of the commanded behavior problem frame. The domain road users on lanes is the biddable domain in the problem frame.

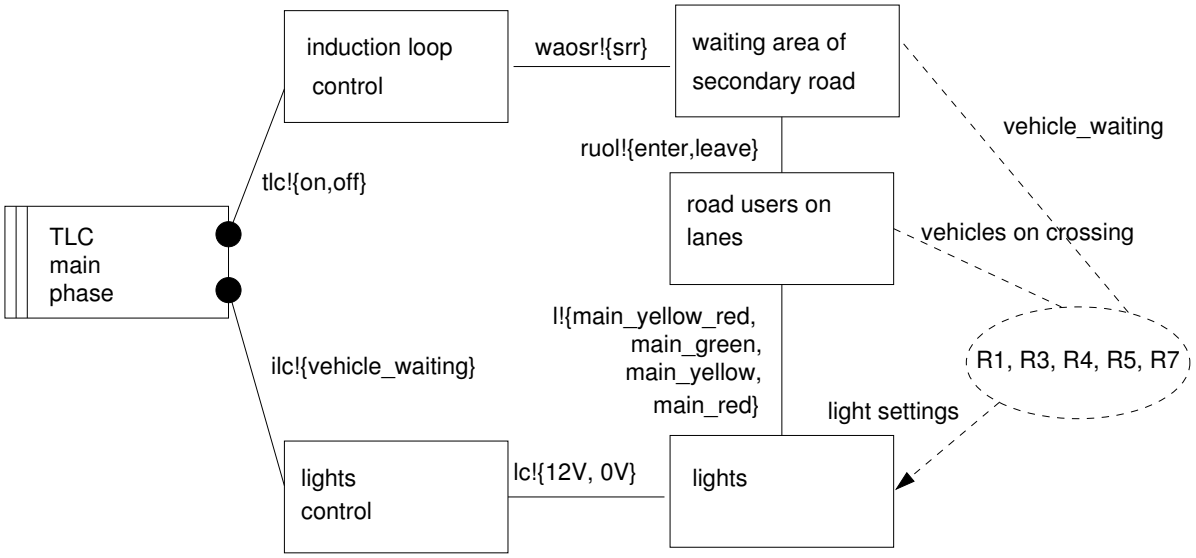


Figure 6.4: Problem Diagram for MainRoadPassing

The following projection operators have been applied:

- The domains crossing, waiting area of main road, 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 and reduced. It only contains the phenomena relevant for the main road passing phase.

6.3.3 Subproblem: EmergencyRequestSecondaryRoadPassing

Fig. 6.5[5] shows the subproblem diagram for the emergency phase of the traffic light [6]. It is an instantiation of the commanded behavior problem frame. The domain fire brigade is the biddable domain in the problem frame.

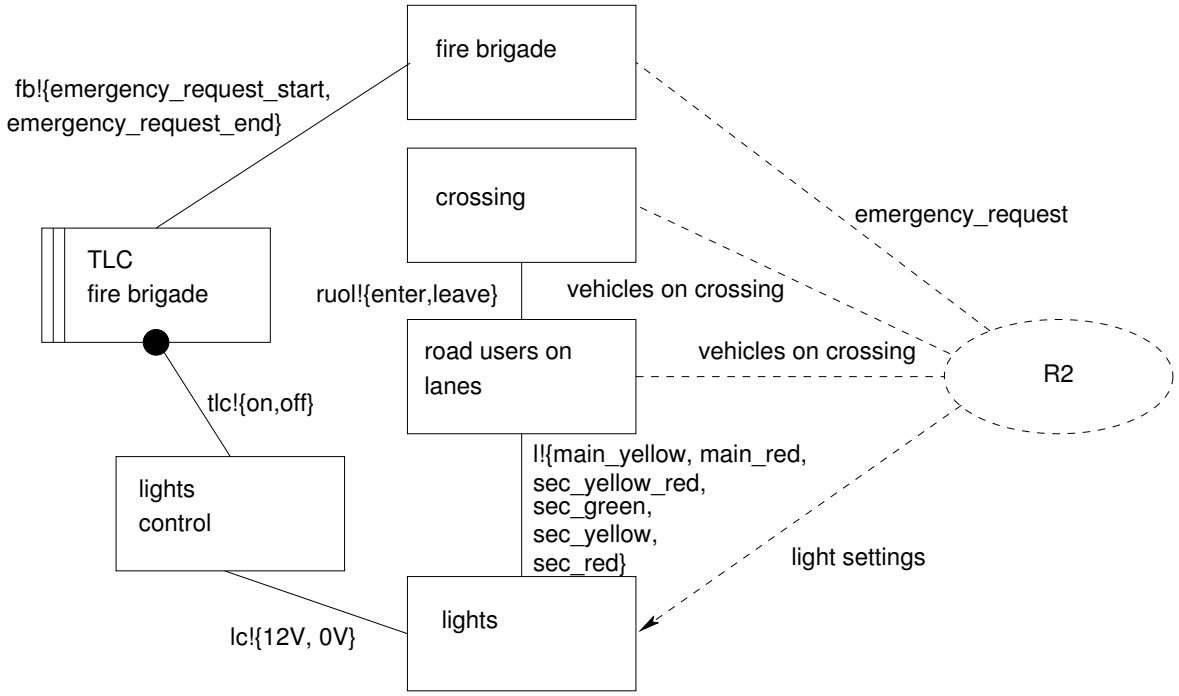


Figure 6.5: Problem Diagram for EmergencyRequestSecondaryRoadPassing

The following projection operators have been applied:

- The domains waiting area of main road, road users on lanes 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 and reduced. It only contains the phenomena relevant for the emergency phase.

[5] Some phenomena between lights and road users on lanes have been added later
[6] Same as the subproblem *help fire brigade* in [HH05b]

6.3.4 Subproblem: BrokenLightSafeState

Fig. 6.6[7] shows the subproblem diagram for the broken light phase of the traffic light[8]. It is an instantiation of the required behavior problem frame.

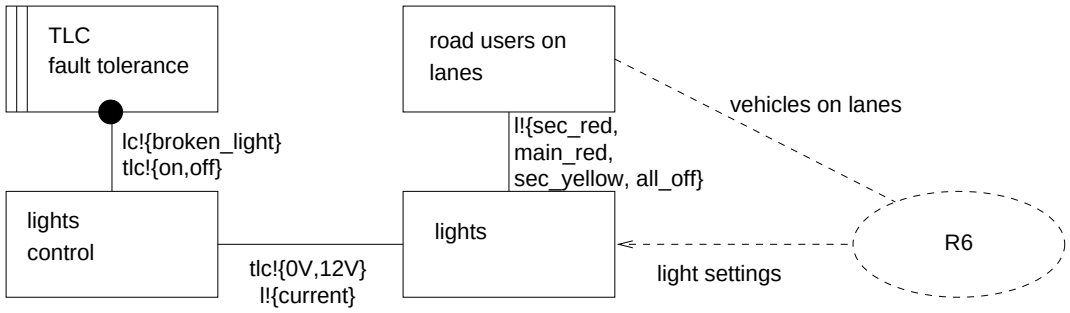


Figure 6.6: Problem Diagram for BrokenLightSafeState

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 and reduced. It only contains the phenomena relevant for the emergency phase.

6.3.5 Validation

The problem diagrams are consistent with the context diagram because 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.

6.3.6 Dependencies between Subproblems

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.

$$
\begin{array}{lll}
< start > & ::= & (< main\_passing > || < fire > || < broken\_light >) \\
< main\_passing > & ::= & (MainRoadPassing < sec\_passing >) \\
< sec\_passing > & ::= & (SecondaryRoadPassing < main\_passing >) \\
< fire > & ::= & EmergencyRequestSecondaryRoadPassing < main\_passing > \\
< broken\_light > & ::= & BrokenLightSafeState
\end{array}
$$

[7] Some phenomena between lights and road users on lanes added later
[8] Same as the subproblem *safe state if light bulb is broken* in [HH05b]

The subproblems *EmergencyRequestSecondaryRoadPassing* acts in parallel to the subproblems *MainRoadPassing* and *SecondaryRoadPassing* in the sense of reacting to phenomena controlled by the environment. Once activated, the subproblem *EmergencyRequestSecondaryRoadPassing* has priority. That implies, only the machine for *EmergencyRequestSecondaryRoadPassing* 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 *EmergencyRequestSecondaryRoadPassing*, *MainRoadPassing*, and *SecondaryRoadPassing* in the same sense. Once activated, only the machine for *BrokenLightSafeState* is allowed to control the lights.

6.4 Step 4: Derive Machine Behavior Specifications for each Subproblem

For each problem diagram, the specification is expressed by sequence diagrams that are given in the following figures. Since it is difficult to express the specification directly, the requirements and the domain knowledge are expressed separately.

6.4.1 Subproblem: SecondaryRoadPassing

Fig. 6.7 shows the first sequence diagram for the subproblem *SecondaryRoadPassing*. The domains crossing and road users on lanes are merged. The domains TLC secondary phase, lights control, and lights are also merged. In this step the requirement is refined by adding timing constrains, e.g., the state SECONDARY PASSING should take 10 seconds.

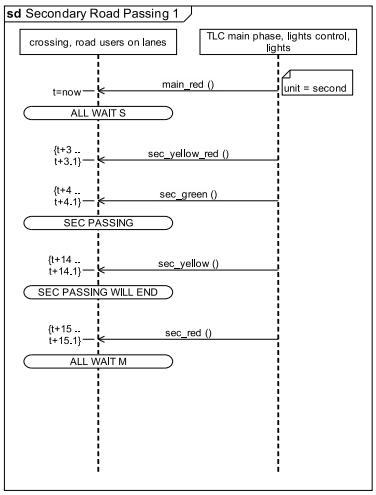


Figure 6.7: Sequence Diagram for SecondaryRoadPassing 1

6.4.2 Subproblem: MainRoadPassing

Fig. 6.8 shows the first sequence diagram for the subproblem *MainRoadPassing*. The sequence diagrams start with a signal instead of a state invariant. This signal is only included to have a time reference for the next signals. For this subproblem the same domains as for the other subproblems are merged. Additionally, the domain induction loop control is part of the machine and therefore merged with it.

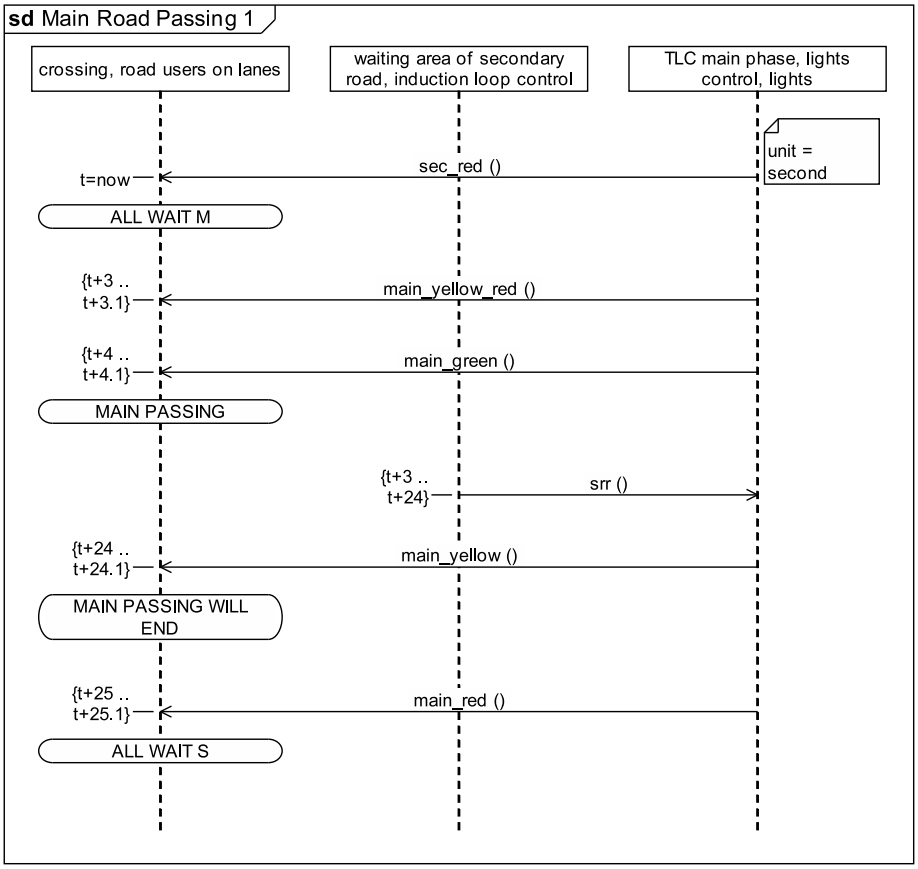


Figure 6.8: Sequence Diagram for MainRoadPassing 1

Fig. 6.9 shows the second sequence diagram for the subproblem *MainRoadPassing*. The sequences express that the state MAIN PASSING takes at least 20 seconds, and therefore the requirement $R3$ and $D4$ are considered.

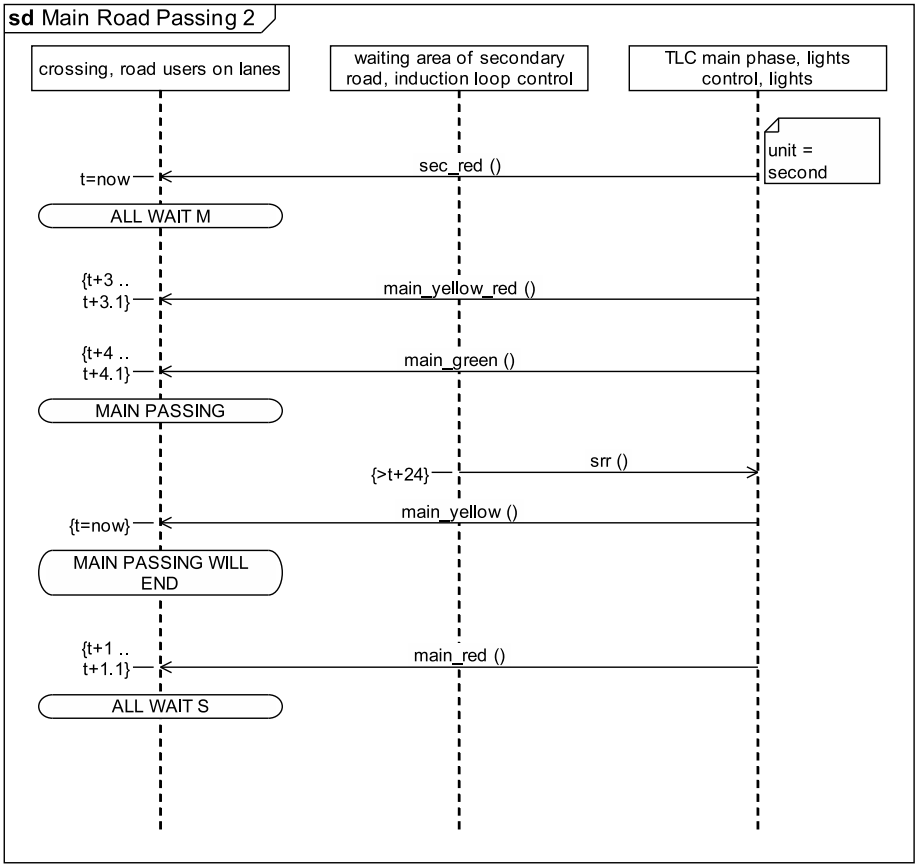


Figure 6.9: Sequence Diagram for MainRoadPassing 2

6.4.3 Subproblem: EmergencyRequestSecondaryRoadPassing

The Figures 6.10, 6.11, 6.12, 6.13, and 6.14 show the sequence diagrams for the subproblem *SecondaryRoadPassing*. The star (*) indicates that the diagram can be applied for all states, whose name begins with the given string. For this subproblem the same domains as for the other subproblems are merged. The signal *sec_yellow_red()* in Fig. 6.12 is only included to annotate the timing invariant.

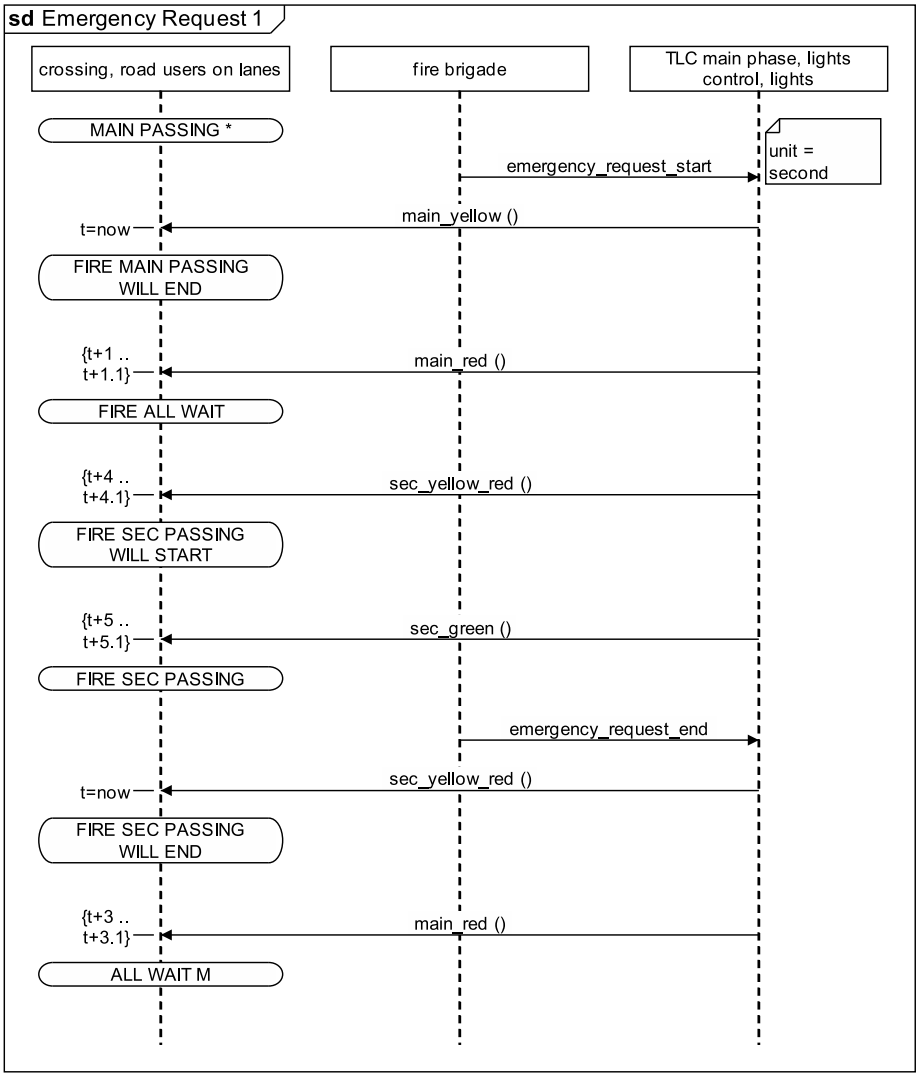


Figure 6.10: Sequence Diagram for EmergencyRequestSecondaryRoadPassing 1

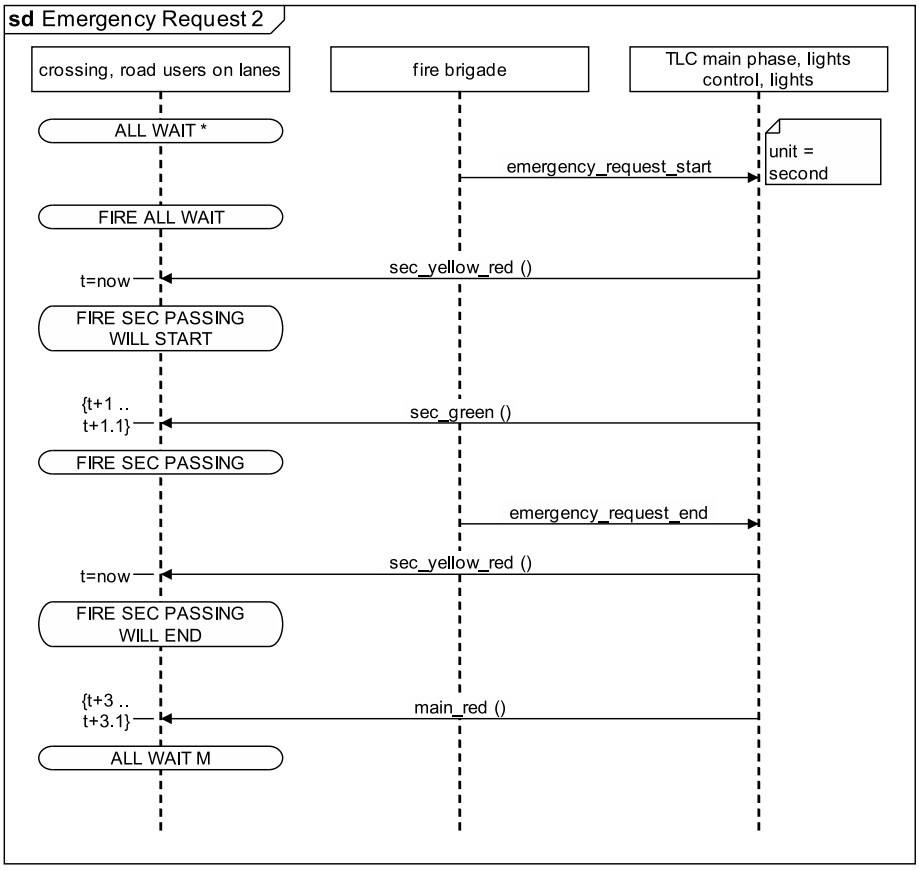


Figure 6.11: Sequence Diagram for EmergencyRequestSecondaryRoadPassing 1

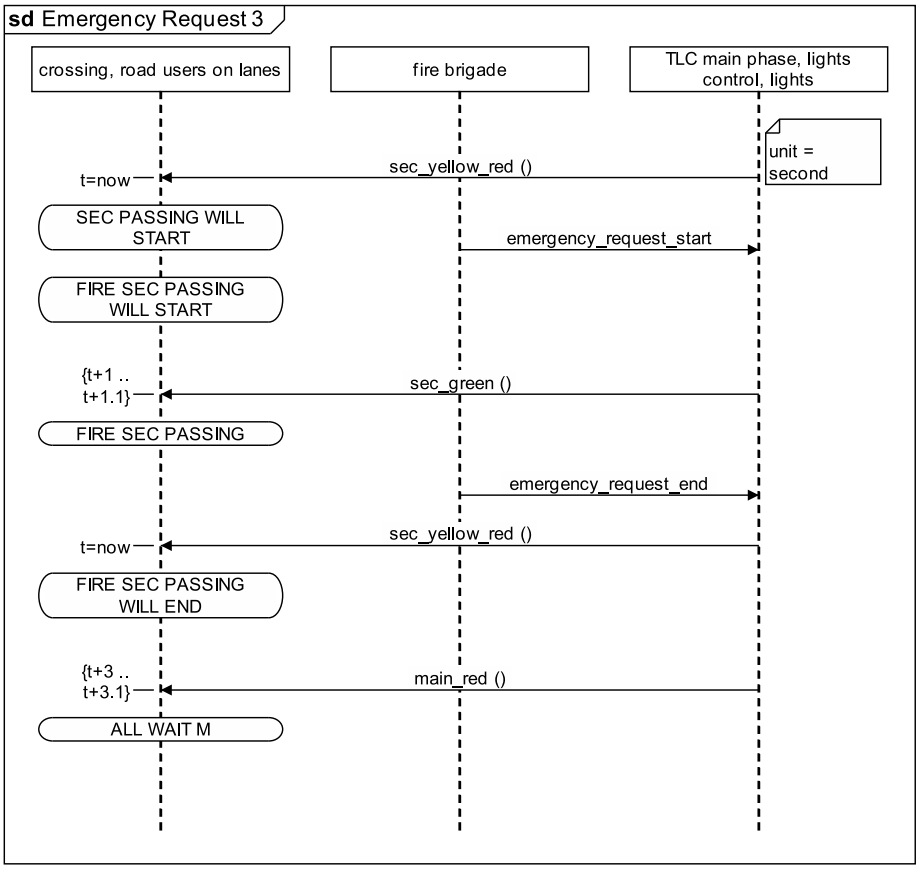


Figure 6.12: Sequence Diagram for EmergencyRequestSecondaryRoadPassing 3

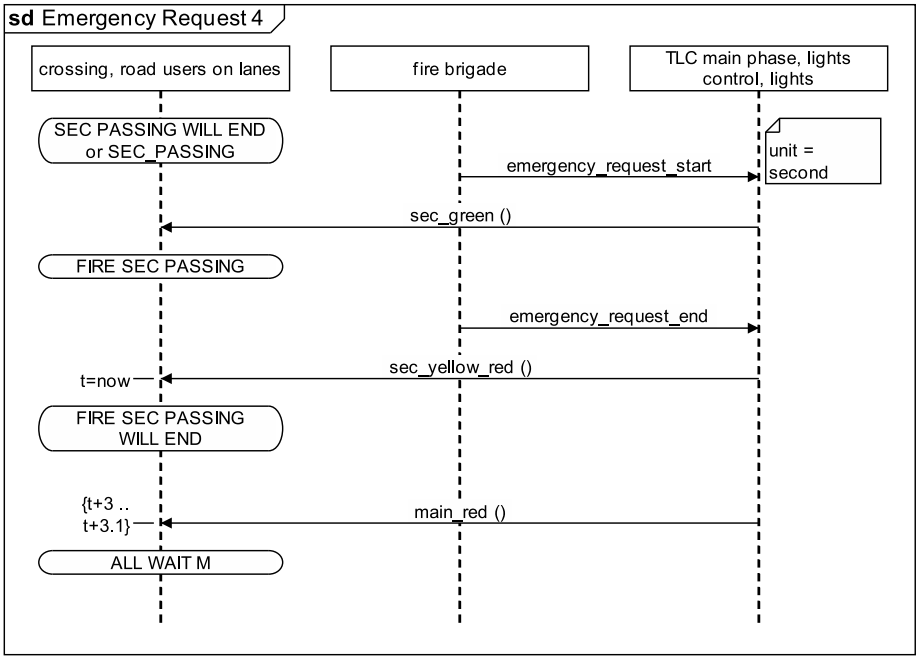


Figure 6.13: Sequence Diagram for EmergencyRequestSecondaryRoadPassing 4

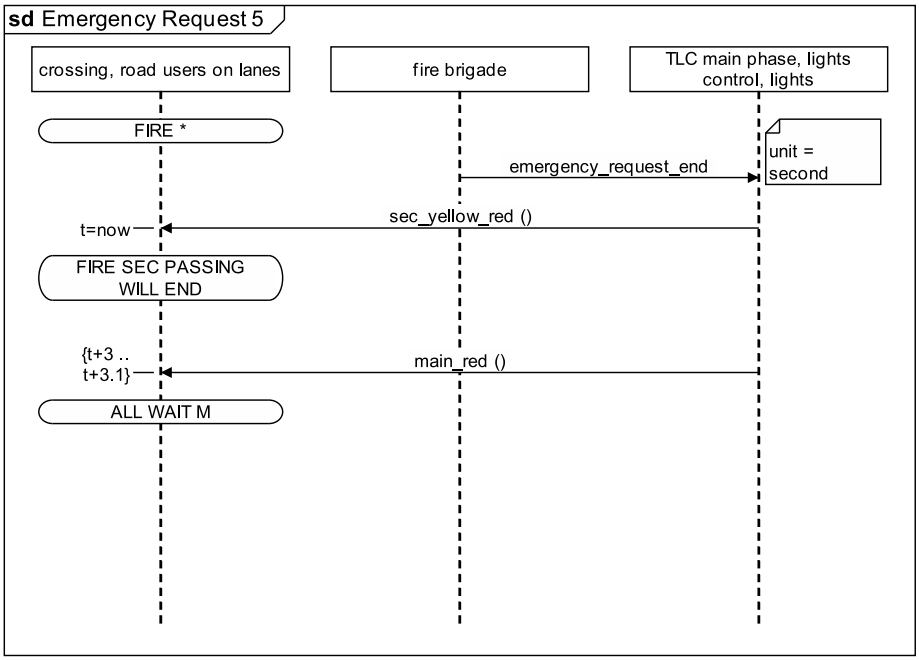


Figure 6.14: Sequence Diagram for EmergencyRequestSecondaryRoadPassing 5

6.4.4 Subproblem: BrokenLightSafeState

Fig. 6.15 shows the sequence diagram for the subproblem *BrokenLightSafeState*. 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 and a controlled variable. A diagram using the more technical phenomenon *current* is hard to understand. Including the domain lights control enforce to include also the abstract phenomenon *on/off*. This phenomenon and also the phenomenon *broken_light* are refined in Step 8 because the domain lights control must be described there.

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.

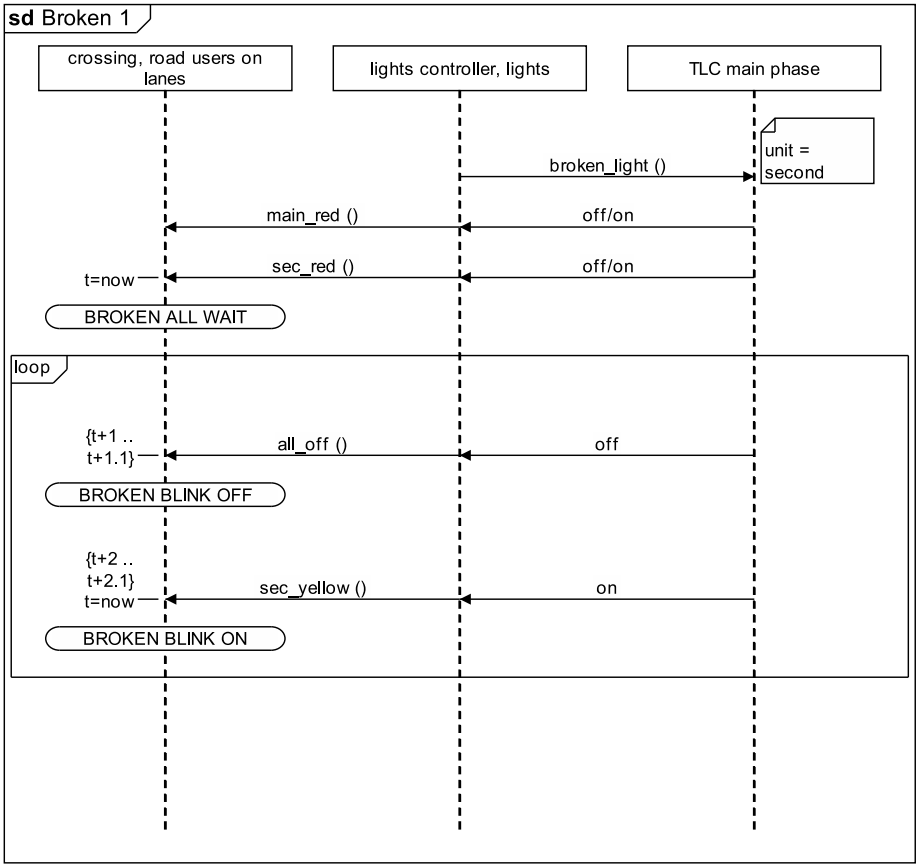


Figure 6.15: Sequence Diagram for BrokenLightSafeState 1

6.4.5 Initialization

Additionally, the initialization must be specified as shown in Fig. 6.16.

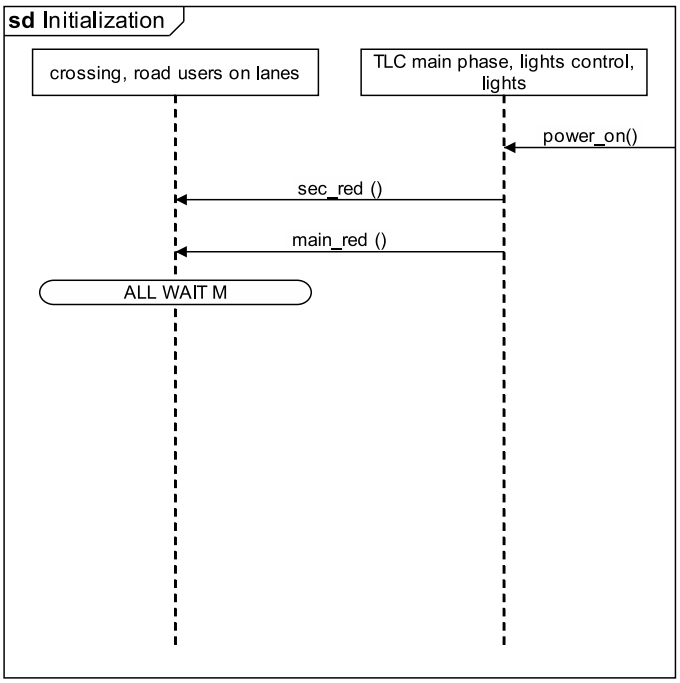


Figure 6.16: Sequence Diagram for Initialization 1

6.4.6 Domain Knowledge

The domain knowledge is expressed using the sequence diagrams in Fig. 6.17. Using this domain knowledge, the machine domain and the domain lights control can be separated from the domain lights.

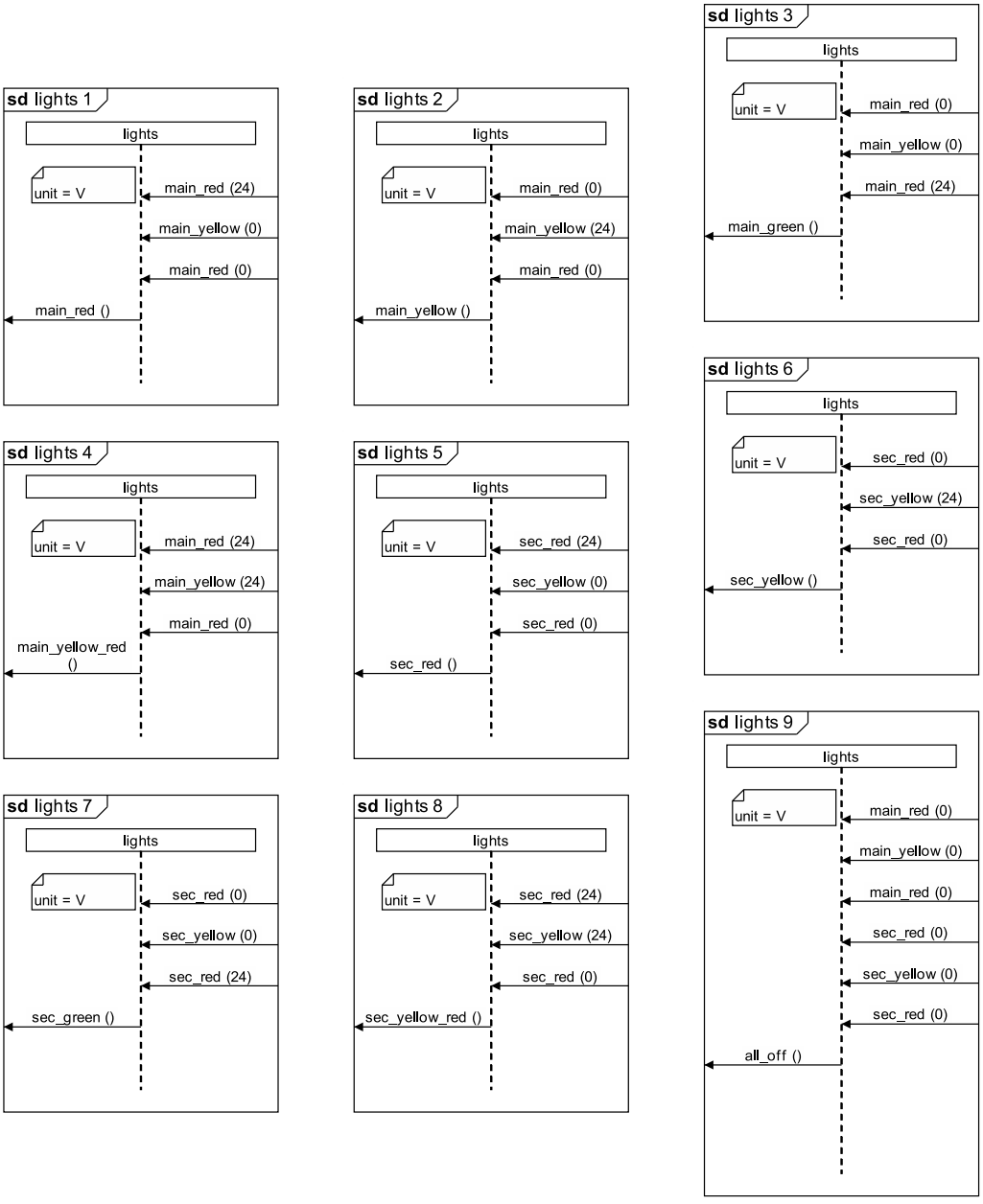


Figure 6.17: Sequence Diagrams for the Lights Domain

6.4.7 Validation

To validate this step, it was assured that all requirements are captured. They are assigned to the subproblems as described in Step 3 and therefore also assigned to the corresponding sequence diagrams. 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 relationships of Step 3 are consistent with the state invariants. The specification can be easily derived from the requirements and the domain knowledge expressed as sequence diagrams.

6.5 Step 5: Design Global System Architecture

In this section the system architecture with its interfaces is developed and the subproblems are associated to the components.

6.5.1 System Architecture

The system architecture shown in Figure 6.18 consists of a software component TrafficLightsControl, which decides on the signaling shown by the physical traffic lights, and two hardware components LightsControl (which connects the software to the physical lights) and InductionLoopControl (which connects the software to the induction loop).

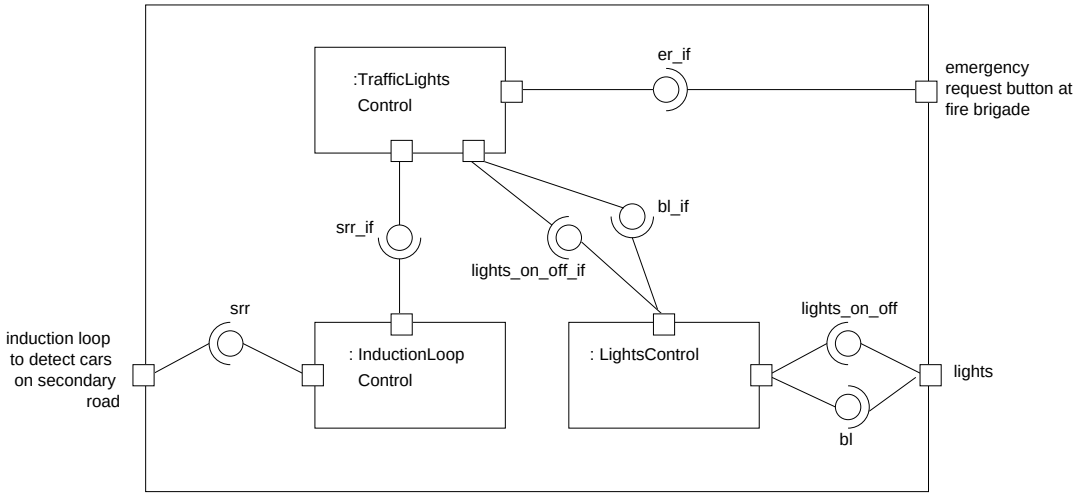


Figure 6.18: System Architecture for Traffic Lights System

6.5.2 Subcomponents

No subcomponents are necessary for this problem.

6.5.3 External and Internal System Architecture Interfaces

The interfaces between the components are described by interface classes that contain the signals that can be exchanged via the interfaces. In the example, we have to refine the abstract signal *main_yellow*, *main_red* etc. used in Step 5 to concrete signals needed to control the physical traffic light elements. For example, the abstract signal *main_red* is refined to the sequence of signals *main_red(24)*, *main_yellow(0)*, and *main_green(0)*. This means that each light bulb is controlled separately, and switching on a light bulb means a volt value of 24V, whereas switching off a light bulb corresponds to a volt value of 0V.

The signals used by the software component traffic light control are more abstract, they have a Boolean parameter for each light that indicates if it must be switched on or off. Figure 6.19 shows the interface classes lights_on_off and lights_on_off_if that contain the signals described above.

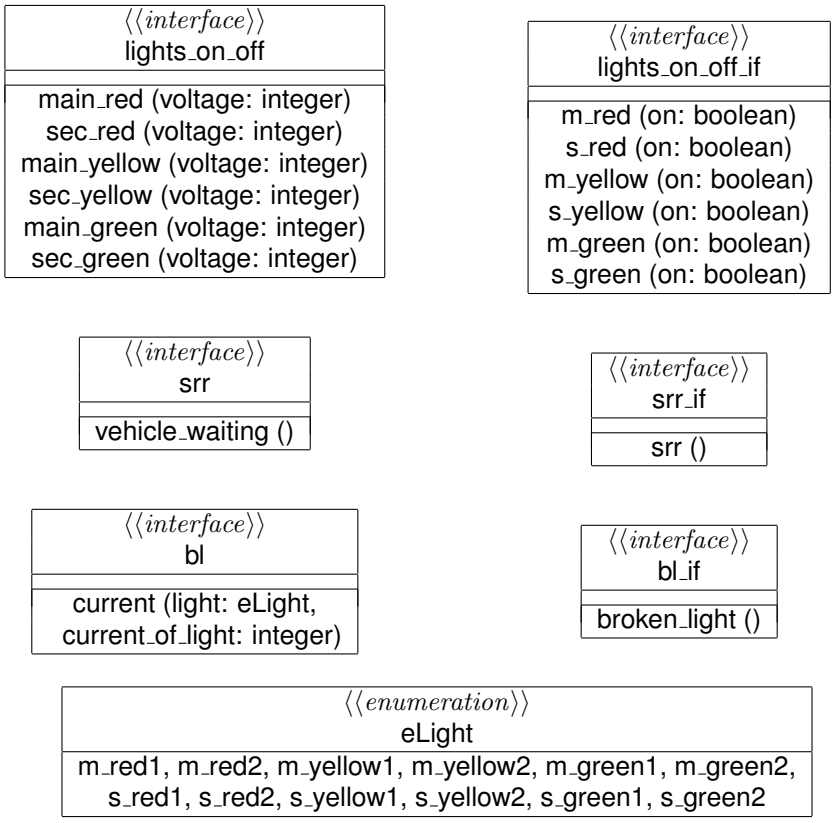


Figure 6.19: Interface classes for the traffic light system

The signal of this 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 TrafficLightControl as a short 5 V signal.

6.5.4 Subproblem Relationships

All subproblems should be implemented in the component TrafficLightsControl. The component TrafficLightsControl is therefore the same as in Step 3. The other components perform simple transformations.

6.5.5 Validation

The external interfaces of the components cover the interfaces of all problem diagrams. Additionally, the architecture cover all specifications developed in Step 4. The machine domain in the context diagram and the domains in the problem diagrams being part of the machine are separate components. The description of the relationships between the subproblems for machine and the components are equal and therefore consistent.

6.6 Step 6: Derive Specifications for all Components being Relevant for the Subproblem

In this step, the interface behavior of all complex components is specified.

This specification can be expressed using the sequence diagram of Step 6 and split the machine domain into the components specified in Step 5. An example is shown in Fig. 6.20. It is derived from Fig. 6.8.

In this step, it is also allowed to merge domains and express the behavior between these components separately. Since the diagrams become very complex in this case and only little additional information is given, the specification of Step 4 can be used and the behavior of the components LightsControl and InductionLoopControl can be specified. Fig. 6.21 shows the specification of the component LightsControl. The component converts the digital signals *(on/off)* into an analog voltage to control the lights. In Fig. 6.22 is shown how a broken light is detected: When a light bulb is supplied with 12 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. Fig. 6.23 shows one possible sequence of interactions when a broken light is detected.

Fig. 6.24 shows the sequence diagram for the component InductionLoopControl. The secondary road request (*ssr()*) is transformed into the signal *vehicle_waiting*. Since the abstract signal *srr* is used, an additional technical description is necessary.

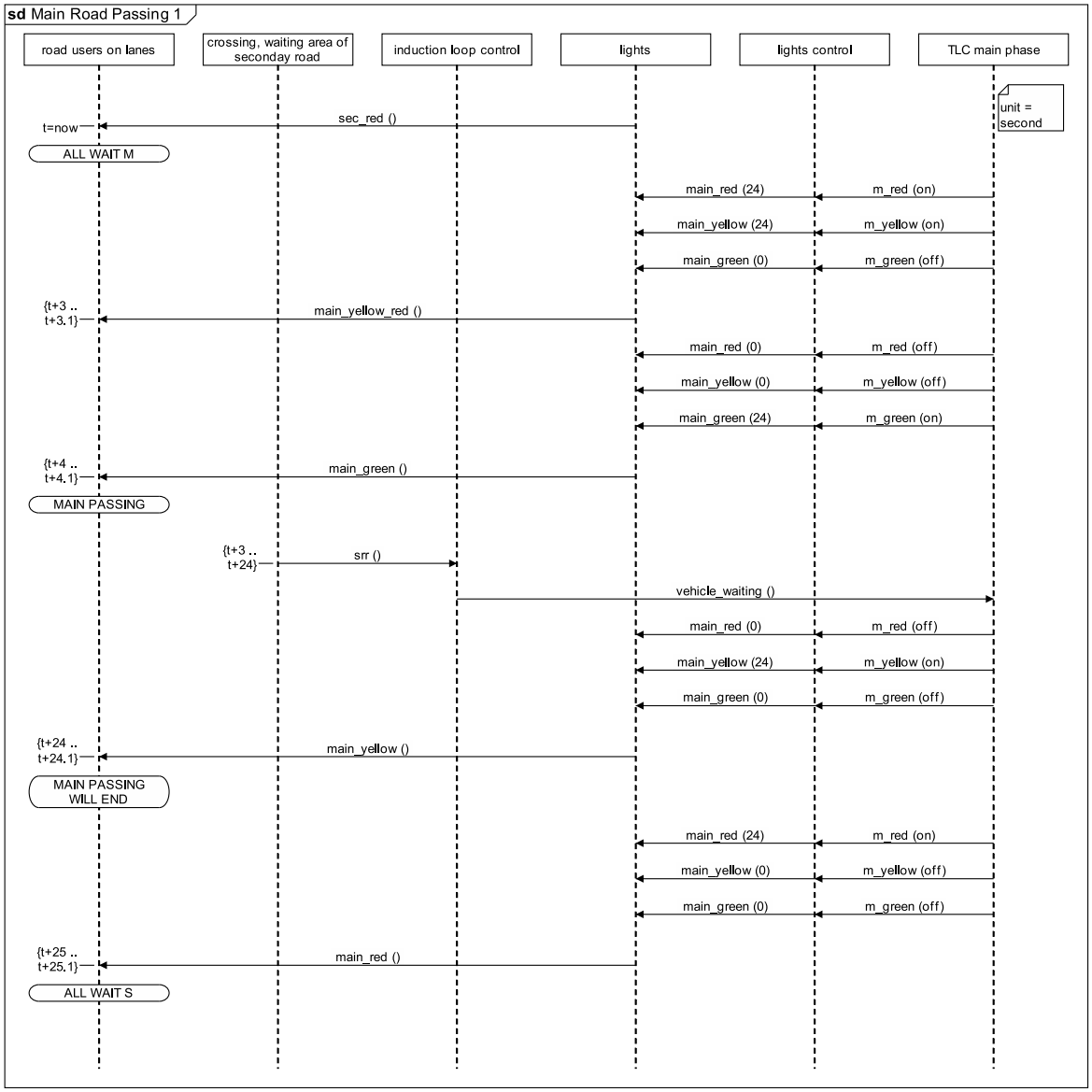


Figure 6.20: Interface Behavior for Subproblem MainRoadPassing 1

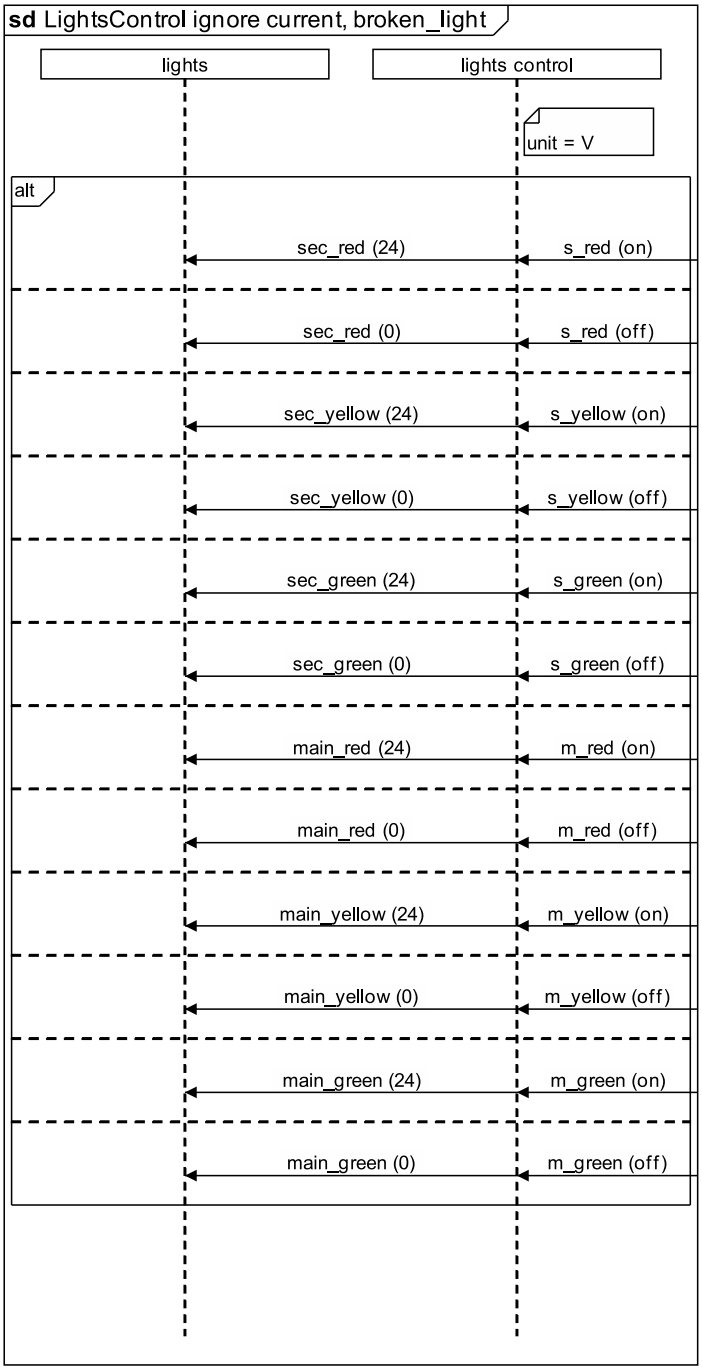


Figure 6.21: Interface Behavior 1 of the Component LightsControl for all Subproblems

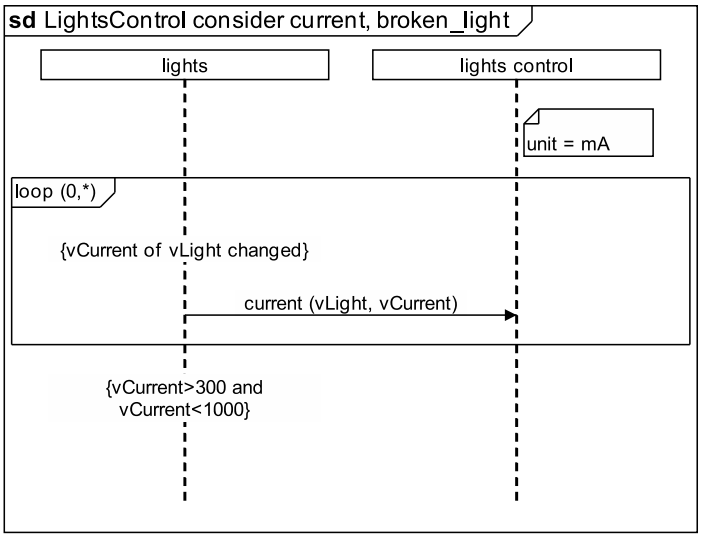


Figure 6.22: Interface Behavior 2 of the Component LightsControl for all Subproblems

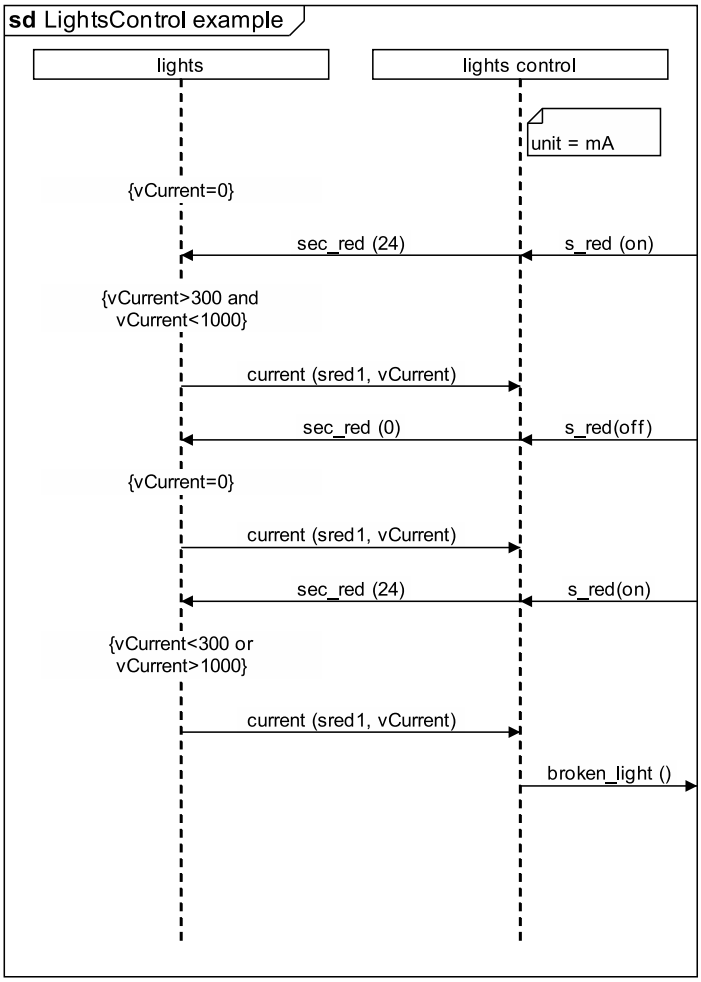


Figure 6.23: Interface Behavior of the Component LightsControl for all Subproblems, Sample Trace

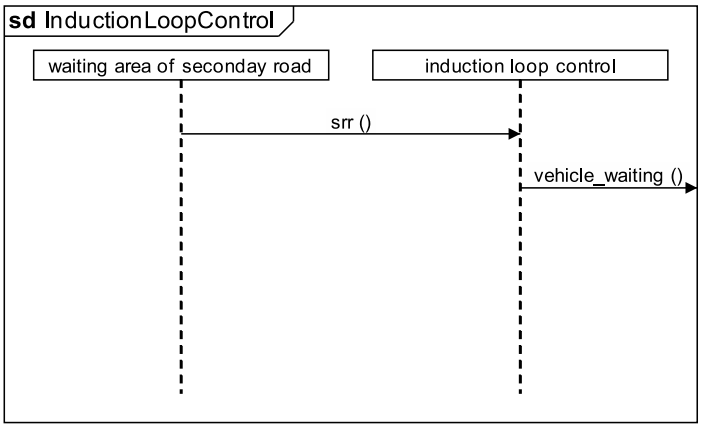


Figure 6.24: Interface Behavior of the Component InductionLoopControl for all Subproblems

6.7 Step 7: Design Software Architecture for each Software Component and each Subproblem

In this step, the architectural patterns of Step 7 in Chapter 5 are instantiated, and the internal interfaces are specified.

6.7.1 Subproblem: SecondaryRoadPassing

Fig. 6.25 shows the software architecture for the subproblem *SecondaryRoadPassing*. It is an instantiation of the required behavior architectural pattern. In this architecture, there is no sensor included. For this reason, the components Sensor IAL and Sensor HAL are removed from the software architecture.

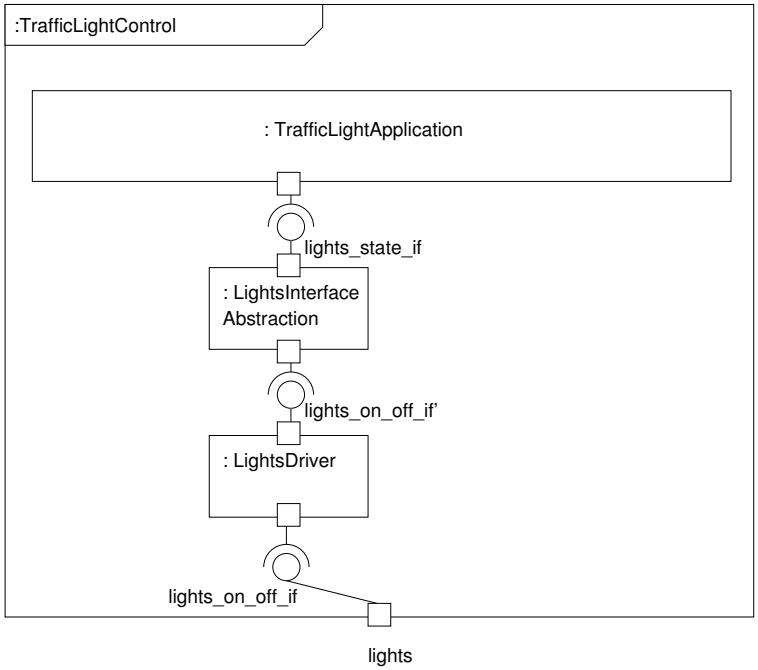


Figure 6.25: Software Architecture for SecondaryRoadPassing

Fig. 6.26 shows the interfaces of the software architecture for the subproblem *SecondaryRoadPassing*.

6.7.2 Subproblem: MainRoadPassing

Fig. 6.27 shows the software architecture for the subproblem *MainRoadPassing*.

Fig. 6.28 shows the interfaces of the software architecture for the subproblem *MainRoadPassing*.

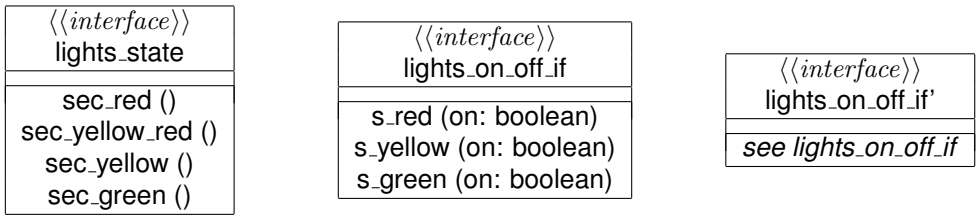


Figure 6.26: Interface Classes for SecondaryRoadPassing

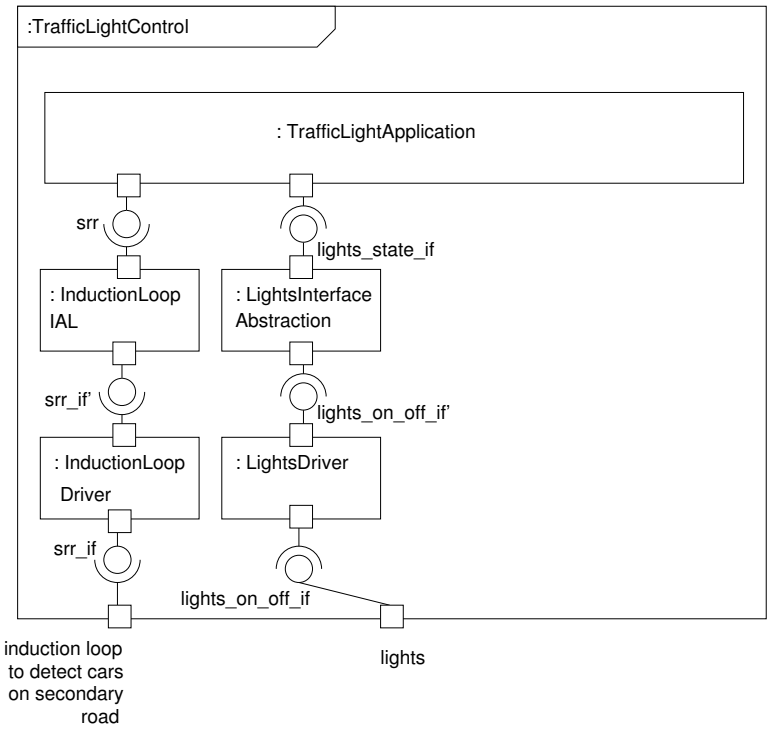


Figure 6.27: Software Architecture for MainRoadPassing

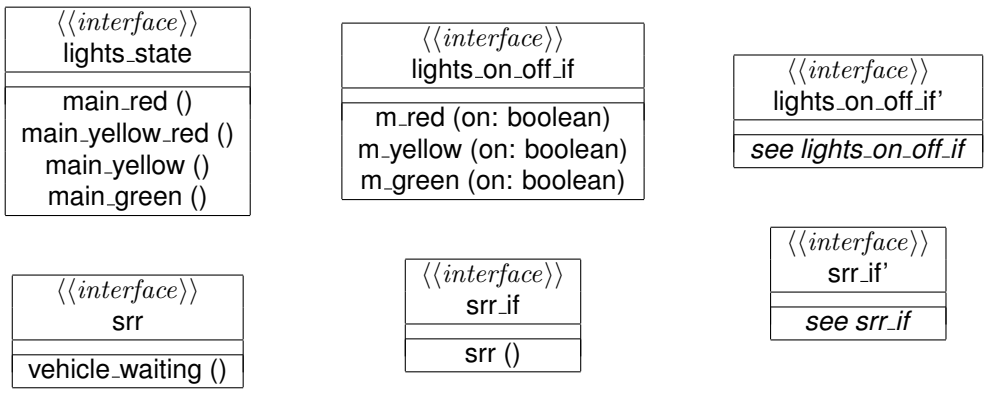


Figure 6.28: Interface Classes for MainRoadPassing

6.7.3 Subproblem: EmergencyRequestSecondaryRoadPassing

Fig. 6.29 shows the software architecture for the subproblem *EmergencyRequestSecondaryRoadPassing*. The interfaces abstraction layer is removed for the emergency request because the interface of the application component is the same as for the component EmergencyRequestDriver.

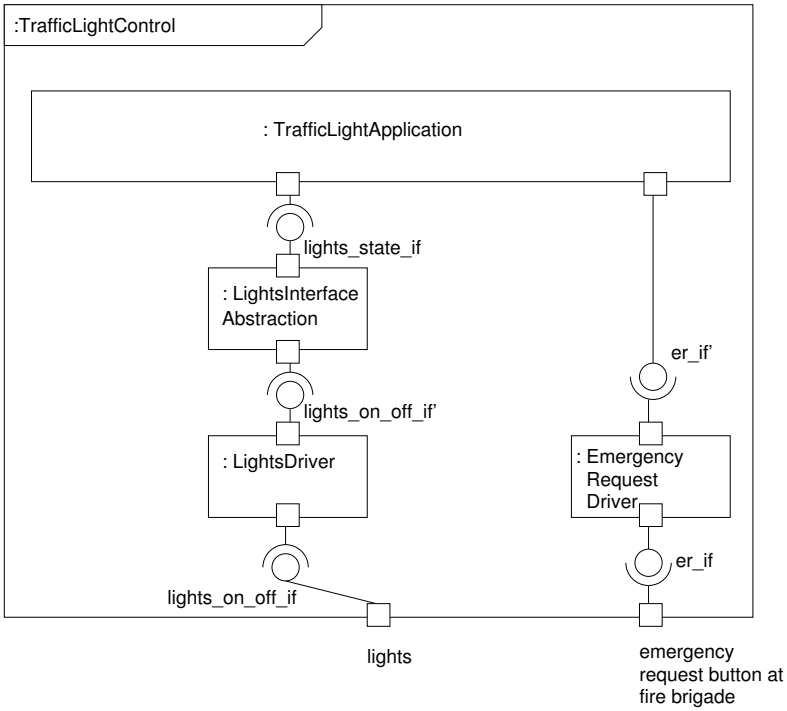


Figure 6.29: Software Architecture for EmergencyRequestSecondaryRoadPassing

Fig. 6.30 shows the interfaces of the software architecture for the subproblem *EmergencyRequestSecondaryRoadPassing*.

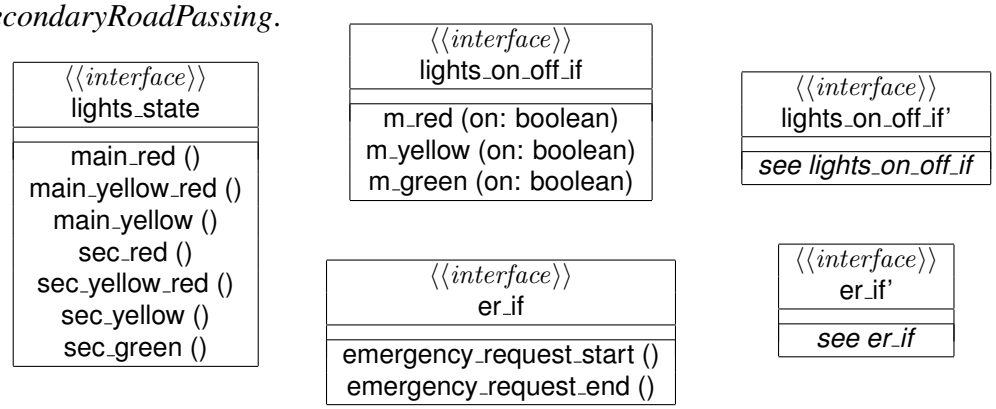


Figure 6.30: Interface Classes for EmergencyRequestSecondaryRoadPassing

6.7.4 Subproblem: BrokenLightSafeState

Fig. 6.31 shows the software architecture for the subproblem *BrokenLightSafeState*. The interfaces abstraction layer is removed for the broken light since the interface of the application component is the same as for the component BrokenLightDriver.

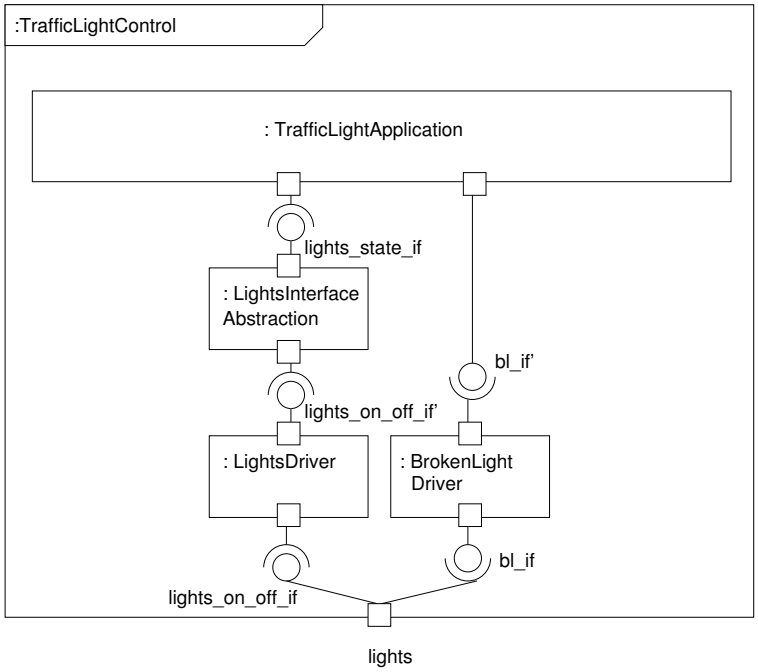


Figure 6.31: Software Architecture for BrokenLightSafeState

Fig. 6.32 shows the interfaces of the software architecture for the subproblem *BrokenLightSafeState*.

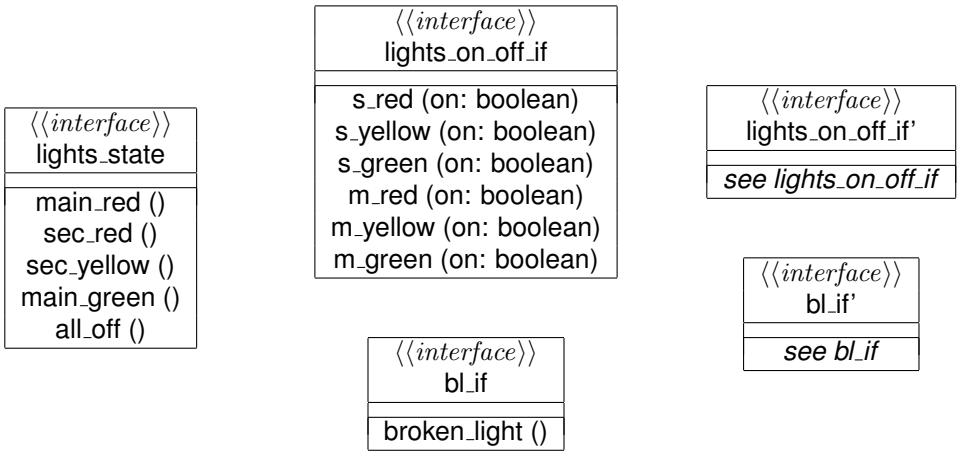


Figure 6.32: Interface Classes for BrokenLightSafeState

6.7.5 Subcomponents

The component TrafficLightApplication has to be refined into a clock that generates cyclic signals, a TimeOutTimer component that generates timeouts and a component TrafficLightBehavior. The separation is the same for all subproblems and shown in Fig. 6.33.

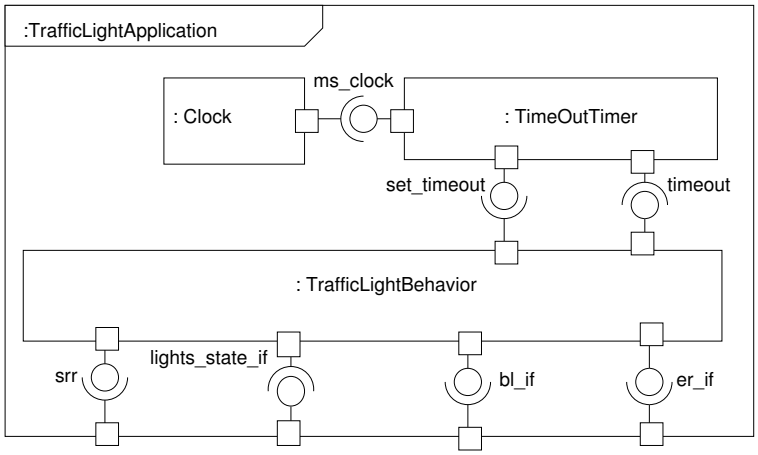


Figure 6.33: Subcomponents of the Component TrafficLightApplication

Fig. 6.34 shows the interfaces in the component TrafficLightApplication.

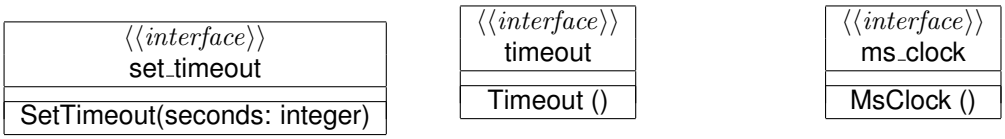


Figure 6.34: Interface Classes in the Component TrafficLightApplication

6.7.6 Validation

Since all architectural diagrams (except of the refinement of the component TrafficLightApplication) are instantiations of the given patterns, no validation of the architecture is necessary. Additionally, it is evaluated that all interfaces are covered. The operations in the external interface classes of the software architecture are the same as the signals in the sequence diagrams of Step 6. The interface of the application components contain the signals used in the sequence diagrams of Step 4. The directions of all signals are consistent to each other and to the system architecture.

6.8 Step 8: Specify Behavior of Software Architecture Components for each Subproblem

In this step, the interface behavior of all components is specified using sequence diagrams. The HAL is the software interface to the hardware. The behavior is described in the data book and therefore not specified here. Therefore, the components TrafficLightApplication, InductionLoopIAL, and LightsInterfaceAbstraction have to be specified.

6.8.1 Component: TrafficLightApplication

The component TrafficLightApplication is included in all subproblems. The specification for all subproblems is the same as the specification of Step 4.

6.8.2 Component: InductionLoopIAL

The component InductionLoopIAL is only included in the subproblem *MainRoadPassing*. The specification shown in Fig. 6.35 is derived from the domain knowledge shown in Fig. 6.24.

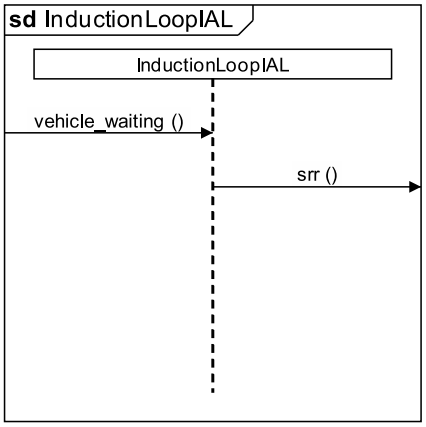


Figure 6.35: Software Architecture for BrokenLightSafeState

The component InductionLoopIAL transforms the signal *vehicle_waiting()* into the signal *srr()* for the application component.

6.8.3 Component: LightsInterfaceAbstraction

The component TrafficLightApplication is included in all subproblems. The specification for this component can be derived from Fig. 6.17 in the same way as for the component InductionLoopIAL.

6.8.4 Validation

The sequence diagrams together describe the same behavior as in Step 6. All signals in the interface classes of Step 7 are captured in at least one sequence diagram and the direction of the signals is consistent with the required or provided interfaces of Step 7. The signals connect the components as connected in the software architecture. In the interfaces abstraction layer no state invariants are relevant. The sequence diagrams and therefore also the state invariants in this step are the same as in Step 6.

6.9 Step 9: Specify Software Components of Software Architectures for each Subproblem

In this step, for all components a component overview description is developed. The used data types with their operations are specified. Additionally, the state machine and invariants are developed.

6.9.1 Component: TrafficLightApplication

The component TrafficLightApplication is split into the subcomponents TrafficLightApplication, Clock, and TimeOutTimer. These components are specified separately in Sections 6.9.2, 6.9.4, and 6.9.3. It is an active component since a clock is included.

6.9.2 Component: TrafficLightBehavior

The data and the interfaces of the components TrafficLightBehavior are specified in Fig. 6.36. It is a passive component since it reacts immediately to input signals. The component requires and provides the same interfaces as specified in Step 7. This component must store if a vehicle is waiting or not.

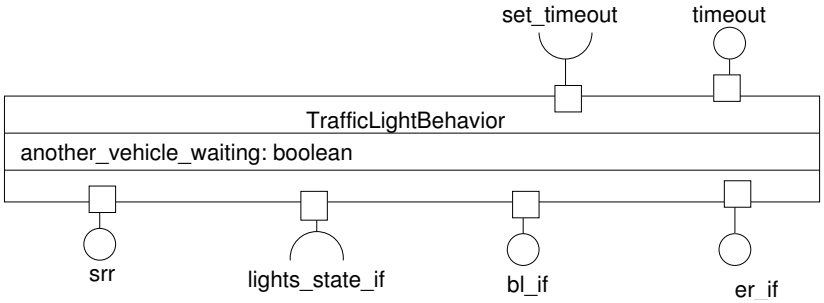


Figure 6.36: Component Overview Description of TrafficLightBehavior

Subproblem: SecondaryRoadPassing

The state machine of the component TrafficLightBehavior and the subproblem *SecondaryRoad-Passing* is shown in Fig. 6.37. It is derived from the sequence diagram in Fig. 6.7.

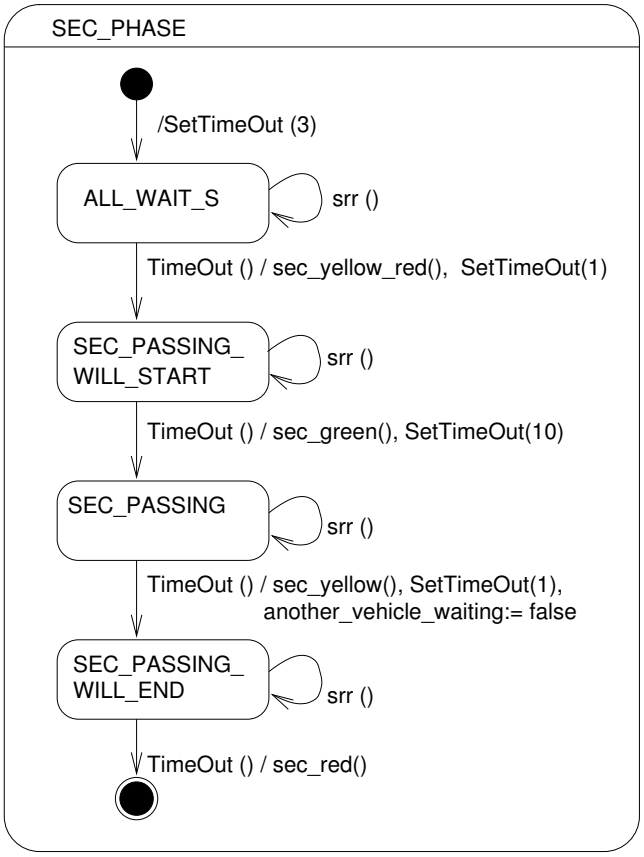


Figure 6.37: State Machine of TrafficLightBehavior, SecondaryRoadPassing

Subproblem: MainRoadPassing

The state machine of the component TrafficLightBehavior and the subproblem *MainRoadPassing* is shown in Fig. 6.38. It summarizes the sequence diagrams in Figures 6.8 and 6.9.

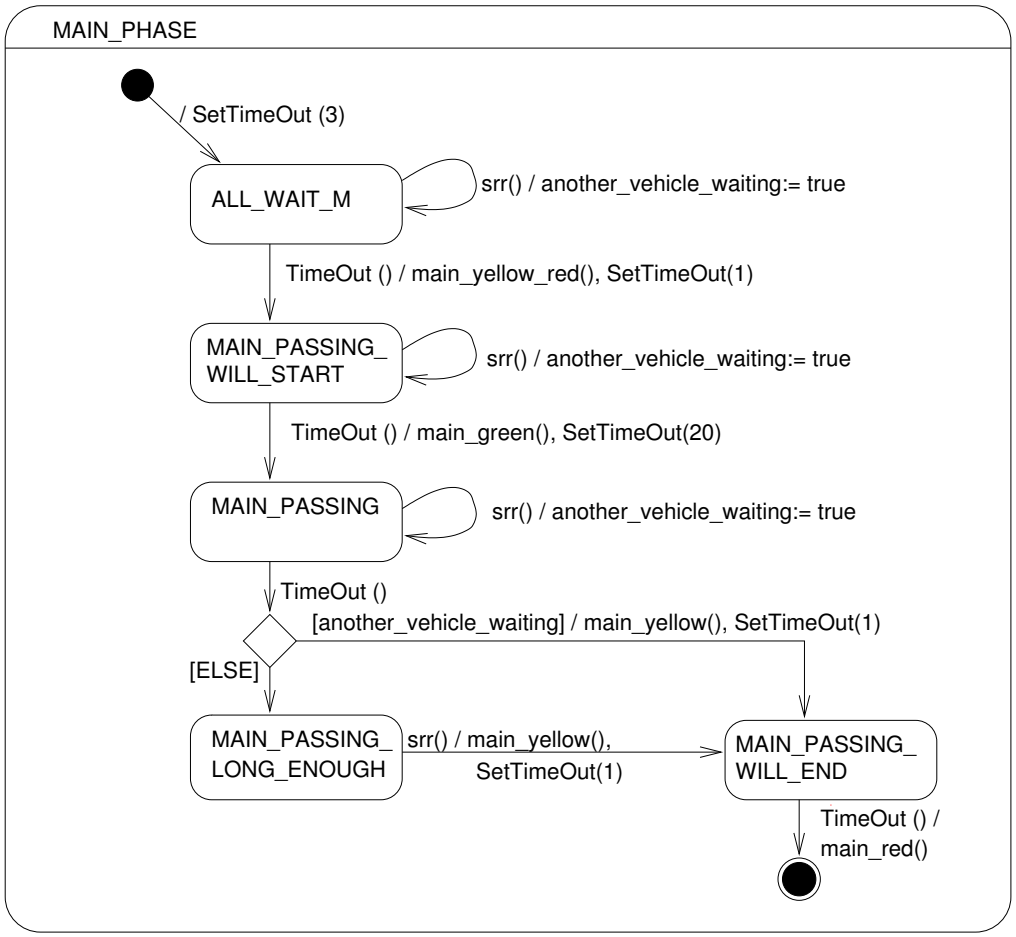


Figure 6.38: State Machine of TrafficLightBehavior, MainRoadPassing

Subproblem: EmergencyRequestSecondaryRoadPassing

The state machine of the component TrafficLightBehavior and the subproblem *EmergencyRequestSecondaryRoadPassing* is shown in Fig. 6.39. It summarizes the sequence diagrams in Figures 6.10, 6.11, 6.12, 6.13, and 6.14.

Subproblem: BrokenLightSafeState

The state machine of the component TrafficLightBehavior and the subproblem *BrokenLightSafeState* is shown in Fig. 6.40. It is derived from the sequence diagram in Fig. 6.15.

6.9.3 Component: TimeOutTimer

In all subproblems, timing requirements have to be fulfilled. Therefore, a component TimeOutTimer is necessary for all subproblems. The class diagram in Fig. 6.41 shows the same required and provided interfaces as Fig. 6.33. Additionally, it includes the data type and operators for the remaining time. The state machine in Fig. 6.42 is using this data. The component is a passive component since it reacts immediately to the input signals of the clock.

The operations can be specified using the following pre- and postconditions:

IsZero() pre true
post Result = true $\Leftrightarrow$ remaining_time = 0

SetTime(x) pre true
post remaining_time = x

DecTime() pre remaining_time $\neq 0$
post remaining_time $=$ remaining_time@pre -1

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

In state stopped $\Rightarrow$ remaining_time = 0

6.9.4 Component: Clock

The component Clock shown in Fig. 6.43 is an active component since it has to work in parallel to all other components to generate cyclic signals. Usually it is a standard component, included in the operating system. Hence, it is not specified here.

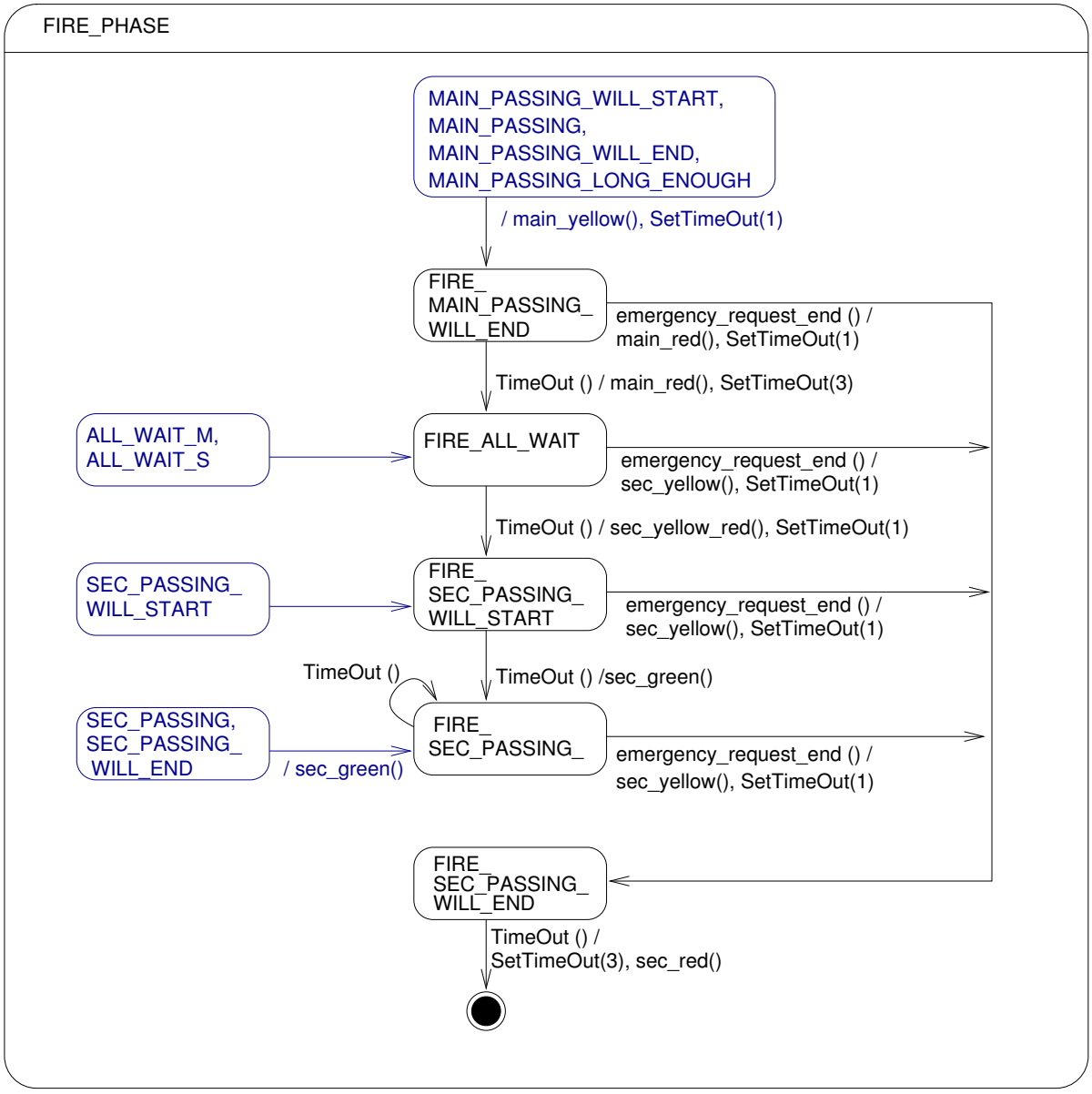


Figure 6.39: State Machine of TrafficLightBehavior, EmergencyRequestSecondaryRoadPassing

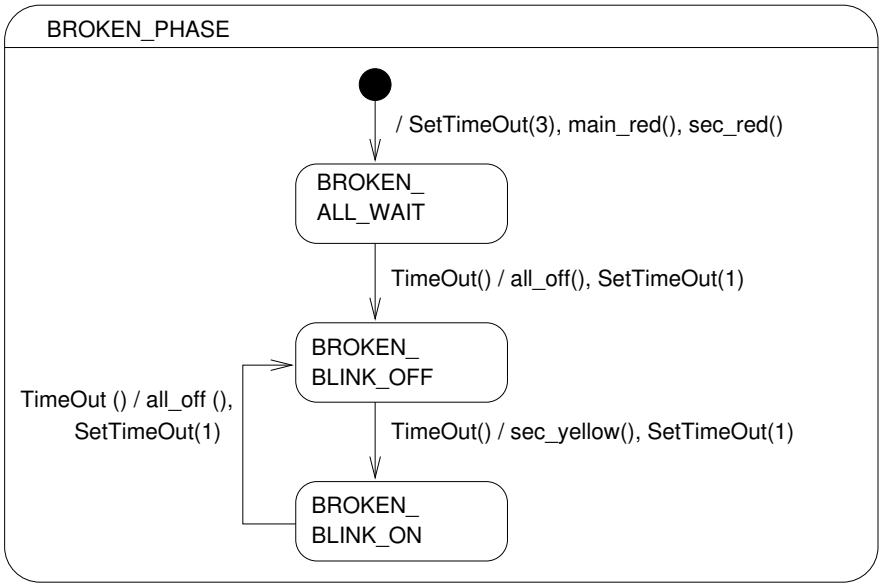


Figure 6.40: State Machine of TrafficLightBehavior, BrokenLightSafeState

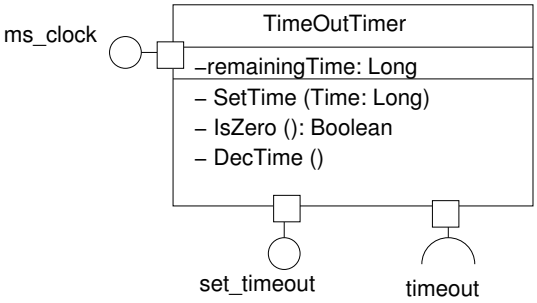


Figure 6.41: Component Overview Description of TimeOutTimer

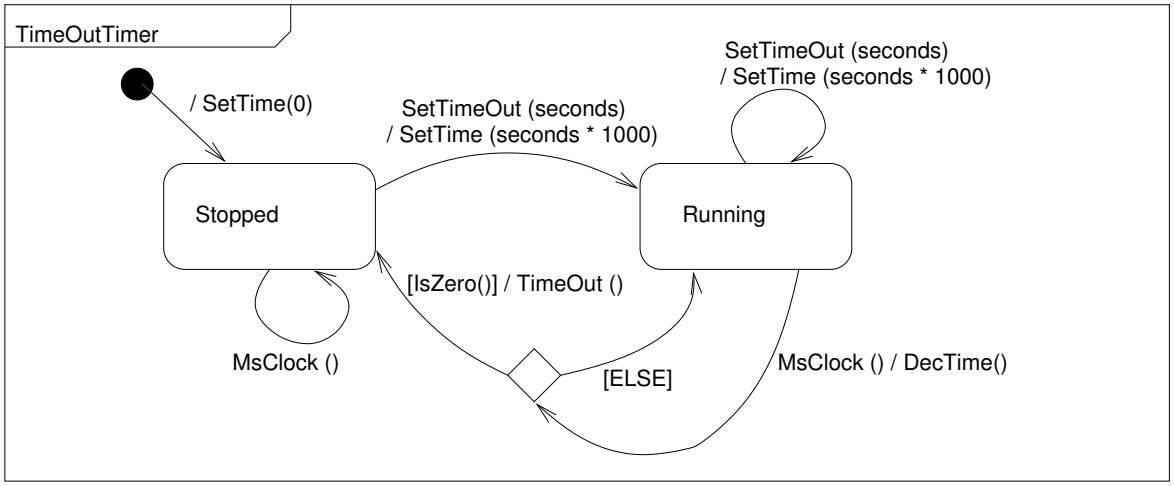


Figure 6.42: State Machine of TimeOutTimer

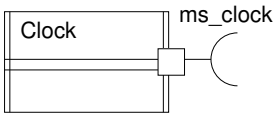


Figure 6.43: Component Overview Description of Clock

6.9.5 Component: InductionLoopIAL

The data and the interfaces of the components InductionLoopIAL are specified in Fig. 6.44. It is a passive component since it directly hands over the input signals. The component requires and provides the same interfaces as specified in Step 7. This component is only included in the subproblems *MainRoadPassing*.

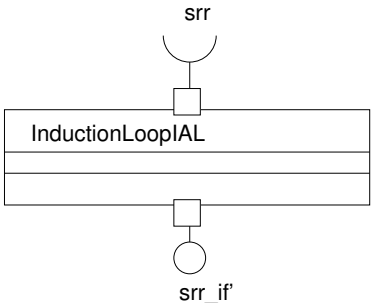


Figure 6.44: Component Overview Description of InductionLoopIAL

Subproblem: MainRoadPassing

The state machine of the component InductionLoopIAL and the subproblem MainRoadPassing is shown in Fig. 6.45.

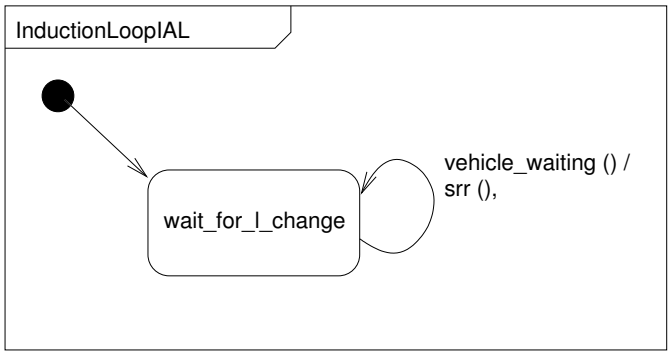


Figure 6.45: State Machine of InductionLoopIAL, MainRoadPassing

6.9.6 Component: LightsInterfaceAbstraction

The class diagram in Fig. 6.46 shows the same required and provided interfaces as Fig. 6.33. Since the state machine for this state contains only one state, the behavior is the same for all subproblems and only one state machine is developed. This component is a passive component since it reacts immediately to the input signals. The state machine is shown in Fig. 6.47.

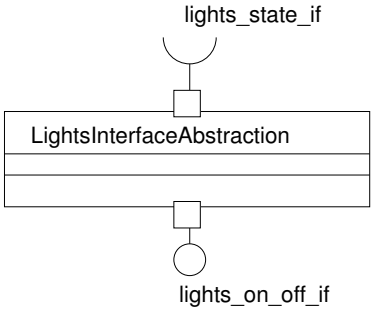


Figure 6.46: Component Overview Description of LightsInterfaceAbstraction

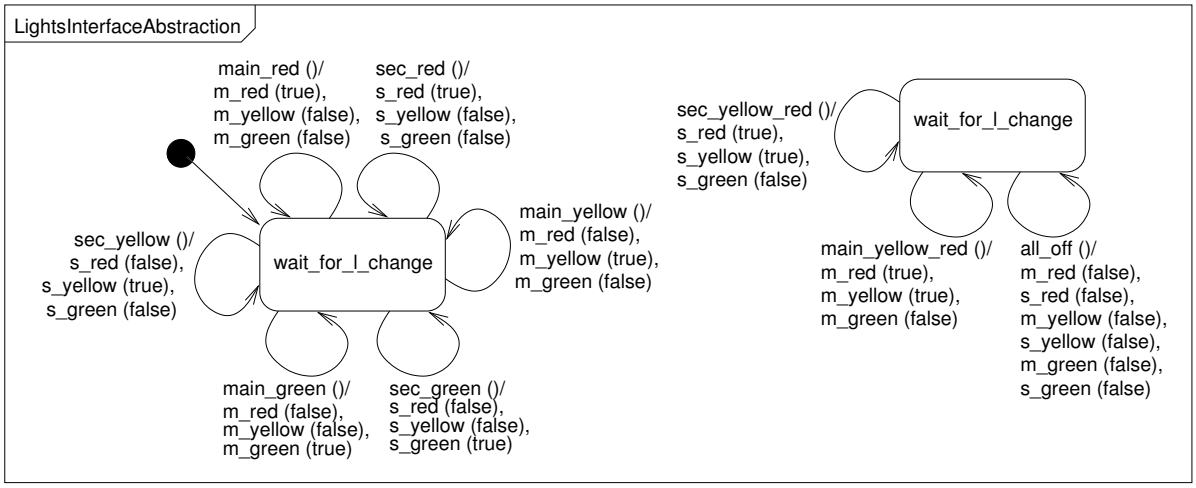


Figure 6.47: State Machine of LightsInterfaceAbstraction, All Subproblems

6.9.7 Validation

Each architectural component is covered, and in all state machines each possible input signal (as specified in Step 7) is taken into account.

The state machine behaves as described in the sequence diagrams of Step 8. All states are covered. Additional states ending with _PASSING_WILL_START are introduced. All state machines of the software architecture together are consistent with the sequence diagrams of Step 6. There is no direct user interaction. Hence, no error cases for user interaction must be considered. In all states, all signals that can occur are covered.

Moreover, all referenced interface classes are the same as the interface classes of the subproblem architecture of the respective component (Step 7).

6.10 Step 10: Develop Global Software Architectures

The composed architecture is shown in Figure 6.48. The subproblems *SecondaryRoadPassing* and *MainRoadPassing* are related sequentially (cf. case 2 in Step 10 of Chapter 5). Hence, the application components have to be merged. Since the subproblems *EmergencyRequestSecondaryRoadPassing* and *BrokenLightSafeState* are related parallel to the other subproblems and share the same output phenomena (cf. 3), they must also be merged. The component LightInterfaceAbstraction has already been merged in Step 9. All components that are HALs (cf. case 1) are merged with the components of the same name in the other subproblem architectures.

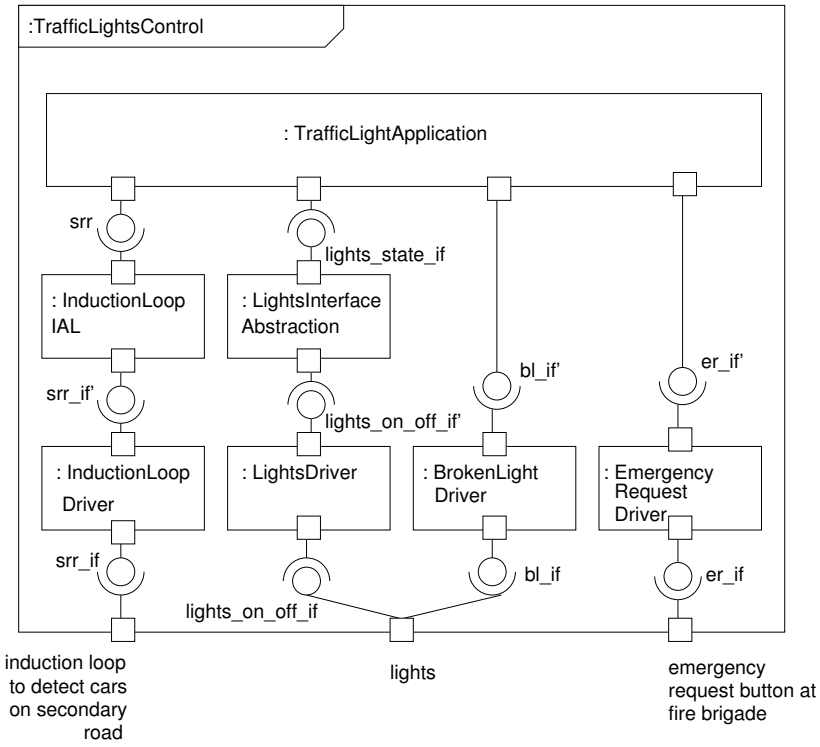


Figure 6.48: Software architecture for traffic light control component

The interface classes are merged as shown in Fig. 6.49.

6.10.1 Validation

The global architecture contains all components and interfaces of all subproblem architectures. Its external interfaces are the same as in the system architecture developed in Step 5.

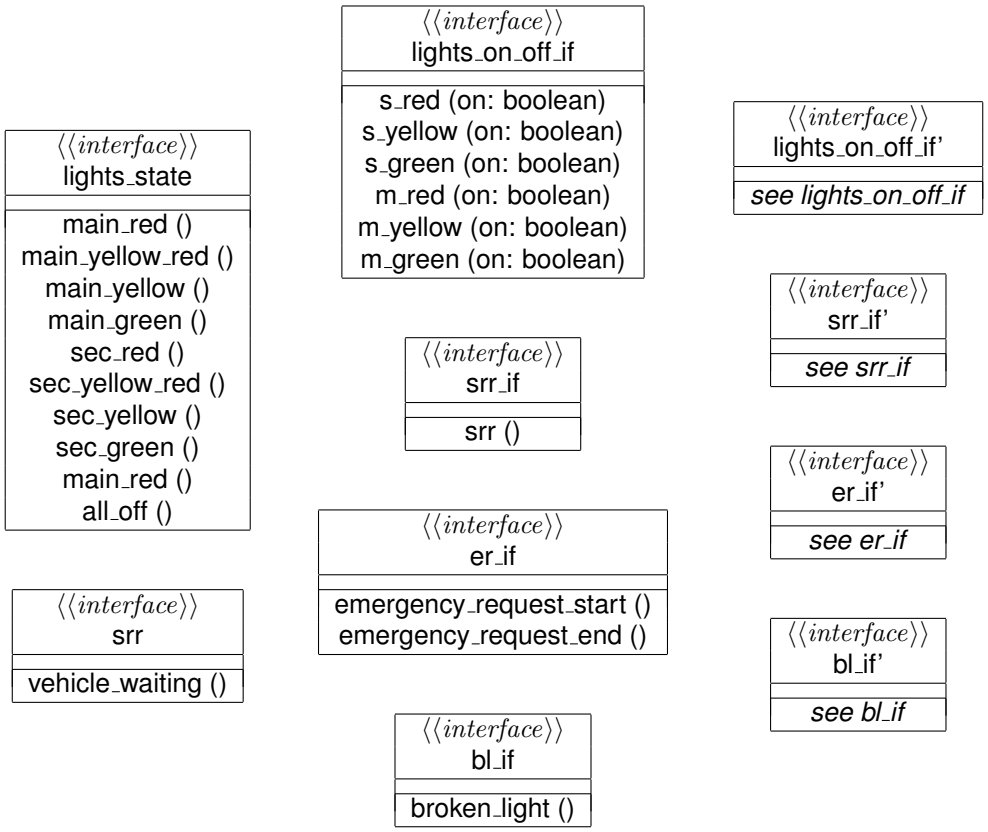


Figure 6.49: Interface classes for the traffic light control

6.11 Step 11: Specify Composed Software Components

In this step, the components of the software architecture for the subproblems are composed.

6.11.1 Component: TrafficLightApplication

The components inside the component TrafficLightApplication are specified separately in the Sections 6.11.2, 6.11.4, and 6.11.3.

6.11.2 Component: TrafficLightBehavior

The data and the interfaces of the component TrafficLightBehavior are already specified in Fig. 6.36. The behavior of the class is described with state machines. The state machines for the subproblems are developed in Step 9 and shown in Figures 6.37, 6.38, 6.39, and 6.40.

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 *EmergencyRequestSecondaryRoadPassing* and *BrokenLight-SafeState* are parallel and activated with the signals *broken_light()* or *emergency_request_start()*. Once activated, they take control over the output signals. Fig. 6.50 shows the composed state machine, consisting of the state machines shown in Figures 6.37, 6.38, 6.39, and 6.40. Additionally, the initialization sequence is considered.

6.11.3 Component: TimeOutTimer

This component is already a composed component, specified in Fig. 6.41 and in Fig. 6.42 of Section 6.9.3.

6.11.4 Component: Clock

This component is part of the operating system. Its overview specification can be found in Fig. 6.43 of Section 6.9.4.

6.11.5 Component: InductionLoopIAL

The component InductionLoopIAL needs no composition since it is only included in one of the subproblem architectures. It is specified in Fig. 6.44 and in Fig. 6.45.

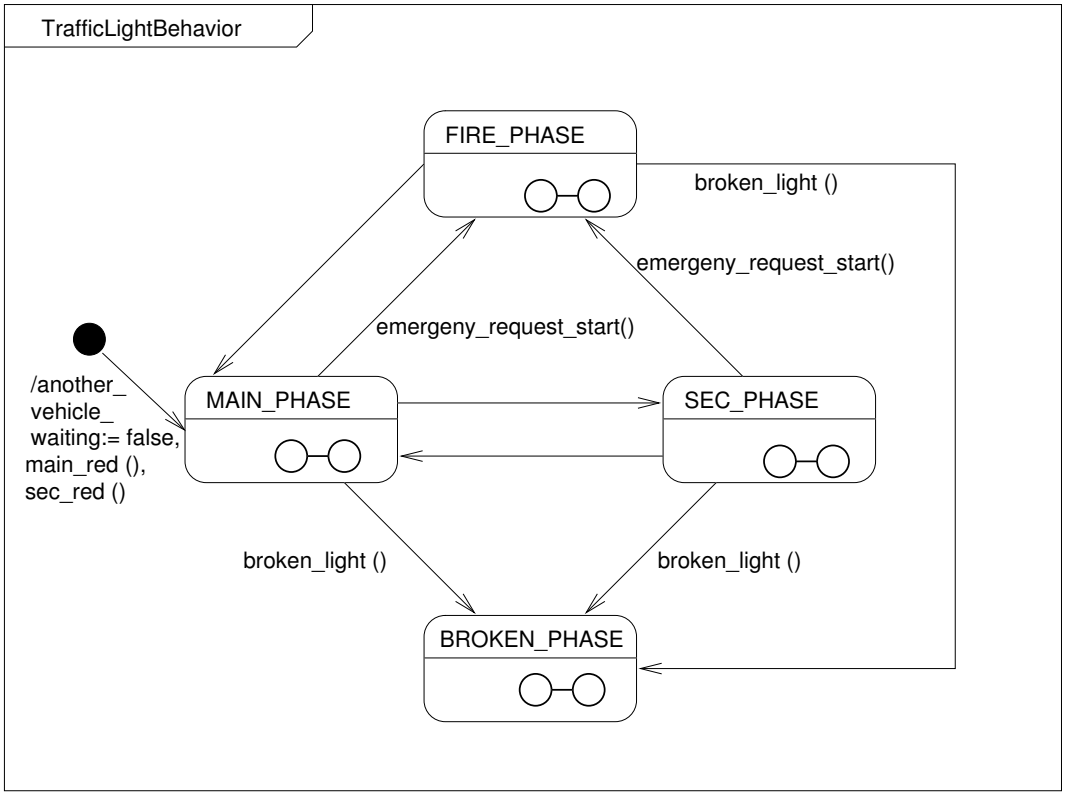


Figure 6.50: Composed State Machine for the Component TrafficLightBehavior

6.11.6 Component: LightsInterfaceAbstraction

This component is already the composed component, specified in Fig. 6.46 and in Fig. 6.47.

6.11.7 Validation

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.

6.12 Step 12: Implement Software Components and Test Environment

In this section, a sample implementation of the components using Java is presented. It makes use of the interface classes of Java to create reusable components.

First, the interface classes are implemented. The interface class lights_state_if, shown in Fig. 6.49, is implemented as follows:

```
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();
}
```

All other interfaces can be implemented using this scheme. To implement the components, the component overview description is implemented. The component overview description of the component TimeOutTimer (cf. Fig. 6.41) can be implemented as follows:

```
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() {}
}
```

The interface classes of the provided interfaces (*ms_clock* and *set_timeout*) must be implemented by this class. All operations specified in these interfaces (*SetTimeOut(int seconds)* and *MsClock()*) must be operations of this class.

Since it is not known which component provides the required interfaces of this component, the required interfaces (here only timeout) become parameters of the constructor (TimeOutTimer). References to the components providing the required interfaces must be stored in private variables (`private timeout to`).

The data (`private long remaining_time = 0`) and the operations on this data (e.g., `private void SetTime(long time)` or `private void DecTime()`) are declared as specified.

Next, the operations are implemented according to the pre- and postconditions. The preconditions are checked with assertions[9] and the postconditions are implemented as follows:

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

```
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-- ;
    }
}
```

[9] Only since Java 1.5

The state machines are implemented inside the public operations specified in the interface classes of the provided interfaces. Additionally, there must be a private attribute for the state (`private int state`) and this attribute must be initialized in the constructor (`state = STOPPED`)[10]. The following code fragment implements the state machine specified in Fig. 6.42.

```
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:
        }
    }

    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:
        }
    }
}
```

Active classes can be implemented using threads as shown in the following code fragment for the component Clock, specified in Fig. 6.43.

[10]Here constants are used; in Java 1.5 enumeration type exists

```
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 );
            }
        }
    }
}
```

To run the test, cases test drivers have to be implemented. These test drivers implement required interfaces of the component that should be tested and stores which operations have been called. Additionally, an operation for the test cases is implemented that checks which operations have been called. The test driver for the interface lights_state_if can be implemented as follows:

```
    class C_lia 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;
        }
    }
```

The test cases can be implemented using the *junit* framework, as shown in the following code fragment. First, the component is initialized. Then the signals are sent to the component and output signals are checked using the test drivers according to the sequence diagrams.

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

    public void testInit() {
        tlb = new TrafficLightBehavior();
        lia = new C_lia();
        tot = new C_tot();

        tlb.connect(tot, lia);
           // sends a signal directly to the provided interfaces
        assertTrue("sec_red not set", lia.checkColor(lia.S_R));
           // checks result using the testdriver
        assertTrue("timoout wrong", tot.checkSetTimeOut(3));
           // checks result using the testdriver
        tlb.Timeout();
           // sends a signal directly to the provided interfaces
        assertTrue("main_red_yellow not set", lia.checkColor(lia.M_RY));
           // checks result using the testdriver
        assertTrue("timeout wrong", tot.checkSetTimeOut(1));
           // checks result using the testdriver
        tlb.Timeout();
           // sends a signal directly to the provided interfaces
        assertTrue("main_green not set", lia.checkColor(lia.M_G));
           // checks result using the testdriver
    }
    ...
}
```

6.12.1 Validation

This step is validated by running the test cases. The output of the test could be as follows:

```
Testsuite: tlc.TrafficLightBehaviorTest
Tests run: 33, Failures: 1, Errors: 0, Time elapsed: 0,217 sec
Testcase: testInit(tlc.TrafficLightBehaviorTest): FAILED
sec_red not set
junit.framework.AssertionFailedError: sec_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:25)
Test tlc.TrafficLightBehaviorTest FAILED
```

Or for a passed test:

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

6.13 Step 13: Integrate Software Components

The software components must be instantiated and connected as specified in Fig. 6.48. The following code fragment shows how to initialized and connect the components. The components can be connected using the constructor (e.g. `LightsDriver`) or they provide an additional interface (e.g., `connect`).

```
package tlc;
public class MainInit {

    BrokenLightDriver bld;
    EmergencyRequestDriver erd;
    InductionLoopDriver ild;

    LightsInterfaceAbstraction lia;
    LightsDriver ld;
    InductionLoopIAL ili;

    TrafficLightBehavior tlb;
    TimeOutTimer tot;
    Clock c;

    public MainInit() {
        ld = new LightsDriver();                    // Actuators
        lia = new LightsInterfaceAbstraction (ld);
        tlb = new TrafficLightBehavior();           // Application
        tot = new TimeOutTimer(tlb);
        c = new Clock(tot);

        tlb.connect(tot, lia);                      // Start Application

        ili = new InductionLoopIAL(tlb);
        bld = new BrokenLightDriver(tlb);           // Sensors
        erd = new EmergencyRequestDriver(tlb);
        ild = new InductionLoopDriver(ili);
    }

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

}
```

6.13.1 Validation

The complete software can be validated by running the test cases for the whole software as specified in Step 6.

6.14 Step 14: Integrate Hardware and Software

This step heavily depends on the used hardware and therefore is not performed.

7 Case Study: Automatic Teller Machine

In this chapter, the development process is applied on an Automatic Teller Machine (ATM) case study. The case study is taken from [CHH05b], extended with further diagrams and adopted on the development process presented in Chapter 5.

The mission of an ATM is to provide customers with money, provided that they are entitled to withdraw the desired amount. The Account-database , the Money supply, the Money Case, and the Card Reader already exist.

7.1 Step 1: Describe Problem

For this example no context diagram of the system in use is created since there is no machine in the system that should be replaced. Figure 7.1 shows the structure of the ATM problem context, where several domains and the corresponding shared phenomena are identified.

The ATM has to fulfill the following requirements:

R1 To use the ATM a valid pin and a bank card is required.

R2 The withdrawal should be refused when the request is bigger than the balance.

R3 The card should be retracted if the customer does not take the ejected card.

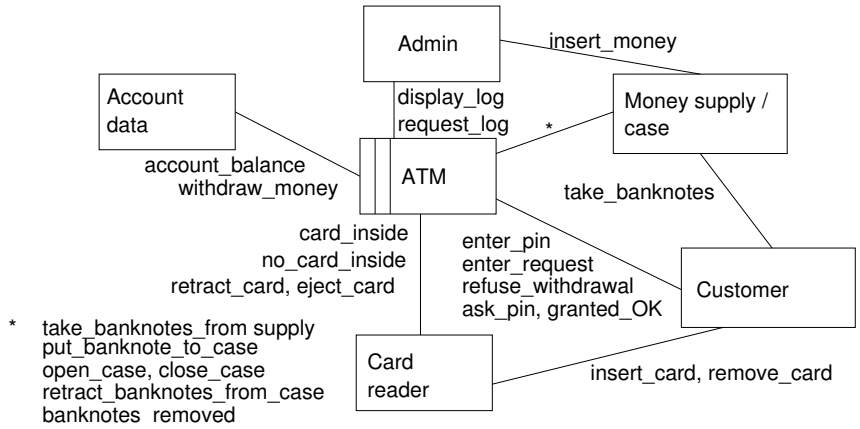


Figure 7.1: Context Diagram for ATM Problem

R4 The account is updated when the customer takes the money.

R5 After the withdrawal was granted and the card ejected, the money should be taken from the supply, put to the money case, and the case should be opened. After the customer took the money, the money case should close, otherwise the money should be retracted.

R6 All input phenomena should be logged.

R7 The logged input phenomena can be queried by the administrator.

The following domain knowledge can be stated:

D1 The Money case sends *banknotes_removed* when the Customer takes the banknotes.

D2 When the phenomena *take_banknotes_from_supply* and *put_banknote_to_case* occur, money will be in the Money supply.

D3 Only after *open_case* and before *close_case* occurs, the customer can take the banknotes.

D4 The Account data provides the actual balance and the balance can be updated.

D5 The Card reader can retract and eject the card. It monitors if a card ins inside or not.

The following statements can be assumed about the customer and the administrator:

A1 Before the Money supply is empty, the Administrator inserts money.

A2 The Administrator regularly checks the logs.

A3 The Customer enters the PIN, the request, and inserts the card to withdraw money.

Since the vocabulary for the ATM context is commonly known, no glossary is created.

All domains and phenomena mentioned in $A1$ – $A3$, $D1$ – $D5$, and $R1$ – $R7$ are in the context diagram. All domains and phenomena of the context diagram relate to some elements of $A1$ – $A3$, $R1$ – $R7$, and $D1$ – $D5$. Therefore, the domains and the phenomena in the context diagram and in A, R or D are consistent.

7.2 Step 2: Consolidate Requirements

To provide customers with money, the requirement $R5$ must be fulfilled and the domain knowledge $D1$ and $D2$ must be true. To check that the customers are entitled to withdraw the desired amount, the requirements $R1$ – $R4$ must be fulfilled, the domain knowledge $D3$ – $D5$ must be true, and the assumption $A3$ must be assumed to be true. The requirements $R6$ and $R7$ are not necessary to fulfill the system mission, but they should be implemented to detect system errors.

7.3 Step 3: Decompose Problem

The ATM consists of the subproblems *Authenticate*, *Request*, *Update Account*, *Take Money*, *Take Card*, *Log*, and *Display Log*. The following figures provide the problem frame instances for these subproblems.

Figure 7.2 shows the problem diagram for *Authenticate*. It is an instantiation of the commanded behavior frame with an additional feedback phenomenon AM!E6. Requirement $R1$ can be assigned to this problem diagram. The domains *Customer keypad* and *Display* are introduced as connection domains belonging to the machine.

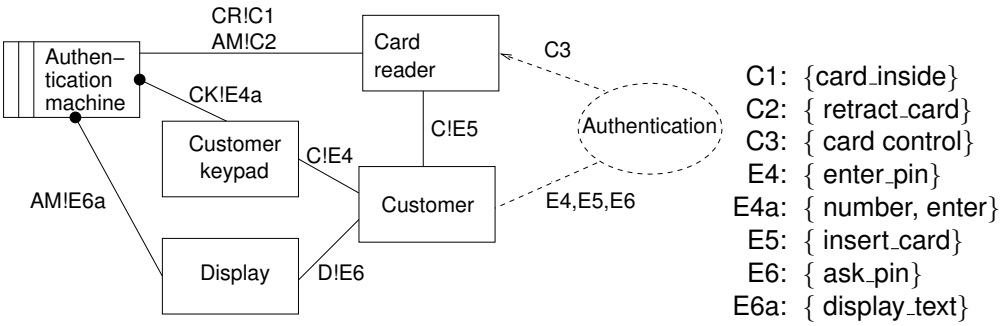


Figure 7.2: Problem Diagram for Authenticate (Commanded Behavior Variant)

The problem diagram for *Request* (shown in Figure 7.3) describes requirement $R2$. It is a variant of the commanded information frame. This variant is called information display and described in [CH04]. Here also the domains Customer keypad and Display are introduced as connection domains belonging to the machine.

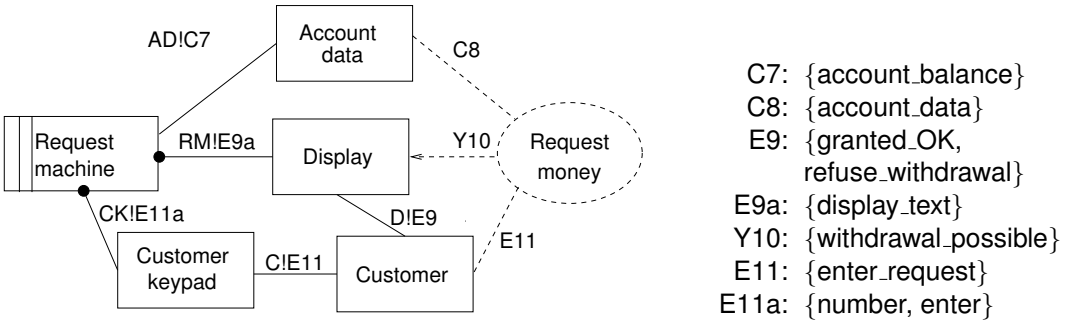
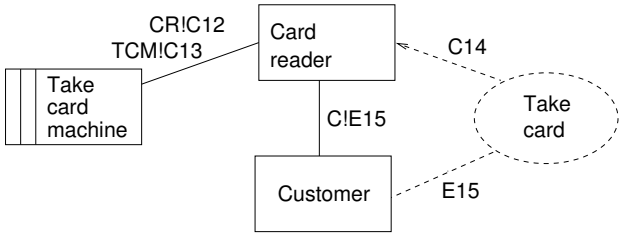


Figure 7.3: Problem Diagram for Request (Commanded Information Variant, Information Display)

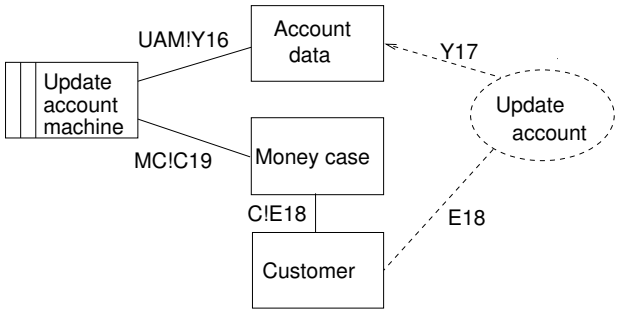
Figure 7.4 shows the problem diagram for *Take Card*. It is an instantiation of the required behavior frame. The Customer is additionally included to show all relevant actions in the environment. Requirement $R3$ can be assigned to this problem diagram.

Figure 7.5 shows the problem diagram for *Update Account*, which is a variant of the *Workpieces* frame. The requirement $R4$ is described with this problem frame.



C12: {card_inside}
C13: {eject_card, retract_card}
C14: {eject, retract}
E15: {take_card}

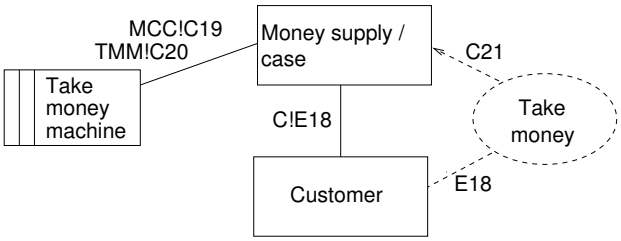
Figure 7.4: Problem Diagram for Take Card (Required Behavior)



Y16: {update_account}
Y17: {account_data}
E18: {take_banknotes}
C19: {banknotes_removed}

Figure 7.5: Problem Diagram for Update Account (Workpieces Variant)

The problem diagram for *Take Money* (Figure 7.6) is a variant of the *Required Behavior* problem frame (Figure 2.4), where we added the Customer and his/her connection with the Money Case. The requirement $R5$ is described with this problem frame.

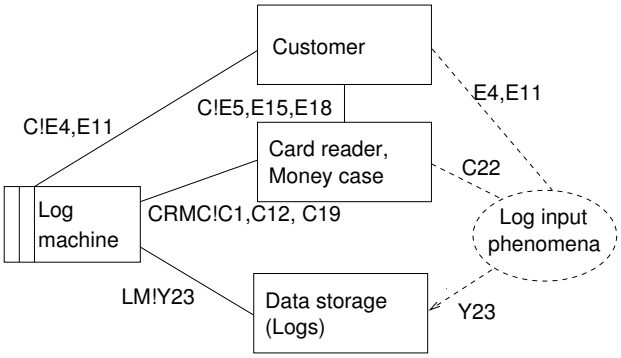


E18: {take_banknotes}
C19: {banknotes_removed}
C20: {take_banknotes_from_supply, put_banknote_to_case, open_case, close_case, retract_banknotes_from_case}
C21: {control_money_supply, control_money_case}

Figure 7.6: Problem Diagram for Take Money (Required Behavior Variant)

The problem diagram for *Log* (cf. requirement $R6$) is shown in Figure 7.7. The workpieces problem frame is instantiated. The domains Card reader, Money case, and Customer in the problem diagram represent the User in the problem frame.

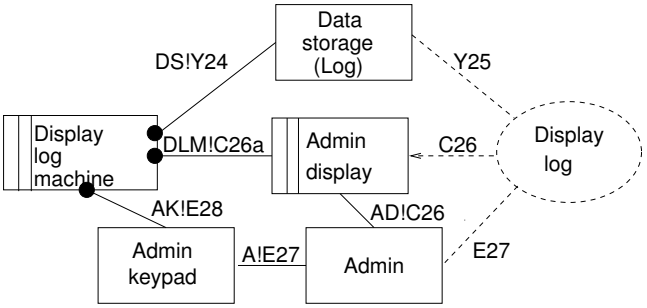
The problem diagram for *Display Log* (shown in Figure 7.8) describes requirement $R7$. It is a variant of the commanded information frame. This variant is called information display and described in [CH04]. The domains Data storage is a lexical domains being part of the



C22: {card_reader_money_case_input_phenomena}
Y23: {log_data}
C1, C12 . . . as given in the other figures

Figure 7.7: Problem Diagram for Log (Workpieces)

machine. The domains Admin display and Admin keypad are connection domains being part of the machine.



Y24: {log_data}
Y25: {logged_input_phenomena}
C26: {log}
C26a: {display_log}
E27: {request_log}
E28: {button_pressed}

Figure 7.8: Problem Diagram for Display Log (Commanded Information Variant, Information Display)

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.

$< start >::= (< idle >|| \ Log \ || \ DisplayLog)$
$< idle >::= (Authenticate < authenticated >| \ Authenticate < idle >)$
$< authenticated >::= (Request < granted >| \ Request < refused >)$
$< granted >::= (TakeCard < granted\_no\_card >| \ TakeCard < idle >)$
$< refused >::= TakeCardRefused < idle >$
$< granted\_no\_card >::= (UpdateAccount \ || \ TakeMoney) < idle >$

The last line means that once the card is removed and withdrawal is granted, both *UpdateAc-*

count and *TakeMoney* will take place in parallel, and then the idle state is reached.

7.4 Step 4: Derive Machine Behavior Specifications for each Subproblem

For each problem diagram, the specification is expressed by sequence diagrams that are given in the following figures.

The sequence diagram for the subproblem *Authenticate* is shown in Figure 7.9. This diagram expresses that the card is retracted after three unsuccessful attempts. The Customer is authenticated if the valid PIN is entered. The Authentication machine in this diagram contains the Customer keypad and the Display.

The sequence diagram for the subproblem *Request* is shown in Figure 7.10. When an authenticated customer enters a request, his/her balance is checked and the access for the customer is granted or refused. The Request machine in this sequence diagram includes the Customer keypad.

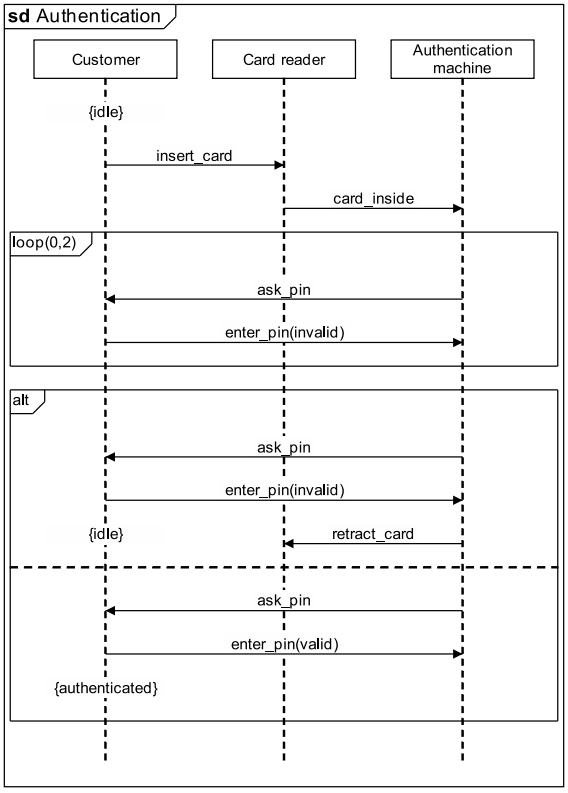


Figure 7.9: Sequence Diagram for Authenticate

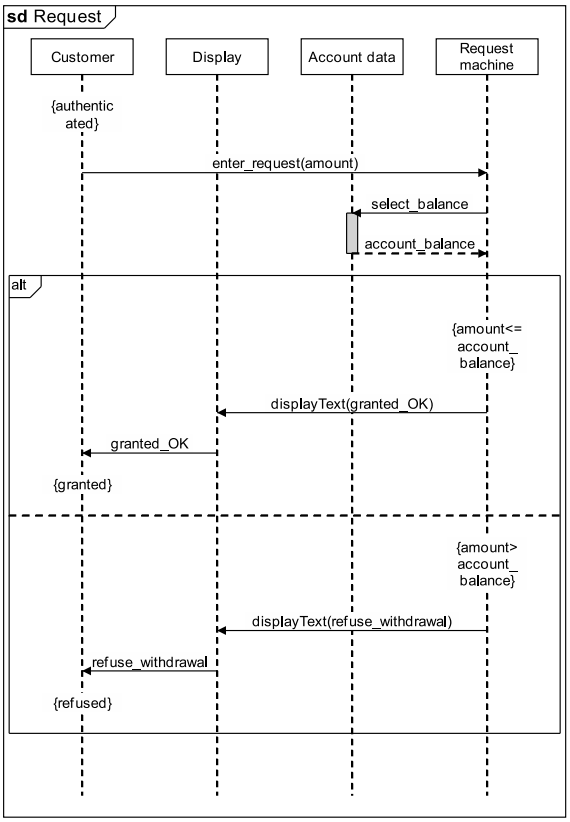


Figure 7.10: Sequence Diagram for Request

The sequence diagrams for the subproblem *Take Card* are shown in Figures 7.11 and 7.12. The sequence is different depending on the state of the customer. If the access is granted, the ejected card might be retracted after a certain time period, or the customer takes the card and he/she can continue with the withdrawal process.

If the access is refused, the ejected card is retracted or the customer takes it. In both cases the access is not granted, as shown in Fig. 7.12.

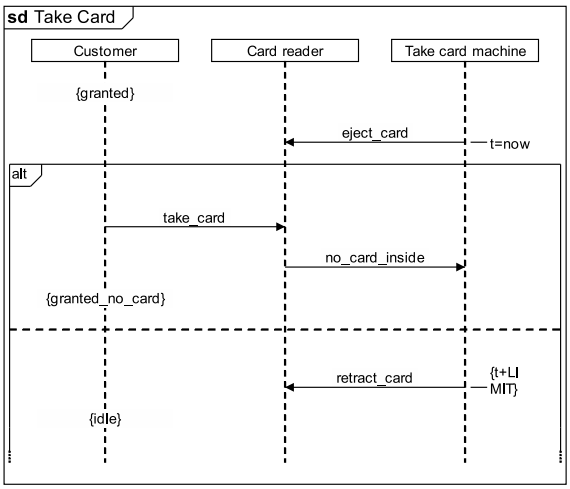


Figure 7.11: 1st Sequence Diagram for Take Card

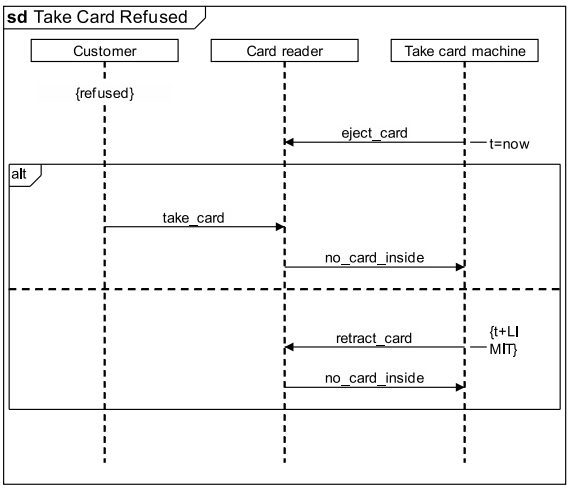


Figure 7.12: 2nd Sequence Diagram for Take Card

The sequence diagram for the subproblem *Update Account* is shown in Figure 7.13. The sequence diagram expresses that the account data are updated when the banknotes are removed.

The sequence diagram expresses the specification $S5$ for the subproblem *Take Money* which is shown in Figure 7.14. It also contains the domain Customer to illustrate the interrelation between the requirement, the domain knowledge and the specification:

R5 ... After the customer took the money, the Money Case should close, otherwise the money should be retracted.

D1 The Money Case sends *banknotes_removed* after the Customer took the banknotes.

S5 ... After the signal *banknotes_removed* occurs, the Money Case should close, otherwise the money should be retracted.

Therefore the implication $D1 \wedge S5 \Rightarrow R5$ is fulfilled. This sequence occurs in parallel to the sequence shown in Fig. 7.13. This is possible because both diagrams start with the same state invariant granted_no_card and only in Fig. 7.13 the state of the customer after this sequence is constraint.

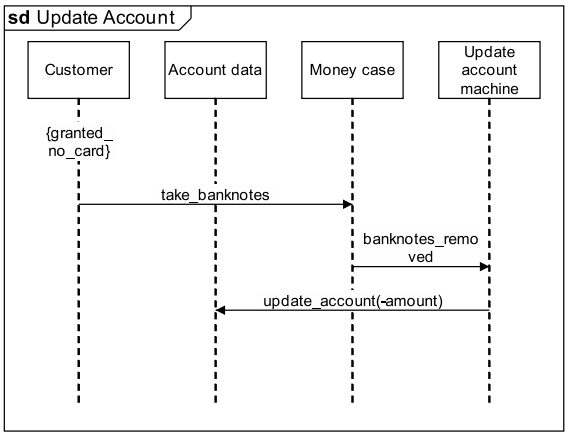


Figure 7.13: Sequence Diagram for Update Account

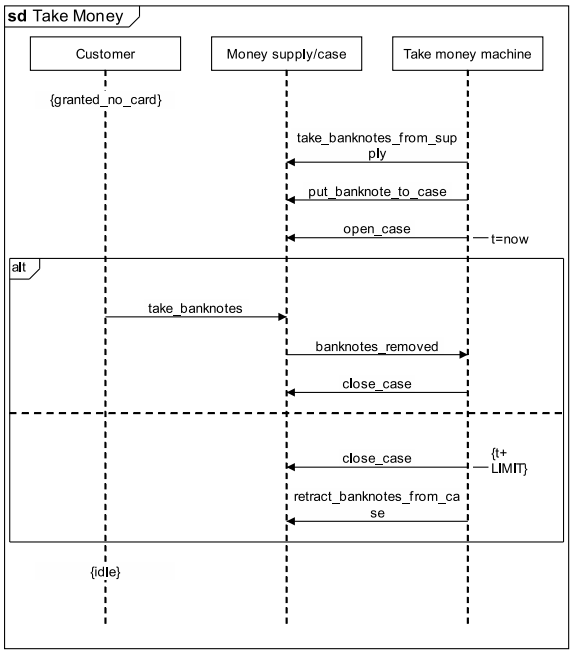


Figure 7.14: Sequence Diagram for Take Money

The sequence diagram for the subproblem *Log* is shown in Figure 7.15. It shows that all input signals are logged in a data storage. This sequence is not constraint by a state invariant and is parallel to all other sequence diagrams.

The sequence diagram for the subproblem *Display Log* is shown in Figure 7.16. It shows that the stored logs can be queried by the administrator. This sequence also is not constraint by a state invariant and is parallel to all other sequence diagrams.

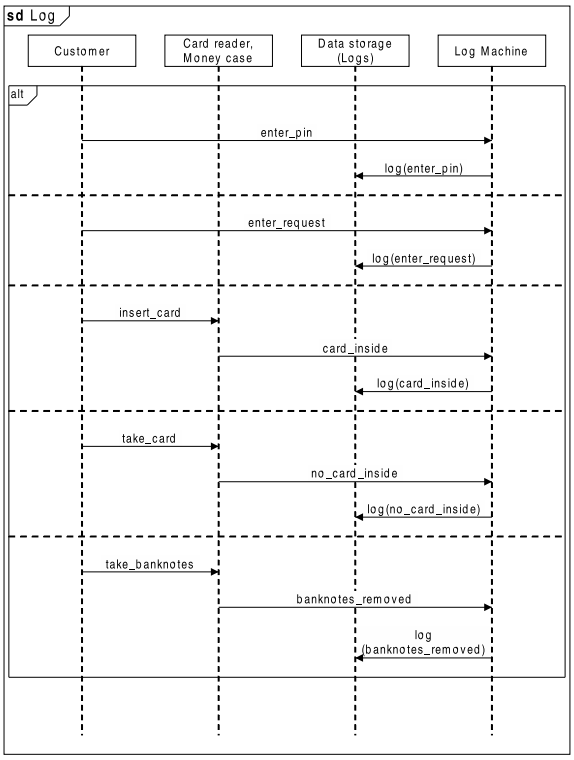


Figure 7.15: Sequence Diagram for Log

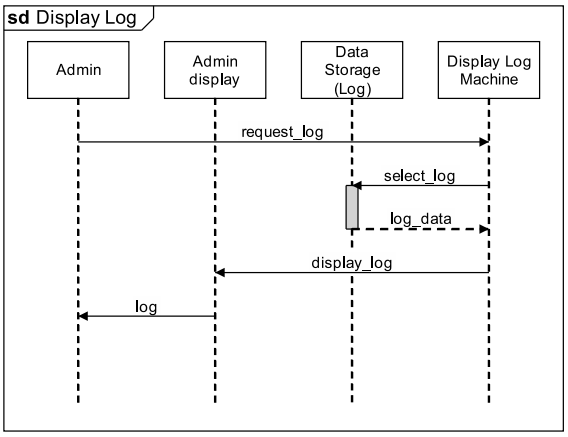


Figure 7.16: Sequence Diagram for Display Log

7.5 Step 5: Design Global System Architecture

From the context diagram and the problem diagrams the architecture in Fig. 7.17 can be developed.

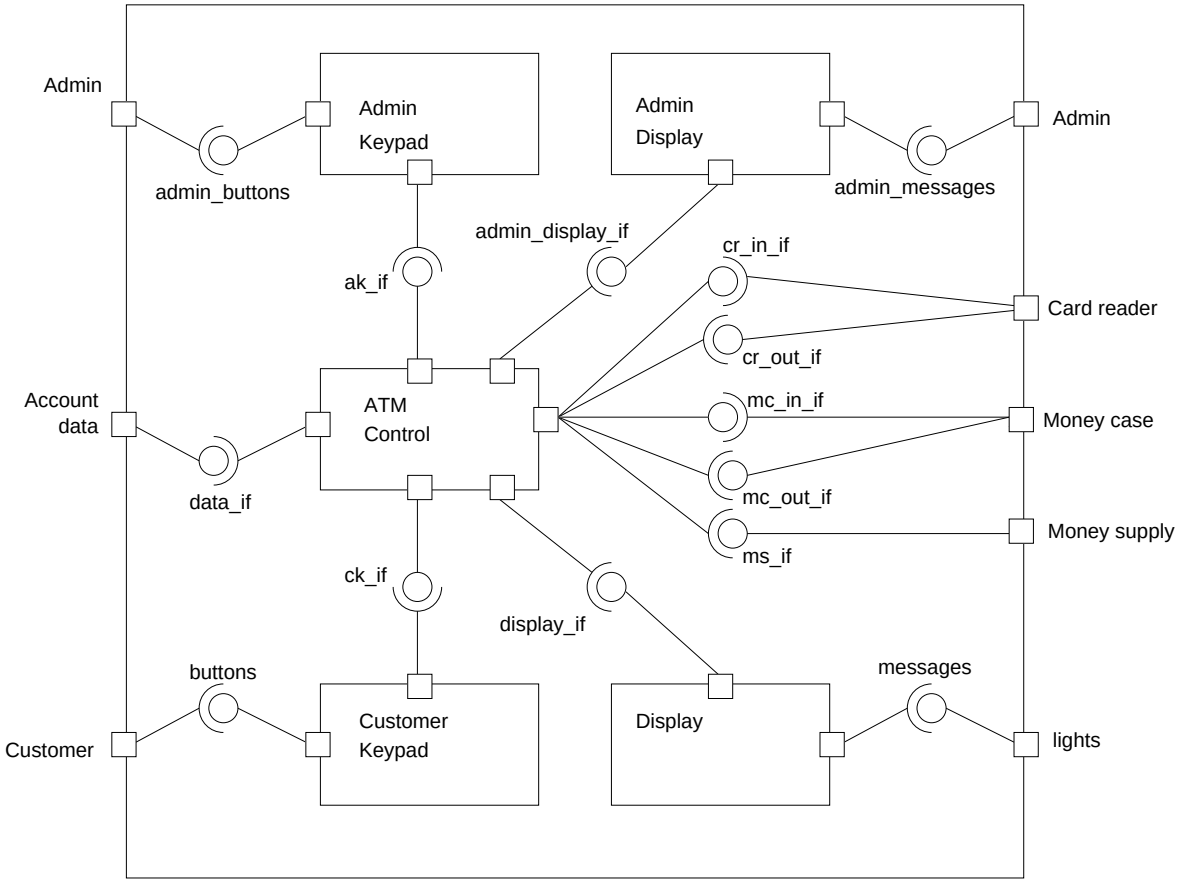


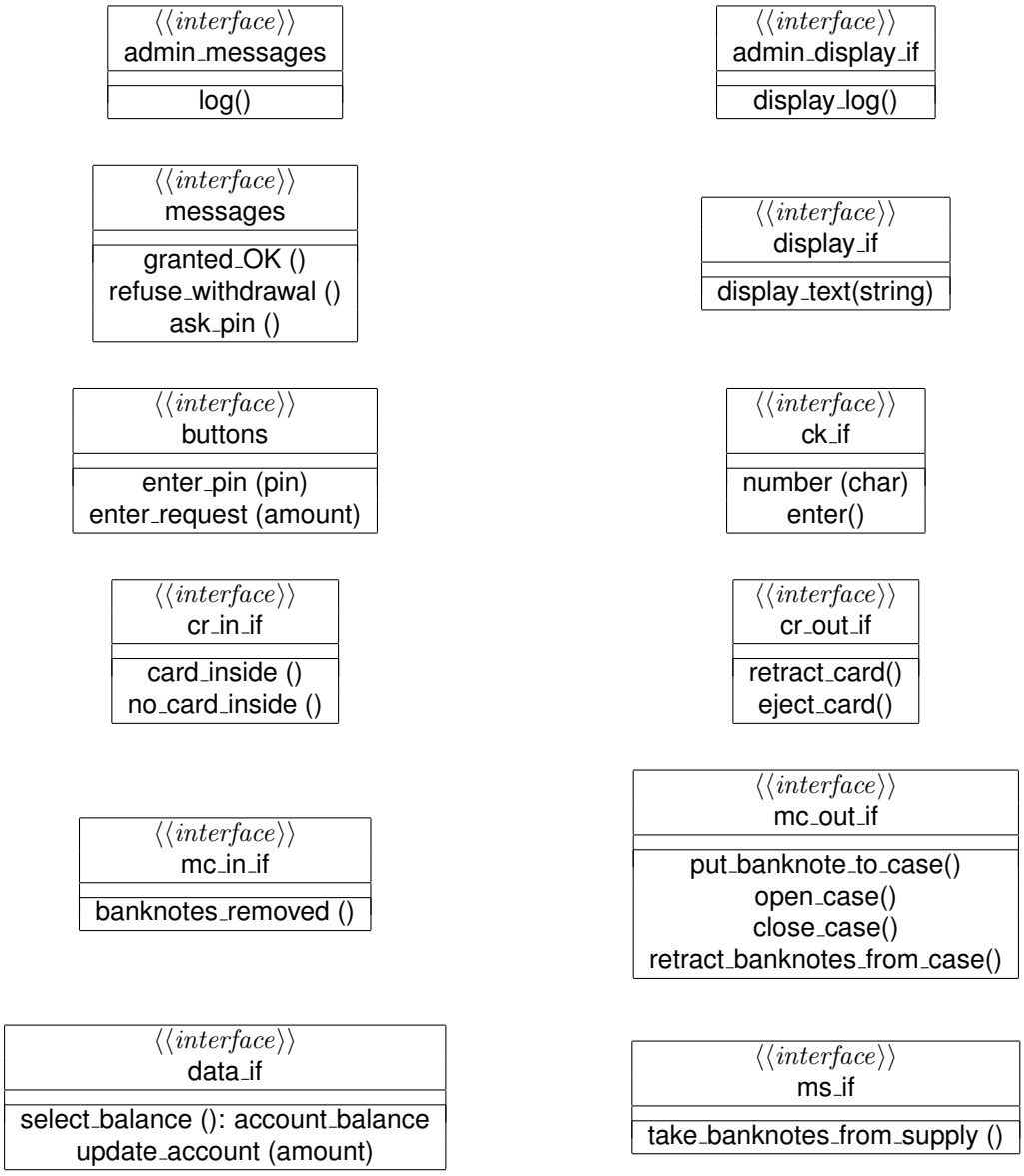
Figure 7.17: ATM System Architecture

All machines in the subproblems have to be implemented in the component ATM Control. The components Admin Keypad, Admin Display, Customer Keypad, and Display perform a simply transformation. Since the problems are very simple, these corresponding subproblems are not shown here.

From the sequences of interactions between machine and environment of all subproblems from Step 4 the following interface classes can be derived:

$\langle\langle interface \rangle\rangle$ admin_buttons
request_log()

$\langle\langle interface \rangle\rangle$ ak_if
button_pressed()



All subproblems should be implemented in the component ATM control. The following grammar for the component ATM control is therefore the same as in Step 3:

$< start >::= (< idle > || Log \ || \ DisplayLog)$
$< idle >::= (Authenticate < authenticated > | \ Authenticate < idle >)$
$< authenticated >::= (Request < granted > | \ Request < refused >)$
$< granted >::= (TakeCard < granted\_no\_card > | \ TakeCard < idle >)$
$< refused >::= TakeCardRefused < idle >$
$< granted\_no\_card >::= (UpdateAccount \ || \ TakeMoney) < idle >$

In the interface description all interfaces are captured. The signals in the sequence diagrams are consistent with the external interfaces in the subproblems. All subproblems are associated to the component ATM control.

7.6 Step 6: Derive Specifications for all Components being Relevant for the Subproblem

Instead of specifying the component ATM Control itself, the components Admin Keypad, Admin Display, Customer Keypad, and Display are described in Figures 7.18, 7.19, 7.20, and 7.21.

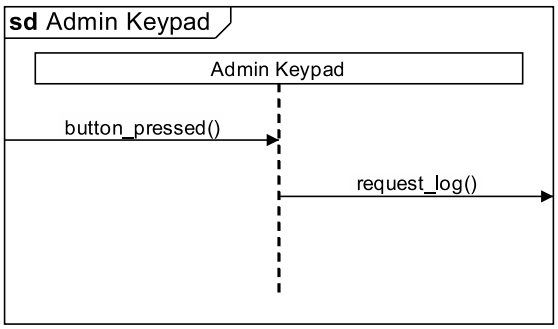


Figure 7.18: Sequence Diagram of the Admin Keypad Behavior

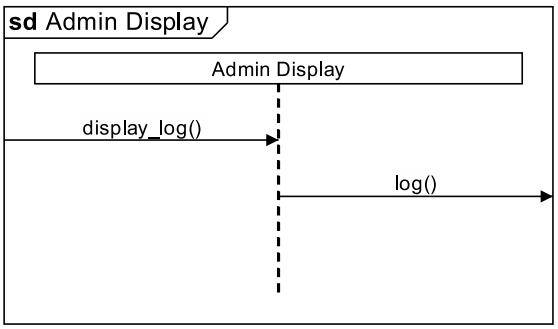


Figure 7.19: Sequence Diagram of the Admin Display Behavior

The specification of the software component can be derived directly from the specifications in Step 4 and these diagrams.

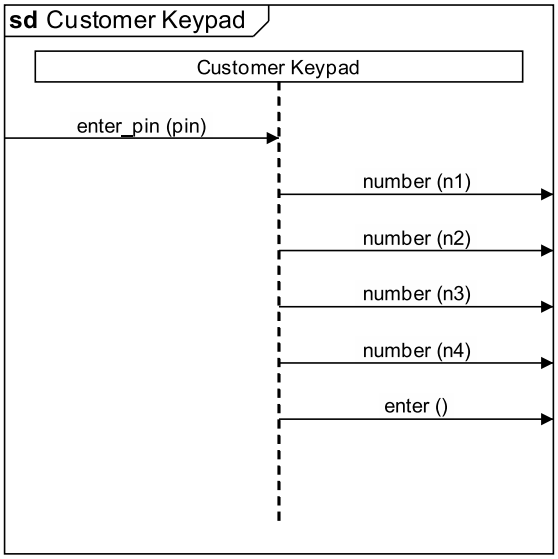


Figure 7.20: Sequence Diagram of Customer Keypad Behavior

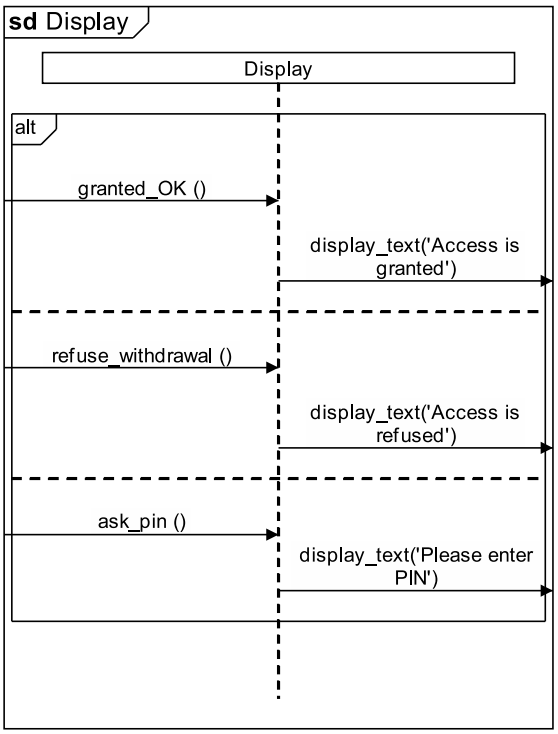


Figure 7.21: Sequence Diagram of the Display Behavior

7.7 Step 7: Design Software Architecture for each Software Component and each Subproblem

For each problem diagram, an architectural pattern from Chapter 5, Step 7 is instantiated.

The architecture for the subproblem *Authenticate* is shown in Fig. 7.22. It consists of an Authentication Application, a Card In IAL, a Card Out IAL, the corresponding HAL components, and a User Interface.

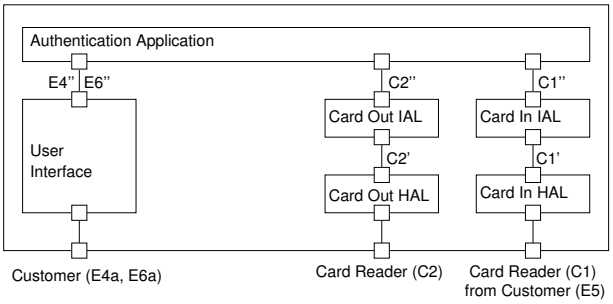


Figure 7.22: Architecture for Authenticate

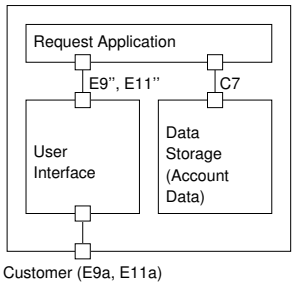


Figure 7.23: Architecture for Request

The architecture for the subproblem *Request* is shown in Fig. 7.23. It consists of a Request Application, a User Interface and a Data Storage.

The architecture for the subproblem *Take Card* (see Fig. 7.24) consists of the hardware abstraction layer and the interface abstraction layer for the Card, and a Take Card Application. The

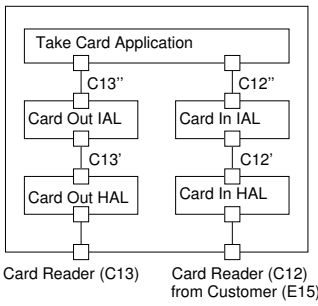


Figure 7.24: Architecture for Take Card

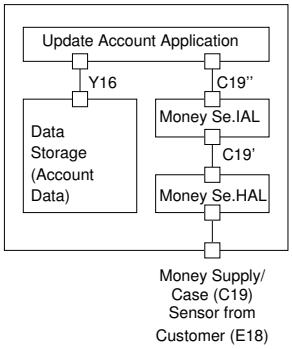


Figure 7.25: Architecture for Update Account

architecture for the subproblems *Update Account* is shown in Fig. 7.25. The architecture for the subproblems *Take Money* is shown in Fig. 7.26. The architecture for the subproblem *Log* consists of a Log Application and all components that handle input phenomena. It is shown in Fig.7.28. The architecture for the subproblem *Display Log* consists of a Display Log Application, a User Interface for the administrator, and a Data Storage containing the logs. (see Fig. 7.27).

As abstract phenomena are used for the case study, the IAL and the HAL are trivial, and most interface signals can be obtained simply by renaming the external signals.

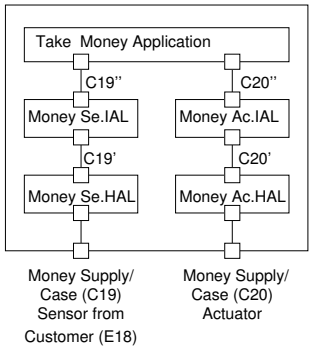


Figure 7.26: Architecture for Take Money

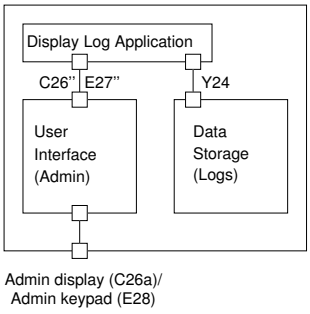


Figure 7.27: Architecture for Display Log

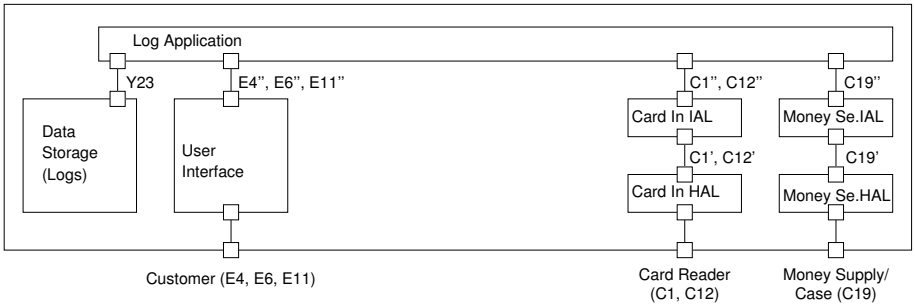


Figure 7.28: Architecture for Log

The following table maps the interface classes in the architectures to the interface classes in the system architecture.

C1 – C1' – C1'' – C12 – C12' – C12'' – cr_in_if
C2 – C2' – C2'' – C13 – C13' – C13'' – cr_out_if
E4a – E11a – ck_if
E4'' – E11'' – buttons
E6a – display_if
E6'' – E9'' – messages
C7 – Y16 – data_if
C19 – C19' – C19'' – ms_if
C20 – C20' – C20'' – mc_out_if
C26'' – admin_messages
C26a – admin_display_if
E27'' – admin_buttons
E28 – ak_if

The following interface classes show the correspondence for one type of interface classes.

$\langle\langle interface \rangle\rangle$ C13
retract_card() eject_card()

$\langle\langle interface \rangle\rangle$ C13'
retract_card'() eject_card'()

$\langle\langle interface \rangle\rangle$ C13''
retract_card''() eject_card''()

7.8 Step 8: Specify Behavior of Software Architecture Components for each Subproblem

The specification of the application components can be derived from the specification in Step 4. Because of the trivial HAL and IAL, the sequence diagrams for the Take Money Application can be constructed just by replacing e.g. *eject_card()* with *eject_card"()*.

The specification of the component User Interface is the same as described in Figures 7.20 and 7.21.

The specification of the component User Interface (Admin) is the same as described in Figures 7.18 and 7.19.

Because of the trivial HAL and IAL, the other components do simple transformation like *eject_card"()* into *eject_card'()*.

7.9 Step 9: Specify Software Components of Software Architectures for each Subproblem

For each component, the required and the provided interfaces are specified. Additionally, the local data of the components is defined using class diagrams. These class diagrams support the reuse of the specified components. As examples the class diagram for the Request Application and the Update Account Application are provided in the Figures 7.29 and 7.30.

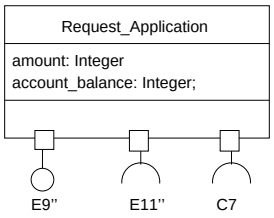


Figure 7.29: Class Diagram for Request Application

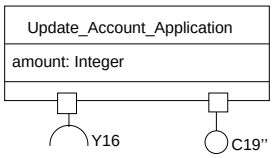


Figure 7.30: Class Diagram for Update Account Application

Each sequence diagram constructed in Step 8 can be transformed into a state machine that is associated to one class diagram. These state machines cover all signals that can occur in their environment.

The state machine for the Authenticate Application terminates if the valid pin is entered. It exits with failed after 3 unsuccessful attempts (cf. Fig. 7.31) as specified in Fig. 7.9.

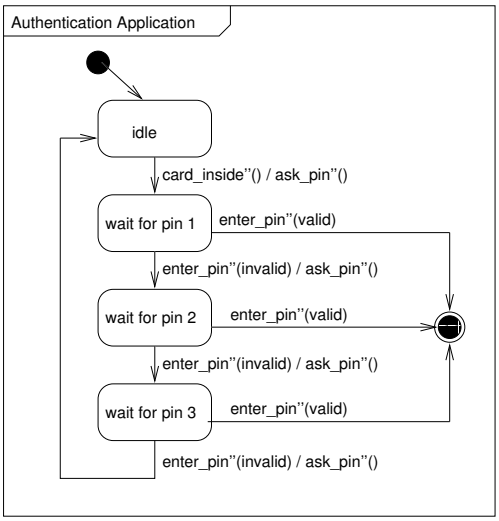


Figure 7.31: State Machine for Authenticate Application

The state machine for the Request Application is shown in Fig. 7.32. It is consistent with the sequence diagrams in the Figures 7.10 and 7.12.

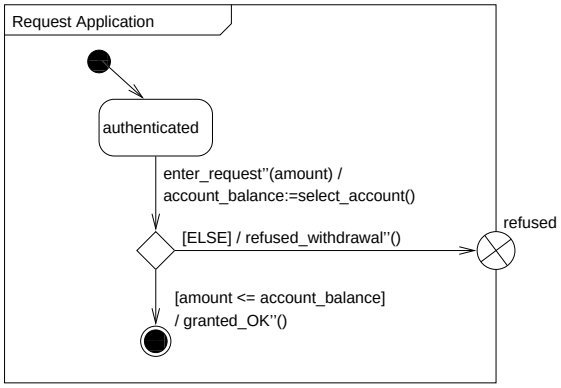


Figure 7.32: State Machine for Request Application

The state machine shown in Fig. 7.33 requires a timer as specified in Section 6.9.3.

The sequence diagram of Fig. 7.13 can be transformed into the state machine shown in Fig. 7.34.

The state machine for taking the money is shown in Fig. 7.35. It ejects the requested amount of money and retracts this money if it was not taken within a certain time limit.

Additionally there is a state machine that logs all input phenomena. This state machine consists of one state. In the transition, for each input signal (enter_pin, enter_request, no_card_inside, card_inside, and banknotes_removed) the signal log with an appropriate parameter (e.g. enter_pin) is sent (cf. 7.36).

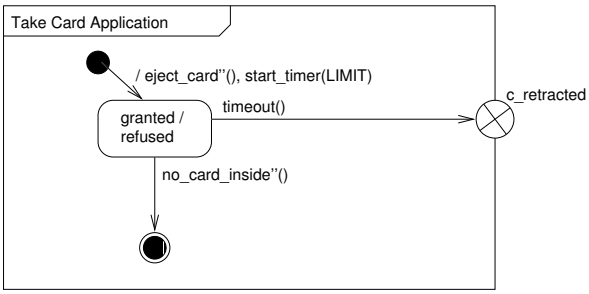


Figure 7.33: State Machine for Take Card Application

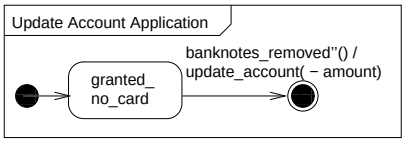


Figure 7.34: State Machine for Update Account Application

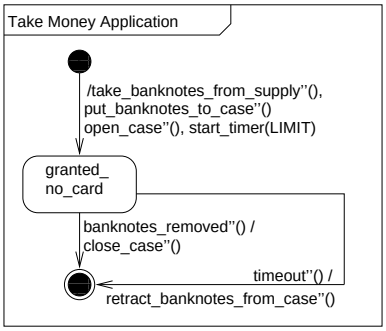


Figure 7.35: State Machine for Take Money Application

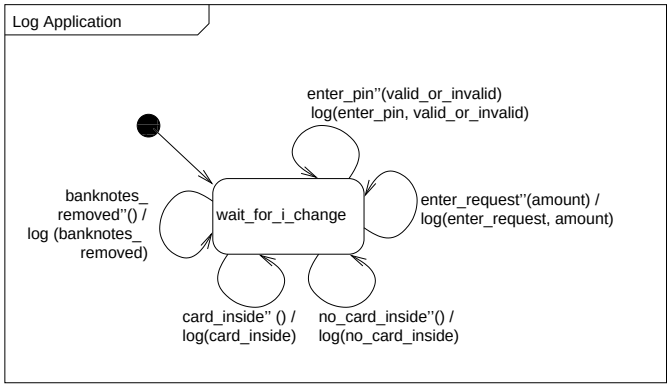


Figure 7.36: State Machine for Log Application

The logged data can be requested by the Admin. This functionality is implemented in the state machine for Display Log Application (cf. 7.37).

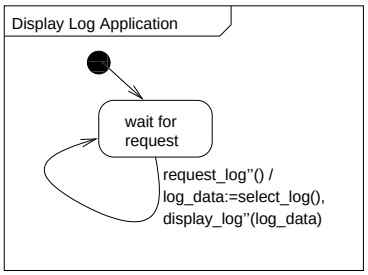


Figure 7.37: State Machine for Display Log Application

Also the other components must be specified with class diagrams and state machines. As an example a class diagram and the state machines for the User Interface components are presented in Figures 7.38, 7.39, 7.40, and 7.41.

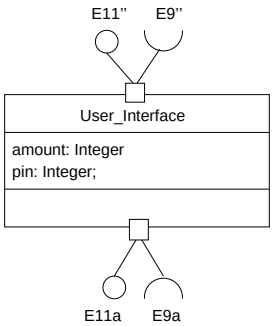


Figure 7.38: Class Diagram for User Interface

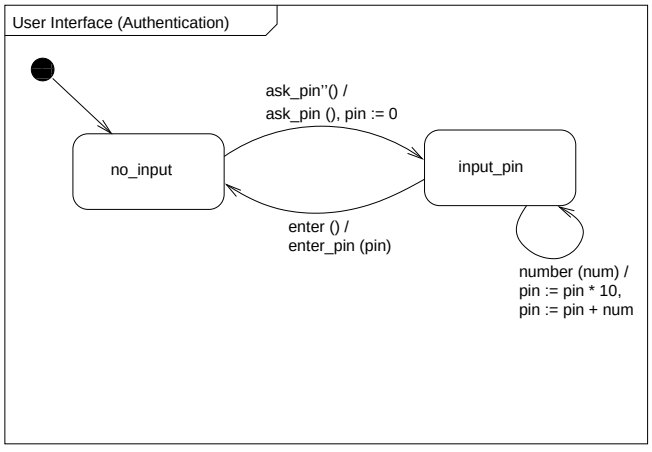


Figure 7.39: State Machine for Authenticate User Interface

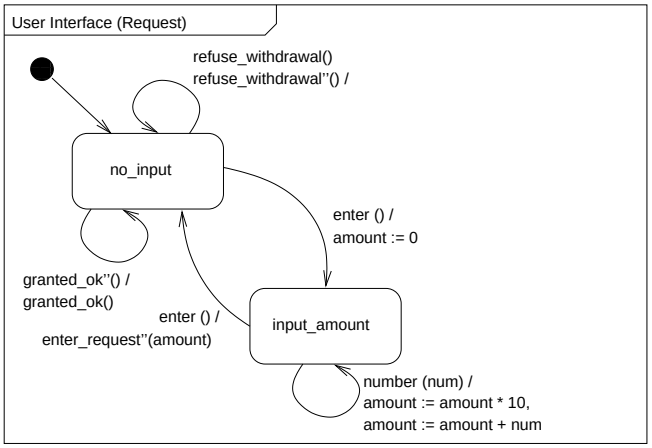


Figure 7.40: State Machine for Request User Interface

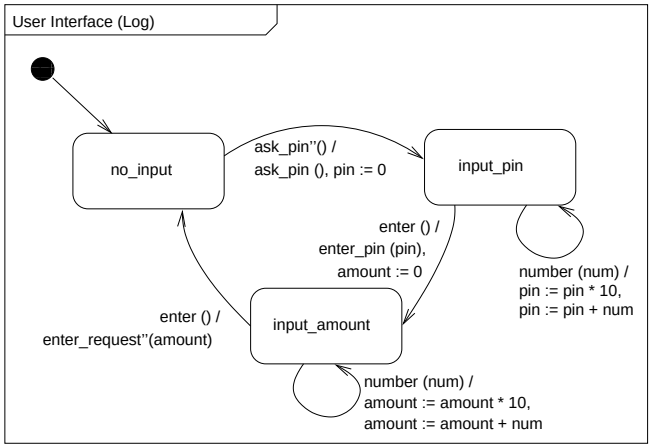


Figure 7.41: State Machine for Log User Interface

7.10 Step 10: Develop Global Software Architectures

The composed architecture for the ATM is shown in Figure 7.42. It shows that the patterns yield appropriate architectures for subproblems fitting to problem frames. It ist also shown that these architectures can be combined in a modular way to obtain an architecture of the overall system according to the rules of Step 10. The following merges have been done:

The problem *Take Money* and the problem *Update Account* are parallel and share some input phenomena (cf. case 4 in Step 10 of Chapter 5). We decided to merge the corresponding application components. The problem *Log* is related parallel to all other subproblems, sharing input phenomena (cf. case 4). We decided to merge the Log Application component with Authenticate Application, Request Application, Update Account Application and the merged application for Take Money/Update Account. The problems *Authenticate*, *Request*, *Update Account* and the merged problem *Take Money*/*Update Account* are related sequentially or by alternative (cf. case 2). Therefore the corresponding applications are also merged. We call the resulting component Main Application. The problem *Display Log* is parallel and does not share any interface phenomena (cf. case 5). Hence, the component Display Log Application is not merged. All components that are IALs or HALs (cf. case 1) are merged with the components of the same name in the other subproblem architectures.

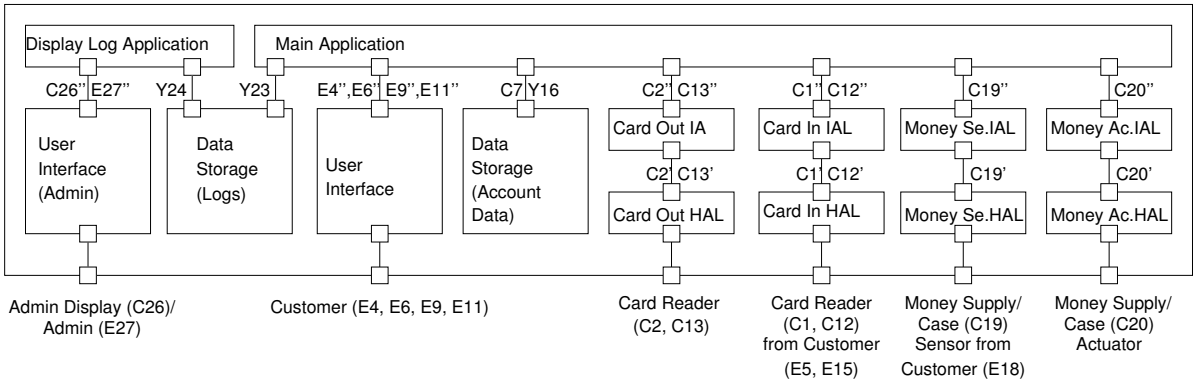


Figure 7.42: Composed Architecture

7.11 Step 11: Specify Composed Software Components

To create the complete state machines we start with the application components.

The application component state machines for Take Money and Update Account are merged by adding the output signal *update_account(-amount)* to the transition in the state machine Take Money activated by *banknotes_removed()* (cf. Fig. 7.43).

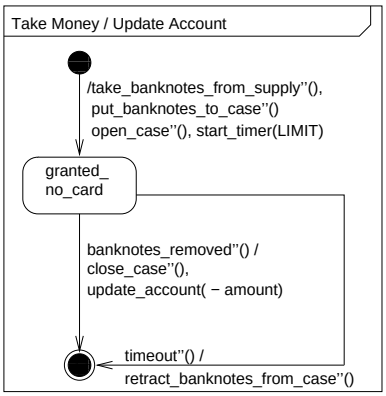


Figure 7.43: Merged State Machine for Take Money and Update Account Application

Then the state machines for Authentication Application, Request Application, and Take Money/Update Account has to be merged with the state machine for the Log Application using the same technique. This merge is presented exemplarily for the state machine Take Money Update Account. Figure 7.44 shows the result of the merge.

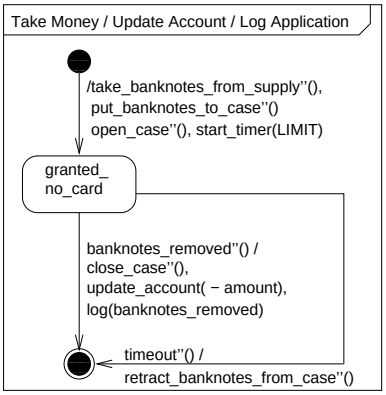


Figure 7.44: Merged State Machine for Take Money, Update Account, and Log Application

After merging the necessary state machines for the parallel subproblems in the application component, the state machines for the sequential and the alternative subproblems can be combined using composite states (see Fig. 7.45). The resulting state machine exactly reflects the grammar describing the dependencies of the subproblems.

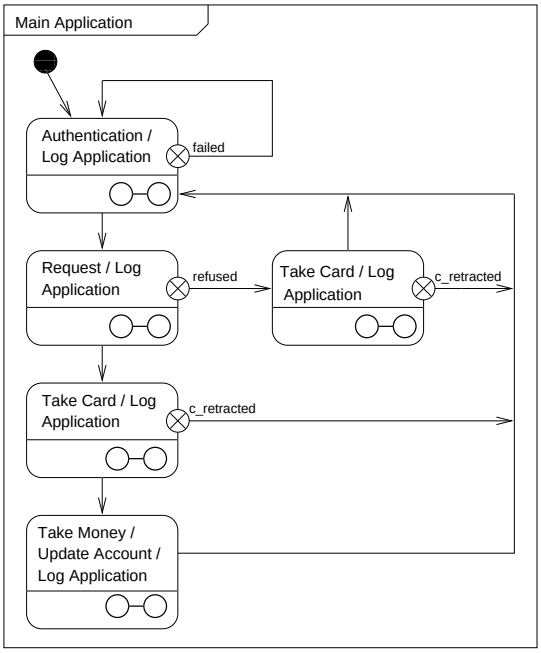


Figure 7.45: State Machine for all Sequential and Alternative Problems

Then the state machines for the IALs, the HALs and the User Interfaces must be merged. The merged state machine for the User Interface is shown in Figure 7.46.

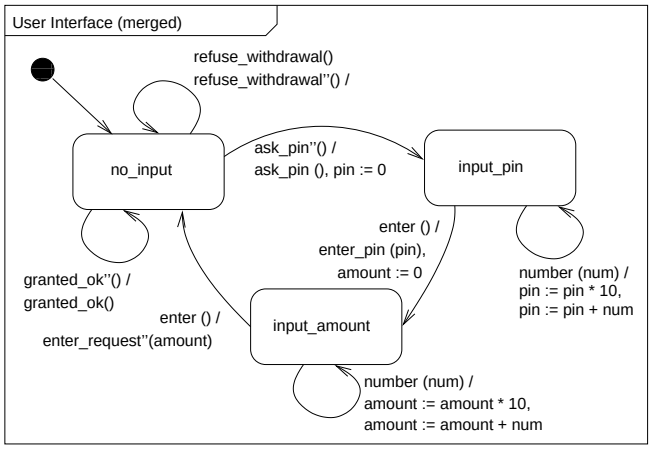


Figure 7.46: Merged State Machine for the User Interface

7.12 Step 12: Implement Software Components and Test Environment

This step can be performed using the same heuristics as described in Section 6.12.

7.13 Step 13: Integrate Software Components

This step can be performed using the same heuristics as described in Section 6.13.

7.14 Step 14: Integrate Hardware and Software

This step heavily depends on the used hardware and therefore it is not performed.

8 Conclusion

8.1 Summary

The *Development Process for Embedded Systems* (DPES) developed in this thesis by refining and adapting the process defined in [HH05b], has the following important characteristics:

The process is **model-based**. Modeling is used for problems, specifications, architecture and component behavior. Consistency checks between the several views of the machine are possible (independently from the used tool), because UML provides a standardized XML-based file format that can be parsed easily.

The process covers not only software but the whole system, consisting of **software and hardware**. Within the process, the hardware-software-partitioning problem is addressed. System and software are specified using the same notation. Therefore, the specification can be refined on the system level (Step 5). This is necessary if more behavioral information is required before the hardware-software-partitioning can be performed. Since hardware and software is covered by the process, machines with redundant hardware can be specified and the influences to the software developed can be described.

The process is **tailored to embedded systems**. The application domains of many embedded systems can be covered by the four-variable-model proposed by Parnas [DLP95]. Apart from the hardware abstraction layer, the four-variable-model is the most important design criterion for the layered architecture proposed in the development process.

The DPES supports the **reuse of components** already in the specification phase (see Step 7). Reuse can further be supported by using design patterns.

In large parts, the process makes use of **UML 2.0**. UML 2.0 combines the advantages of the widely known UML and the Specification and Definition Language (SDL) that is used for telecommunication protocols. In contrast to UML 1.4, the layered architecture can be expressed adequately with UML 2.0. In contrast to SDL, UML 2.0 allows a much more flexible structure of components that allows better reuse of components.

The DPES can be mapped onto the V-Model [BD93], as shown in Fig. 8.1.The development of **test cases** is an elementary part of the process. The development of test cases is structured, problem-based and requirement-based. The test specifications are expressed as sequence diagrams (see Steps 4, 6 and 8), and test cases can be derived (or generated) from these diagrams just by replacing points of time with time frames expressing when desired events are expected. Therefore, the sequence diagrams are the link between the tests and the specifications.

The item *Component design* in the V-Model is mapped to the software component architecture

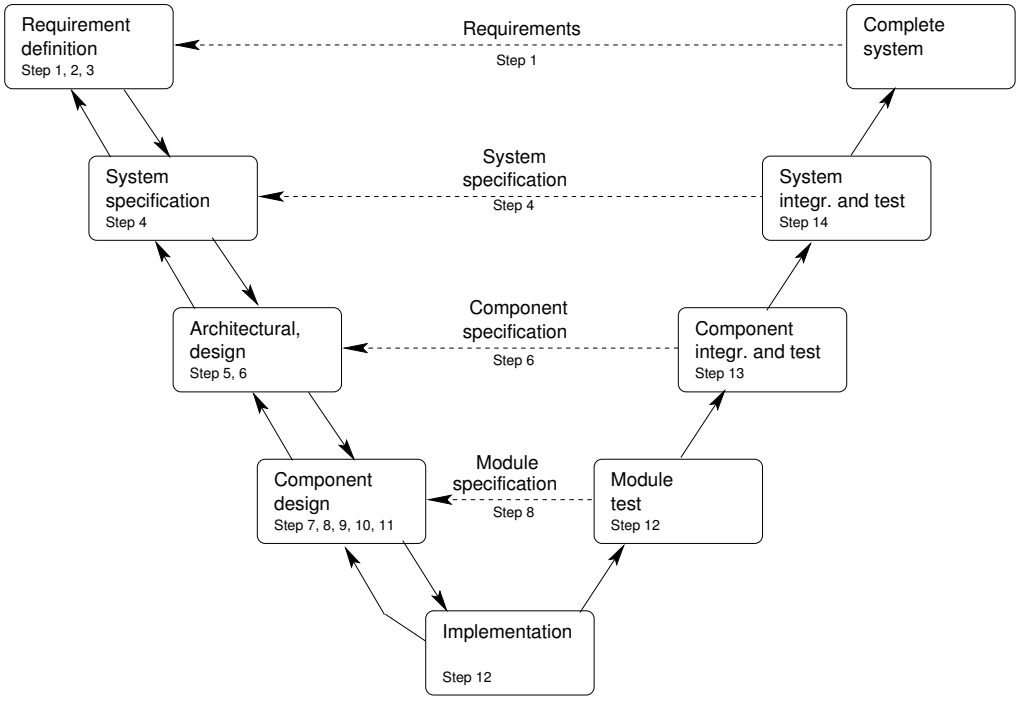


Figure 8.1: Mapping to the V-Model

(Step 7 for subproblems and Step 10 composed) and to the specification of its components (Step 8 for subproblem sequences, Step 9 for subproblem state machines and Step 11 for composed state machines).

The DPES can also be applied on software development projects, where no hardware has to be developed. In this case Step 5, 6, and 13 can be omitted.

For each step of the development process, **validation conditions** have been defined. These conditions can be checked using reviews and inspections. However, for many of the validation conditions, formal proof or demonstration is also possible.

The process is defined in such a way that **tool support** can be added in a modular way, based on existing tools.

By applying the process described in [HH05b], some problem occurs and points for improvements of the development process have been identified. E.g., the process did not cover the case, when a machine exists, that has to be replaced. The process gave little help for designing the system and software architecture, the sequence diagrams were too difficult for some problems, and an incomplete specification was detected in a very late step. These problems are handled by the process presented in Chapter 5. Table 8.1 shows the steps that have been modified to solve the described problems.

Finally, the process has been developed in an industrial context, and it was **successfully applied in practice** in several projects for developing security- and safety-critical systems. The improved process has been checked, by applying it on two case studies.

Step / Problem	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Existing Machine	×													
System Architecture Design			×		×									
Subproblem Composition			×		×					×	×			
Readability of Sequence Diagrams				×		×								
Interoperability with other Methods					×	×								
Software Architecture Design							×							
Interface Control							×							
Notation				×										
Interface Specification				×										
IAL and HAL Specification								×						
Jackson – Four Variable Model Integration			×											
Contradicting Requirements				×					×		×			
Contradicting Requirements				×					×		×			
Completeness of Specification				×					×		×			

Table 8.1: Mapping: Points for Improvements - Steps of DPES

8.2 Future work

The improved process should be applied in an industrial context to develop complex real-life applications.

This DPES can be applied without special tools. But especially the validation conditions can be checked automatically. The standardized XMI file format defined by the OMG [UML] can be used to check the consistency between several models created during the development process.

To develop subcomponents the Steps 4 and 5 or the corresponding steps for the software components must be applied recursively. The input and output of these steps should be modified to allow a better subcomponent development.

In Steps 12 and 13 heuristics for other programming languages should be added. For all languages rules should be developed that help to integrate composed components.

It should be checked how and where the behavioral and structural patterns defined e.g., in [KJ04] can be integrated in the developed process.

The process can be enhanced by using formal methods. Then, it should be possible to export the (UML-)models to formal verification tools such as Atelier B, FDR, SPIN or SVM. Within these tools safety or security properties can be checked. For hardware-software-codesign, export from and to VHDL is very usefull.

The UML-diagrams used in the process are contained in the draft version of SysML [Sys05]. Additional elements of SysML could be used to describe same aspect more precisely.

This process contains the problem frames developed by Jackson and the corresponding architectural patterns. Additional problem frames and corresponding architectural patterns, especially for security or safety problems, should be added. Once developed, they help to reuse challenges from other projects. Additionally, a base of reuseable components can be developed, that can be integrated in the architectures and reduce the development effort.

Bibliography

[BCK98] L. Bass, P. Clements, and R. Kazman. Software Architecture in Practice. Addison-Wesley, 1998.

[BD93] A.-P. Bröhl and W. Dröschel. Das V-Modell. Oldenbourg, 1993.

[BH99] R. Bharadwaj and C. Heitmeyer. Hardware/software co-design and co-validation using the scr method. In Proceedings IEEE International High-Level Design Validation and Test Workshop (HLDV 99), 1999.

[BP03] Manfred Broy and Wolfgang Pree. Ein Wegweiser für Forschung und Lehre im Software-Engineering eingebetteter Systeme. Informatik Spektrum, 18:3–7, Februar 2003.

[CC99] Common criteria for information technology security evaluation, 1999. aligns to ISO/IEC 14508:1999, see http://www.commoncriteria.org.

[CD01] J. Cheesman and J. Daniels. UML Components – A Simple Process for Specifying Component-Based Software. Addison-Wesley, 2001.

[CH04] Christine Choppy and Maritta Heisel. Une approache à base de "patrons" pour la spécification et le développement de systèmes d'information. In Proceedings Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL'2004, pages 61–76, 2004.

[CHH05a] Christine Choppy, Denis Hatebur, and Maritta Heisel. Architectural patterns for problem frames. IEE Proceedings – Software, Special Issue on Relating Software Requirements and Architectures, 152(4):198–208, 2005.

[CHH05b] Christine Choppy, Denis Hatebur, and Maritta Heisel. Composing architectures based on architectural patterns for problem frames. Technical report, Université Paris XIII and Universität Duisburg-Essen, December 2005. `http://swe.uni-duisburg-essen.de/intern/comparch05.pdf`.

[DCJ94] Stephanie Bodoff Chris Dollin Helena Gilchrist Fiona Hayes Derek Coleman, Patrick Arnold and Paul Jeremaes. Object-Oriented Development, The Fusion Method. Prentice-Hall, 1994.

[DLP95] J. Madey D. L. Parnas. Functional documents for computer systems. In Science of Computer programming, volume 25, pages 41–61, 1995.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, 1995.

[Hei98] M. Heisel. Agendas – a concept to guide software development activites. In R. N. Horspool, editor, Proc. Systems Implementation 2000, pages 19–32. Chapman & Hall London, 1998.

[HH05a] Denis Hatebur and Maritta Heisel. Problem frames and architectures for security problems. In Bjørn Axel Gran, Rune Winter, and Gustav Dahll, editors, Proceedings of the 24th International Conference on Computer Safety, Reliability and Security (SAFECOMP), LNCS 3688, pages 390–404. Springer-Verlag, 2005.

[HH05b] Maritta Heisel and Denis Hatebur. A model-based development process for embedded systems. In T. Klein, B. Rumpe, and B. Schätz, editors, Proc. Workshop on Model-Based Development of Embedded Systems, number TUBS-SSE-2005-01. Technical University of Braunschweig, 2005. Available at http://www.sse.cs.tu-bs.de/publications/MBEES-Tagungsband.pdf.

[HS99] M. Heisel and J. Souquières. A method for requirements elicitation and formal specification. In J. Akoka, M. Bouzeghoub, I. Comyn-Wattiau, and E. Métais, editors, Proceedings 18th International Conference on Conceptual Modeling, ER'99, LNCS 1728, pages 309–324. Springer-Verlag, 1999.

[Int98] International Electrotechnical Commission. Functional safety of electrical/electronic/programmable electronic safty-relevan systems - part 1: General requrements, 1998.

[Jac95] M. Jackson. Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices. Addison-Wesley, 1995.

[Jac01] M. Jackson. Problem Frames. Analyzing and structuring software development problems. Addison-Wesley, 2001.

[JZ95] M. Jackson and P. Zave. Deriving specifications from requirements: an example. In Proceedings 17th Int. Conf. on Software Engineering, Seattle, USA, pages 15–24. ACM Press, 1995.

[KJ04] Michael Kirchner and Prashant Jain. Pattern-Oriented Software Architecture - patterns of resource management, volume Volume 3. John Wiley & Sons Ltd, 2004. 07TWQ5120.

[OMG05] Object Management Group OMG. Uml superstructure specification, v2.0, 2005. availble under http://www.omg.org/docs/formal/05-07-04.pdf.

[RHJN04] L. Rapanotti, J. G. Hall, M. Jackson, and B. Nuseibeh. Architecture driven problem decomposition. In Proceedings of 12th IEEE International Requirements Engineering Conference (RE'04), Kyoto, Japan, 6-10 September 2004.

[Sim04] David E. Simon. An Embedded Software Primer. Addison-Wesley, 2004.

[Sys05] SysML Parners. Systems Modeling Language (SysML) Specification, 2005. see http://www.sysml.org.

[Tan92] Andrew S. Tanenbaum. Modern Operating Systems. Prentice Hall, 1992.

[UML] UML Revision Task Force. OMG UML Specification. `http://www.uml.org`.

[ZJ97] P. Zave and M. Jackson. Four dark corners for requirements engineering. ACM Transactions on Software Engineering and Methodology, 6(1):1–30, January 1997. Also availble under http://www.research.att.com/~pamela/ori.html#fre.