Chapter 7
Systematic Architectural Design based on Problem Patterns

Christine Choppy and Denis Hatebur and Maritta Heisel

C. Choppy Universite Paris 13, LIPN, CNRS UMR 7030 Christine.Choppy@lipn.univ-paris13.fr

D. Hatebur University Duisburg-Essen Denis.Hatebur@uni-duisburg-essen.de

M. Heisel University Duisburg-Essen maritta.heisel@uni-duisburg-essen.de

Abstract We present a method to derive systematically software architectures from problem descriptions. The problem descriptions are based on the artifacts that are set up when following Jackson's problem frame approach. They include a context diagram describing the overall problem situation and a set of problem diagrams that describe subproblems of the overall software development problem. The different subproblems should be instances of problem frames, which are patterns for simple software development problems.
Starting from these pattern-based problem descriptions, we derive a software architecture in three steps. An initial architecture contains one component for each subproblem. In the second step, we apply different architectural and design patterns and introduce coordinator and facade components. In the final step, the components of the intermediate architecture are re-arranged to form a layered architecture, and interface and driver components are added.
All artefacts are expressed as UML diagrams, using specific UML profiles. The method is tool-supported. Our tool supports developers in setting up the diagrams, and it checks different validation conditions concerning the semantic integrity and the coherence of the different diagrams.
We illustrate the method by deriving an architecture for an automated teller machine.

7.1 Introduction

A thorough problem analysis is of invaluable benefit for the systematic development of high-quality software. Not only is there a considerable risk that software development projects fail when the requirements are not properly understood, but also the artefacts set up during requirements analysis can be used

as a concrete starting point for the subsequent steps of software development, in particular, the development of software architectures.

In this chapter, we present a systematic method to derive software architectures from problem descriptions. We give detailed guidance by elaborating concrete steps that are equipped with *validation conditions*. The method works for different types of systems, e.g., for embedded systems, web-applications, and distributed systems as well as standalone ones. The method is based on different kinds of patterns. On the one hand, it makes use of *problem frames* [20], which are patterns to classify simple software development problems. On the other hand, it builds on architectural and design patterns.

The starting point of the method is a set of diagrams that are set up during requirements analysis. In particular, a *context diagram* describes how the software to be developed (called *machine*) is embedded in its environment. Furthermore, the overall software development problem must be decomposed into simple subproblems, which are represented by *problem diagrams*. The different subproblems should be instances of problem frames.

From these pattern-based problem descriptions, we derive a software architecture that is suitable to solve the software development problem described by the problem descriptions. The problem descriptions as well as the software architectures are represented as UML diagrams, extended by stereotypes. The stereotypes are defined in profiles that extend the UML metamodel [29].

The method to derive software architectures from problem descriptions consists of three steps. In the first step, an initial architecture is set up. It contains one component for each subproblem. The overall machine component has the same interface as described in the context diagram. All connections between components are described by stereotypes (e.g., ‹‹call_and_return››, ‹‹shared_memory››, ‹‹event››, ‹‹ui››).

In the second step, we apply different architectural and design patterns. We introduce coordinator and facade components and specify them. A facade component is necessary if several internal components are connected to one external interface. A coordinator component must be added if the interactions of the machine with its environment must be performed in a certain order. For different problem frames, specific architectural patterns are applied.

In the final step, the components of the intermediate architecture are re-arranged to form a layered architecture, and interface and driver components are added. This process is driven by the stereotypes introduced in the first step. For example, a connection stereotype ‹‹ui›› motivates to introduce a user interface component. Of course, a layered architecture is not the only possible way to structure the software, but a very convenient one. We have chosen it because a layered architecture makes it possible to divide platform-dependent from platform-independent parts, because different layered systems can be combined in a systematic way, and because other architectural styles can be incorporated in such an architecture. Furthermore, layered architectures have proven useful in practice.

Our method exploits the subproblem structure and the classification of subproblems by problem frames. Additionally, most interfaces can be derived from the problem descriptions [19]. Stereotypes guide the introduction of new components. They also can be used to generate adapter components automatically.

The re-use of components is supported, as well.
The method is tool-supported. We extended an existing UML tool by providing two new profiles for it. The first UML profile allows us to express the different models occurring in the problem frame approach using UML diagrams. The second one allows us to annotate composite structure diagrams with information on components and connectors. In order to automatically validate the semantic integrity and coherence of the different models, we provide a number of validation conditions. The underlying tool itself, which is called UML4PF, is based on the Eclipse development environment [1], extended by an EMF-based [2] UML tool, in our case, Papyrus UML [3].
In the following, we first discuss the basic concepts of our method, namely problem frames and architectural styles (Section 7.2). Subsequently, we describe the problem descriptions that form the input to our method in Section 7.3. In Section 7.4, we introduce the UML profile for architectural descriptions that we have developed and which provides the notational elements for the architectures we derive. In Section 7.5, we describe our method in detail. Not only do we give guidance on how to perform the three steps, but we also give detailed validation conditions that help to detect errors as early as possible. As a running example, we apply our method to derive a software architecture for an automated teller machine. In Section 7.6, we describe the tool that supports developers in applying the method. Section 7.7 discusses related work, and in Section 7.8, we give a summary of our achievements and point out directions for future work.

7.2 Basic Concepts

Our work makes use of problem frames to analyse software development problems and architectural styles to express software architectures. These two concepts are briefly described in the following.

7.2.1 Problem Frames

Problem frames are a means to describe software development problems. They were proposed by Michael Jackson [20], 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 requirement."* Problem frames are described by *frame diagrams*, which basically consist of rectangles, a dashed oval, and different links between them, see Figure 7.1. The task is to construct a *machine* that establishes the desired behaviour of the environment (in which it is integrated) in accordance with the requirements.

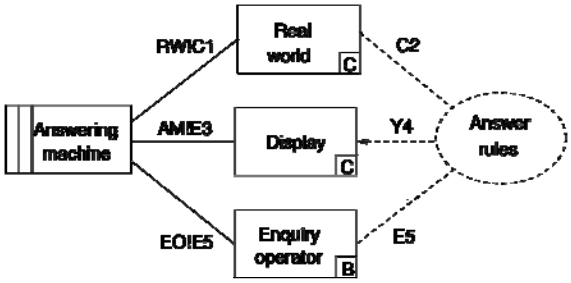

Fig. 7.1**.** *Commanded Information* problem frame

Plain rectangles denote *domains* that already exist in the application environment. Jackson [20, p. 83f] considers three main domain types:

- "A **biddable domain** usually consists of people. The most important characteristic of a biddable domain is that it is physical but lacks positive predictable internal causality. That is, in most situations it is impossible to compel a person to initiate an event: the most that can be done is to issue instructions to be followed."
  Biddable domains are indicated by B (e.g., Enquiry operator in Figure 7.1).
- "A **causal domain** is one whose properties include predictable causal relationships among its causal phenomena."
  Often, causal domains are mechanical or electrical equipment. They are indicated with a C in frame diagrams. (e.g., Display in Figure 7.1). Their actions and reactions are predictable. Thus, they can be controlled by other domains.
- "A **lexical domain** is a physical representation of data – that is, of symbolic phenomena. It combines causal and symbolic phenomena in a special way. The causal properties allow the data to be written and read." Lexical domains are indicated by X.

A rectangle with a double vertical stripe denotes the machine to be developed, and *requirements* are denoted with a dashed oval. The connecting lines between domains represent interfaces that consist of *shared phenomena*. Shared phenomena may be events, operation calls, messages, and the like. They are observable by at least two domains, but controlled by only one domain, as indicated by an exclamation mark. For example, in Figure 7.1 the notation EO! E5 means that the phenomena in the set E5 are *controlled* by the domain Enquiry operator and *observed* by the Answering machine.
To describe the problem context, a connection domain between two other domains may be necessary. Connection domains establish a connection between other domains by means of technical devices. Connection domains are, e.g., video cameras, sensors, or networks.
A dashed line represents a requirement reference, and an arrow indicates that the

requirement *constrains* a domain.[1] If a domain is constrained by the requirement, we must develop a machine, which controls this domain accordingly. In Figure 7.1, the Display domain is constrained, because the Answering machine changes it on behalf of Enquiry operator commands to satisfy the required Answer rules.

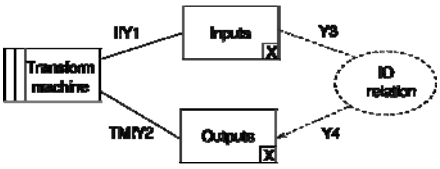

Fig. 7.2 *Transformation* problem frame

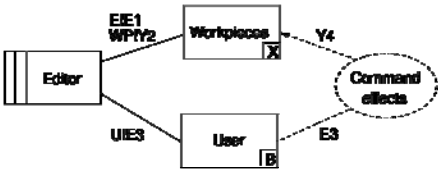

Fig. 7.3. *Simple Workpieces* problem frame

The *Commanded Information* frame in Figure 7.1 is a variant of the *Information Display* frame where there is no operator, and information about the states and behaviour of some parts of the physical world is continuously needed. We present in Figure 7.4 the *Commanded Behaviour* frame in UML notation. That frame addresses the issue of controlling the behaviour of the controlled domain according to the commands of the operator. The *Required Behaviour* frame is similar but without an operator; the control of the behaviour has to be achieved in accordance with some rules. Other basic problem frames are the *Transformation* frame in Figure 7.2 that addresses the production of required outputs from some inputs, and the *Simple Workpieces* frame in Figure 7.3 that corresponds to tools for creating and editing of computer processable text, graphic objects etc.
Software development with problem frames proceeds as follows: first, the environment in which the machine will operate is represented by a *context diagram*. Like a frame diagram, a context diagram consists of domains and interfaces. However, a context diagram contains no requirements. Then, the problem is decomposed into subproblems. Whenever possible, the decomposition is done in such a way that the subproblems fit to given problem frames. To fit a subproblem to a problem frame, one must instantiate its frame diagram, i.e., provide instances for its domains, interfaces, and requirement. The instantiated frame diagram is called a *problem diagram*.

Besides problem frames, there are other elaborate methods to perform requirements engineering, such as i* [31], Tropos [7], and KAOS [6]. These methods are goal-oriented. Each requirement is elaborated by setting up a goal structure. Such a goal structure refines the goal into subgoals and assigns responsibilities to actors for achieving the goal. We have chosen problem frames and not one of the goal-oriented requirements engineering methods to derive architectures, because the elements of problem frames, namely domains, may be mapped to components of an architecture in a fairly straightforward way.

---

[1] In the following, since we use UML tools to draw problem frame diagrams (see Figure 7.4), all requirement references will be represented by dashed lines with arrows and stereotypes ‹‹refersTo››, or ‹‹constrains›› when it is constraining reference.

7.2.2 Architectural Styles

According to Bass, Clements, and Kazman [5],

> 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.

Architectural styles are patterns for software architectures. A style is characterised by [5]:

- a set of component types (e.g., data repository, process, procedure) that perform some function at run-time,
- a topological layout of these components indicating their run-time interrelationships,
- a set of semantic constraints (for example, a data repository is not allowed to change the values stored in it),
- a set of connectors (e.g., subroutine call, remote procedure call, data streams, sockets) that mediate communication, coordination, or cooperation among components.

When choosing a software architecture, usually several architectural styles are possible, which means that all of them could be used to implement the functional requirements. In the following, we will mostly use the *Layered* architectural style for the top-level architecture. The components in this layered architecture are either *Communicating Processes* (active components) or used with a *Call-and-Return* mechanism (passive components)[2]. We use UML 2 composite structure diagrams to represent architectural patterns as well as concrete architectures.

7.3 Problem Description

To support problem analysis according to Jackson [20] with UML [29], we created a new UML profile. In this profile stereotypes are defined. A stereotype extends a UML meta-class from the UML meta-model, such as Association or Class [28].
In the following subsections, we describe our extensions to the problem analysis approach of Jackson (Section 7.3.1), we explain how the different diagrams can be created with UML and our profile (Section 7.3.2), we describe our approach to express connections between domains (Section 7.3.3), and we enumerate the documents that form the starting point for our architectural design method in Section 7.3.4. We illustrate these concepts on an ATM example in Section 7.3.5.

---

[2] The mentioned architectural styles are described in [25].

7.3.1 Extensions

In contrast to Jackson, we allow more than one machine domain in a problem diagram so that we can model distributed systems. In addition to Jackson's diagrams, we express technical knowledge (that we know or can acquire before we proceed to the design phases) about the machine to be built and its environment in a ***technical context diagram*** [19]. In a technical context diagram we introduce connection domains describing the direct technical environment of the machine, e.g., the platform, the operating system or mail server. Additionally, we annotate the technical realisation of all connections as described in Section 7.3.3. With UML it is possible to express aggregation and composition relations between classes and to use multiplicities. Thus we can express that one domain is part of another domain, e.g., that a lexical domain is part of the machine. UML distinguishes between active and passive classes. Active classes can initiate an action without being triggered before. Passive classes just react to some external trigger. Since domains are modelled as classes, they now can also be active or passive. Biddable domains are always active, and lexical domains are usually passive.

7.3.2 Diagram Types

The different diagram types make use of the same basic notational elements. As a result, it is necessary to explicitly state the type of diagram by appropriate stereotypes. In our case, the stereotypes are ‹‹ContextDiagram››, ‹‹ProblemDiagram››, ‹‹ProblemFrame››, and ‹‹TechnicalContextDiagram››. These stereotypes extend (some of them indirectly) the meta-class Package in the UML meta-model.
According to the UML superstructure specification [29], it is not possible that one UML element is part of several packages. For example a class Customer should be in the context diagram package and also in some problem diagrams packages.[3] Nevertheless, several UML tools allow one to put the same UML element into several packages within graphical representations. We want to make use of this information from graphical representations and add it to the model (using stereotypes of the profile). Thus, we have to relate the elements inside a package explicitly to the package. This can be achieved with a dependency stereotype ‹‹isPart›› from the package to all included elements (e.g., classes, interfaces, comments, dependencies, associations).
The ***context diagram*** (see e.g., Figure 7.8) contains the machine domain(s), the relevant domains in the environment, and the interfaces between them. Domains are represented by classes with the stereotype ‹‹Domain››, and the machine is marked by the stereotype ‹‹Machine››. Instead of ‹‹Domain››, more specific stereotypes such as ‹‹BiddableDomain››, ‹‹LexicalDomain›› or ‹‹CausalDomain››

[3] Alternatively, we could create several Customer classes, but these would have to have different names.

can be used. Since some of the domain types are not disjoint, more than one stereotype can be applied on one class.

In a ***problem diagram*** (see e.g., Figure 7.9), the knowledge about a sub-problem described by a set of requirements is represented. A problem diagram consists of sub-machines of the machines given in the context diagram, the relevant domains, the connections between these domains and a requirement (possibly composed of several related requirements), as well as of the relation between the requirement and the involved domains. A requirement refers to some domains and constrains at least one domain. This is expressed using the stereotypes ‹‹refersTo›› and ‹‹constrains››. They extend the UML meta-class Dependency. Domain knowledge and requirements are special statements. Furthermore, any domain knowledge is either a fact (e.g., physical law) or an assumption (usually about a user's behaviour).

The ***problem frames*** (patterns for problem diagrams) have the same kind of elements as problem diagrams. To instantiate a problem frame, its domains, requirement and connections have to be replaced by concrete ones. Figure 7.4 shows the commanded behaviour problem frame in UML notation, using our profile.

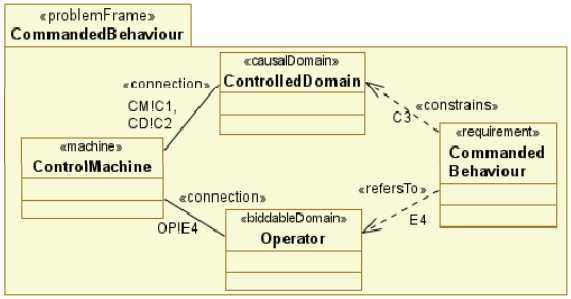

Fig. 7.4. *Commanded Behaviour* problem frame

7.3.3 Associations and Interfaces

For phenomena between domains, we want to keep the notation introduced by Jackson. Our experience is that this notation is easy to read and easy to maintain. In Jackson's diagrams, interfaces between domains (represented as classes) show that there is at least one phenomenon shared by the connected classes. In UML, associations describe that there is some relation between two classes. We decided to use associations to describe the interfaces in Jackson's diagrams. An example for such an interface is depicted in Figure 7.5. The association AD! {showLog} has the stereotype ‹‹connection›› to indicate that there are shared phenomena between the associated domains. The AdminDisplay controls the phenomenon showLog. In general, the name of the association contains the phenomena and the controlling

domain. We represent different sets of shared phenomena with a different control direction between two domains by a second interface class.

Fig. 7.5. Interface class generation – drawn

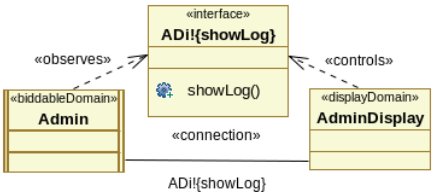

Fig. 7.6. Interface class generation –transformed

Jackson's phenomena can be represented as operations in UML interface classes. The interface classes support the transition from problem analysis to problem solution. Some of the interface classes in problem diagrams become external interfaces of the architecture. In case of lexical domains, they may also be internal interfaces of the architecture. A ‹‹connection›› can be transformed into an interface class controlled by a domain and observed by other domains. To this end, the stereotypes ‹‹observes›› and ‹‹controls›› are defined to extend the meta-class Dependency in the UML meta-model. The interface should contain all phenomena as operations. We use the name of the association as name for the interface class. Figure 7.6 illustrates how the connection given in Figure 7.5 can be transformed into such an interface class.

To support a systematic architectural design, more specific connection types can be annotated in problem descriptions. Examples of such stereotypes which can be used instead of ‹‹connection›› are, e.g., ‹‹network_connection›› for network connections, ‹‹physical›› or ‹‹electrical›› for physical connections, and ‹‹ui›› for user interfaces (see e.g., Figure 7.8). Our physical connection can be specialised into hydraulic flow or hot air flow. These flow types are defined in SysML [26]. For the control signal flow type in SysML, depending on the desired realisation, the stereotypes ‹‹network_connection››, ‹‹event››, ‹‹call_return››, or ‹‹stream›› can be used. Figure 7.7 shows a hierarchy of stereotypes for connections. This hierarchy can be easily extended by new stereotypes.

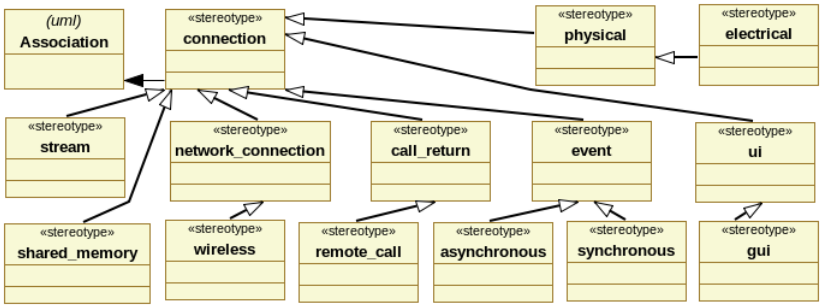

Fig. 7.7. Connection Stereotypes

For these stereotypes, more specialised stereotypes (not shown in Figure 7.7) can

be defined that consider the technical realisation, e.g. events (indicated with the stereotype ‹‹event››) can be implemented using Windows Message Queues (‹‹wmq››), Java Events (‹‹java_events››), or by a number of other techniques. Network connections (‹‹network_connection››) can be realised, e.g., by HTTP (‹‹http››) or the low-level networking protocol TCP (‹‹tcp››).

7.3.4 Inputs and Prerequisites for Architectural Design

As a prerequisite, our approach needs a coherent set of requirements. The architectural design starts after the specification is derived and all *frame concerns* [20] have been addressed. To derive software architectures, we use the following diagrams from requirements analysis:

- context diagram,
- problem diagrams, and
- technical context diagram

Moreover, it may be necessary to know any restrictions that apply concerning the interaction of the machines with their environment. For example, in the automatic teller machine, the user must first authenticate before s/he may enter a request to withdraw a certain amount of money. In the following, we refer to this information as *interaction restrictions*.

7.3.5 The Automatic Teller Machine (ATM)

As a running example, we consider an automatic teller machine (ATM) . Its context diagram – which is identical to the technical context diagram[4] – is shown in Figure 7.8. According to this diagram, Customers can interact with the ATM in the following way:

- withdraw money by inserting their banking card into the CardReader (insert_card),
- enter their PIN (enter_PIN),
- enter a request to withdraw a certain amount of money (enter_request),
- remove their card from the CardReader, and
- take money from the MoneySupply_Case.

The ATM context diagram in Figure 7.8 contains the ATM as the machine to be built. In the ATM environment, we can find the Admin responsible for checking the logs of the ATM with the phenomenon request_log and for filling the MoneySupply_Case with money (phenomenon insert_money).

---

[4] The technical context diagram is identical to the context diagram, because it is not necessary to describe new connection domains representing the platform or the operating system.

In some cases, it is possible that the ATM refuses a withdrawal (refuse_withdrawal). Each ATM is connected with the AccountData of at least one bank. We use multiplicities to express this aspect.
The different domains are annotated with appropriate specialised ‹‹domain›› stereotype, e.g., the Customer is biddable and the AccountData is lexical. The connections are marked with specialisations of the stereotype ‹‹connection››, e.g., a user interface (‹‹ui››) between Customer and ATM, and a physical connection (‹‹physical››) between Customer and CardReader.

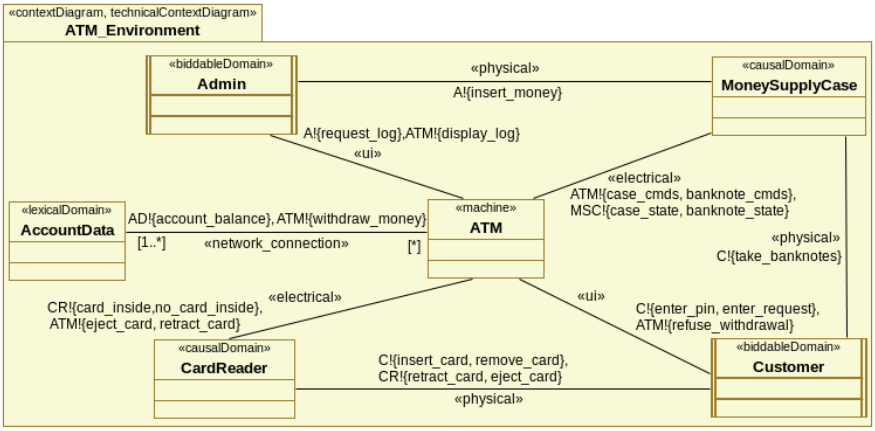

Fig. 7.8. The ATM context diagram / technical context diagram

The card reader controller subproblem in Figure 7.9 is an instance of a variant of Commanded Behaviour (see Figure 7.4). In this variant, we introduce a physical connection between the Customer and the CardReader that models the fact that the customer can physically insert a card into the card reader. Although the phenomena of that interface are used by the CardReader to inform the CardReaderController whether there is a card inside the card reader, they have no interface with the machine.

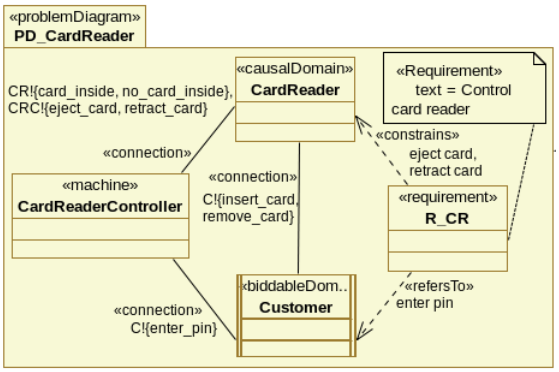

Fig. 7.9. Problem diagram for the card reader controller in UML notation

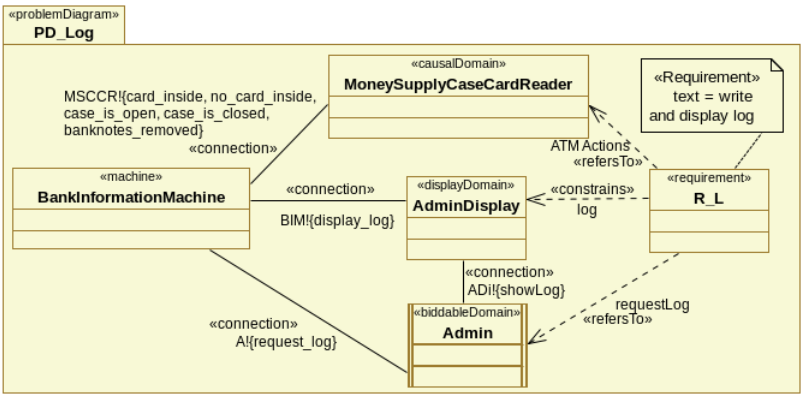

Fig. 7.10. Problem diagram for the administrator log check

A subproblem problem diagram is given in Figure 7.10. It concerns the BankInformationMachine and is an instance of a variant of the commanded information frame. (see Figure 7.1).
The interfaces in context diagram are refined and split to obtain the interfaces in the problem diagrams. For example, MSC! {case_state, banknote_state} is refined into MSC! {case_is_open, case_is_closed, banknotes_removed}. Connection domains, e.g. a AdminDisplay are introduced. Additionally, domains are combined or split. For example, MoneySupplyCaseCardReader (MSCCR) combines MoneySupplyCase (MSC) and CardReader (CR).

7.4 Architectural Description

For each machine in the context diagram, we design an architecture that is described using composite structure diagrams [29]. In such a diagram, the

components with their ports and the connectors between the ports are given. The components are another representation of UML classes. The ports are typed by a class that uses and realises interfaces. An example is depicted in Figure 7.12. The ports (with this class as their type) provide the implemented interfaces (depicted as lollipops) and require the used interfaces (depicted as sockets), see Figure 7.11.
In our UML profile we introduced stereotypes to indicate which classes are components. The stereotype ‹‹Component›› extends the UML meta-class Class. For re-used components we use the stereotype ‹‹ReusedComponent››, which is a specialisation of the stereotype ‹‹Component››. Reused components may also be used in other projects. This fact must be recorded in case such a component is changed. A machine domain may represent completely different things. It can either be a distributed system (e.g., a network consisting of several computers), a local system (e.g., a single computer), a process running on a certain platform, or just a single task within a process (e.g., a clock as part of a graphical user interface). The kind of the machine can be annotated with the stereotypes ‹‹distributed››, ‹‹local››, ‹‹process››, or ‹‹task›››. They all extend the UML meta-class Class.
For the architectural connectors, we allow the same stereotypes as for associations, e.g. ‹‹ui›› or ‹‹tcp››, described in Section 7.3.2 . However, these stereotypes extend the UML meta-class Connector (instead of the meta-class Association).

7.5 Deriving Architectures from Problem Descriptions

We now present our method to derive software architectures from problem descriptions in detail. For each of its three steps, we specify the input that is needed, the output that is produced, and a procedure that can be followed to produce the output from the input. Of course, these procedures are not automatic, and a number of decisions have to be taken by the developer. Such developer decisions introduce the possibility to make errors. To detect such errors as early as possible, each step of the method is equipped with validation conditions. These validation conditions must be fulfilled if the developed documents are semantically coherent. For example, a passive component cannot contain an active component. The validation conditions cannot be complete in a formal sense. Instead, they constitute necessary but not sufficient conditions for the different documents to be semantically valid. New conditions can be defined and integrated in our tool as appropriate.
Our method leads the way from problem descriptions to software architectures in a systematic way, which is furthermore enhanced with quality assurance measures and tool support (see Section 7.6).

7.5.1 Initial Architecture

The purpose of this first step is to collect the necessary information for the architectural design from the requirements analysis phase, to determine which component has to be connected to which external port, to make coordination problems explicit (e.g. several components are connected to the same external domain), and to decide on the machine type and to verify that it is appropriate (considering the connections). At this stage, the submachine components are not yet coordinated.

The *inputs* for this step are the technical context diagram and the problem diagrams. The *output* is an initial architecture, represented by a composite structure diagram. It is set up as follows. There is one *component* for a machine with stereotype ‹‹machine››, and it is equipped with ports corresponding to the interfaces of the machine in the technical context diagram, see Figure 7.11.

Inside this component, there is one component for each submachine identified in the problem diagrams, equipped with ports corresponding to the interfaces in the problem diagrams, and typed with a class. This class has required and provided *interfaces*. A controlled interface in a problem diagram becomes a required interface of the corresponding component in the architecture. Usually, an observed interface of the machine in the problem diagram will become a provided interface of the corresponding component in the architecture. However, if the interface connects a lexical domain, it will be a required interface containing operations with return values (see [15, Section 3.1]). The ports of the components should be connected to the ports of the machine, and stereotypes describing the technical realisation of these connectors are added. A stereotype describing the type of the machine (local, distributed, process, task) is added, as well as stereotypes ‹‹ReusedComponent›› or ‹‹Component›› to all components. If appropriate, stereotypes describing the type of the components (local, distributed, process, task) are also added.

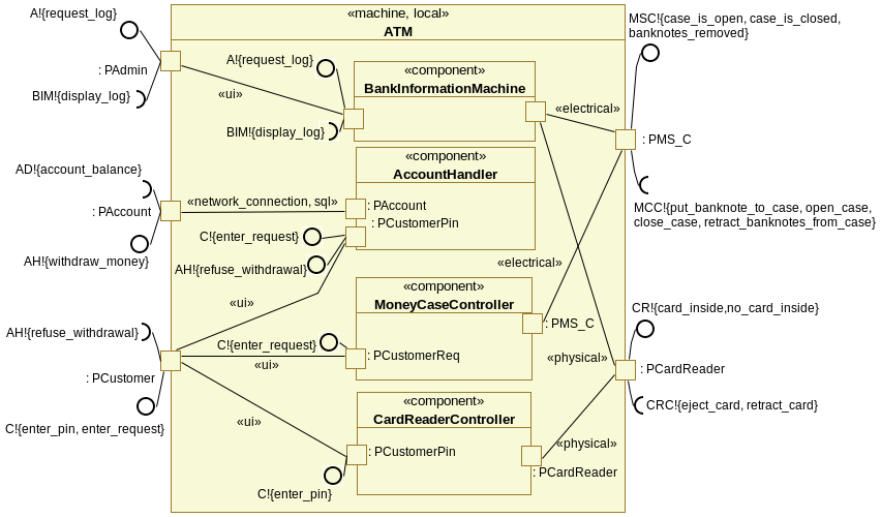

Fig. 7.11. The ATM initial architecture

The initial architecture of the ATM is given in Figure 7.11. Starting from the technical context diagram in Figure 7.8, and the problem diagrams (including the ones given in Figures 7.9 and 7.10), the initial ATM architecture has one component, ATM, with stereotype ‹‹machine, local›› and the ports (typed with :PAdmin, :PAccount, :PCustomer, :PMS_C, :PCardReader) that correspond to the interfaces of the machine in the technical context diagram. The components (CardReaderController, BankInformationMachine, MoneyCaseController, and AccountHandler) correspond to the submachines identified for this case study (e.g., CardReaderController in Figure 7.9, and BankInformationMachine in Figure 7.10). Phenomena at the machine interface in the technical context diagram (e.g. CR! {card_inside}, A! {request_log}, BIM! {display_log}) now occur in external interfaces of the machine. Phenomena controlled by the machine are associated with provided interfaces (e.g. BIM! {display_log}), and phenomena controlled otherwise (e.g. by the user), are associated with required interfaces (e.g., A! {request_log}).

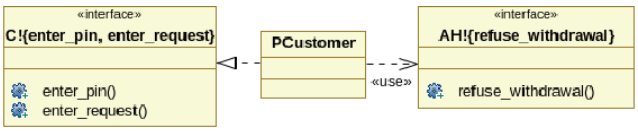

Fig. 7.12. Port Type of PCustomer

Note that connections in the technical context diagram in Figure 7.8 not related to the ATM (such as the one between Admin and MoneySupply_Case) are not reflected in this architecture.

The ports have a class as a type. This class uses and realises interfaces. For example, as depicted in Figure 7.12, the class PCustomer uses the interface AH! {refuse_withdrawal} and realises the class C! {enter_pin, enter_request}. The ports with

this class as a type provide the interface C! {enter_pin, enter_request} (depicted as a lollipop) and requires the interface HL! {refuse_withdrawal} (depicted as a socket).
We have defined two sets of validation conditions for this first phase of our method. The first set is common to all architectures (and hence should be checked after each step of our method), whereas the second one is specialised for the initial architecture. We give a selection of the validations conditions in the following. The complete sets can be found in [11].

Validation conditions for All architectures:

VA.1 Each machine in all problem diagrams must be a component or a re-used component in the architectural description.
VA.2 All components must be contained in a machine or another component.
VA.3 For each operation in a *required* interface of a port of a component, there exists a connector to a port *providing* an interface with this operation, or it is connected to a re-used component.
VA.4 The components' interfaces must fit to the connected interfaces of the machine, i.e., each operation in a required or provided interface of a component port must correspond to an operation in a required or provided interface of a connected machine port.
VA.5 Passive components cannot contain any active components.
VA.6 A class with the stereotype ‹‹Task›› cannot contain classes with the stereotype ‹‹Process››, ‹‹Local››, or ‹‹Distributed››.
A class with the stereotype ‹‹Process›› cannot contain classes with the stereotype ‹‹Local›› or ‹‹Distributed››.
A class with the stereotype ‹‹Local›› cannot contain classes with the stereotype ‹‹Distributed››.

Validation conditions specific to the Initial architecture:

VI.1 For each provided or required interface of machine ports in the architecture, there exists a corresponding interface in the technical context diagram.
VI.2 For each machine in the technical context diagram:
each stereotype name of all associations to the machine (or a specialization of this stereotype) must be included in the set of stereotype names of the connectors from the internal components to external interfaces inside the machine.
VI.3 Each stereotype name of the connectors from components to external interfaces inside an architectural machine component (or their supertypes) must be included in the set of associations to the corresponding machine domain in the technical context diagram.

As already noticed, these validation conditions can be checked automatically, using the tool described in Section 7.6.

7.5.2 Intermediate Architecture

The purpose of this step is to introduce coordination mechanisms between the different submachine components of the initial architecture and its external interfaces, thus obtaining an implementable intermediate architecture. Moreover, we exploit the fact that the subproblems are instances of problem frames by applying architectural patterns that are particularly suited for some of the problem frames. Finally, we decide whether the components should be implemented as active or passive components.

The *input* to this step are the initial architecture, the problem diagrams as instances of problem frames, and a specification of interaction restrictions[5] (see Section 7.3.4). The *output* is an intermediate architecture that is already implementable. It contains coordinator and facade components as well as architectural patterns corresponding to the used problem frames. The intermediate architecture is annotated with the stereotype ‹‹intermediate_architecture›› to distinguish it from the final architecture.

The intermediate architecture is set up as follows. When several internal components are connected to one external interface in the initial architecture, a *facade* component[6] is added. That component has one provided interface containing all operations of some external port and several used interfaces as provided by the submachine components. In our ATM example, several components are connected with external interface :PCustomer in Figure 7.11; therefore a CustomerFacade component is added in Figure 7.13.

If interaction restrictions have to be taken into account, we need a component to enforce these restrictions. We call such a component an a *coordinator* component. A coordinator component has one provided interface containing all operations of some external port and one required interface containing all operations of some internal port. To ensure the interaction restrictions, a state machine can be used inside the component. Typically, coordinator components are needed for interfaces connected to biddable domains (also via connection domains). This is because often, a user must do things in a certain order. In our example, a user must first authenticate before being allowed to enter a request to withdraw money. Therefore, we introduce a CustomerCoordinator in Figure 7.13. Moreover, we need a MSC_Coordinator component, because money should only be put into the money supply case after the user has taken his or her card from the card reader.

---

5 Our method does not rely on how these restrictions are represented. Possible representations are sequence diagrams, state machines, or grammars.

6 See the corresponding design pattern by Gamma et al. [16]: “Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystems easier to use.”

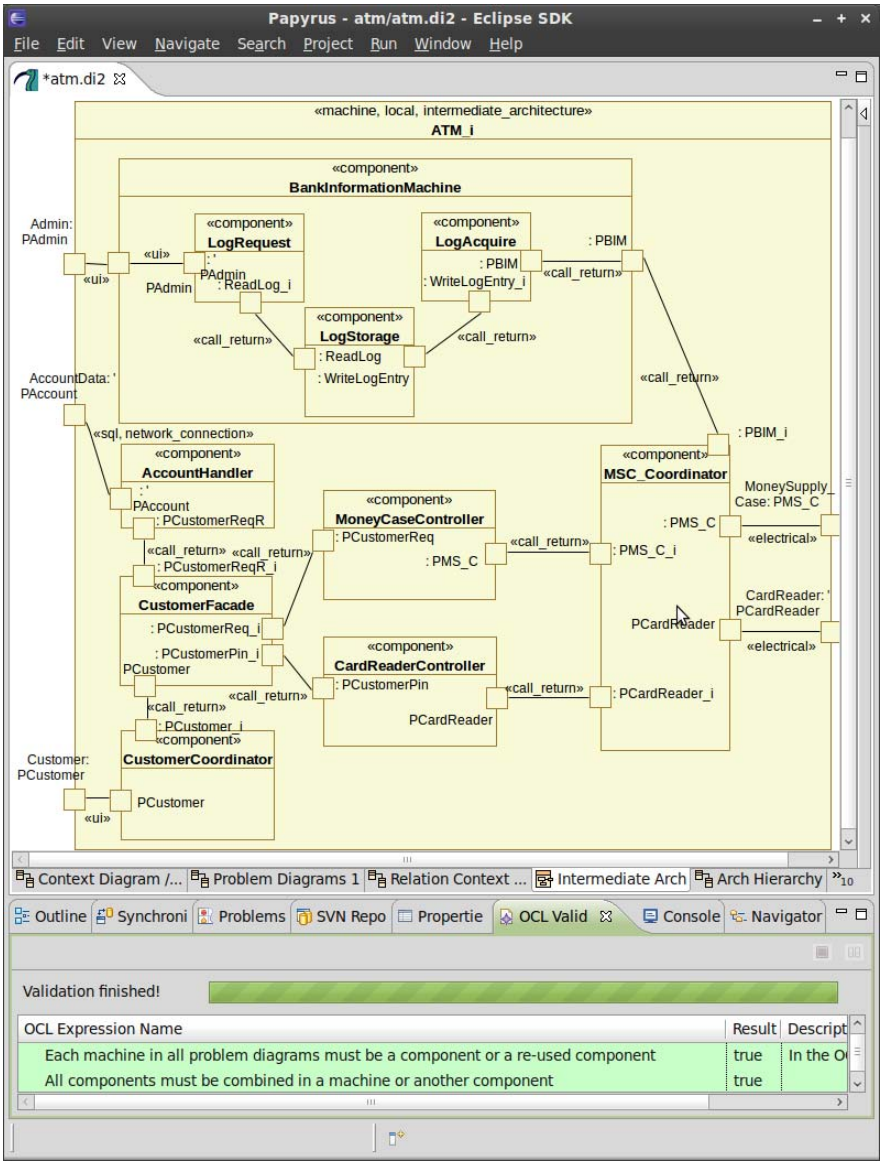

Fig. 7.13. Screenshot of the ATM intermediate architecture

Figure 7.13 also contains a sub-architecture for the component BankInformationMachine. This sub-architecture is an instance of the architectural pattern associated with the *commanded information* problem frame. This pattern contains components that are associated with the acquisition, the storage, and the request for information.

Figure 7.14 shows the architectural pattern for transformation problems. It is a

pipe-and-filter architecture. The architectural pattern for the *required behaviour* frame (not shown here) requires the machine to be an active component.

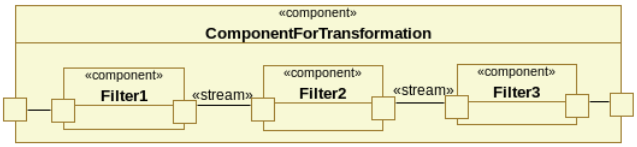

Fig. 7.14. Pattern for component realising transformation

After adding facade and coordinator components and applying architectural patterns related to problem frames, we have to decide for each component if it has to be active or not. In the case of the ATM, all components are reactive (even if the CardReaderController and the MoneyCaseController use timers for timeouts). For new connectors, their technical realisation should be added as stereotypes. For the ATM example, we use the stereotype ‹‹call_return››. Finally, for all newly introduced components it has to be specified if they are a ‹‹Component›› or a ‹‹ReusedComponent››. In Figure 7.13, we have no re-used components.
To validate the intermediate architecture, we have to check (among others) the following conditions (in addition to the conditions to given in Section 7.5.1).
Validation conditions for the interMediate architecture:

VM.1 All components of the initial architecture must be contained in the intermediate architecture.
VM.2 The connectors connected to the ports in the intermediate architecture must have the same stereotypes or more specific ones than in the initial architecture.
VM.3 The stereotypes ‹‹physical›› and ‹‹ui››, and their subtypes are not allowed between components.

7.5.3 Layered Architecture

In this step, we finalise the software architecture. We make sure to handle the external connections appropriately. For example, for a connection marked ‹‹gui››, we need a component handling the input from the user. For ‹‹physical›› connections, we introduce appropriate driver components, which are often re-used. We arrange the components in three layers. The highest layer is the *application layer*. It implements the core functionality of the software, and its interfaces mostly correspond to high-level phenomena, as they are used in the context diagram. The lowest layer establishes the connection of the software to the outside world. It consists of user interface components and *hardware abstraction layer* (HAL) components, i.e., the driver components establishing the connections to hardware components. The low-level interfaces can mostly be obtained from the technical context diagram. The middle layer consists of *adapter* components that

translate low-level signals from the hardware drivers to high-level signals of the application components and vice versa. If the machine sends signals to some hardware, then these signals are contained in a required interface of the application component, connected to an adapter component. If the machine receives signals from some hardware, then these signals are contained in a provided interface of the application component, connected to an adapter component.

The *input* to this step are the intermediate architecture, the context diagram, the technical context diagram, and the interaction restrictions. The *output* is a layered architecture. It is annotated with the stereotype ‹‹layered_architecture›› to distinguish it from the intermediate architecture. Note, however, that a layered architecture can only be defined for a machine or component with the stereotype ‹‹local›, ‹‹process›› or ‹‹task››. For a distributed machine, a layered architecture will be defined for each local component.

To obtain the layered architecture, we assign all components from the intermediate architecture to one of the layers. The submachine components as well as the facade components will belong to the application layer. Coordinator components for biddable domains should be part of the corresponding (usually: user) interface component, whereas coordinator components for physical connections belong the the application layer. As already mentioned, connection stereotypes guide the introduction of new components, namely user interface and driver components. All components interfaces must be defined, where guidance is provided by the context diagram (application layer) and the technical context diagram (external interfaces).

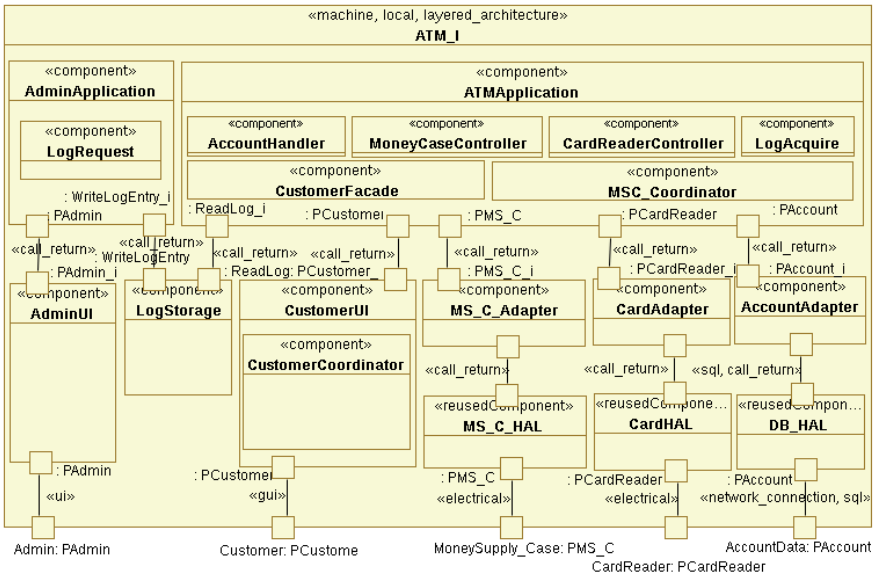

Fig. 7.15. The ATM layered architecture

The final software architecture of the ATM is given in Figure 7.15. Note that we

have two independent application components, one for the administrator and the other handling the interaction with the customers. This is possible, because there are no interaction restrictions between the corresponding subproblems. However, both applications need to access the log storage. Therefore, the component LogStorage does not belong to one of the application components. Each of the biddable domains Admin and Customer is equipped with a corresponding user interface. For the physical connections to the card reader and the money supply case, corresponding HAL and adapter components are introduced. Because the connection to the account data was defined to be a ‹‹network_connection›› already in the initial architecture, the final architecture contains a DB_HAL component.
The validation conditions to be checked for the layered architecture are similar to the validation conditions for the intermediate architectures. Conditions VM.3 must also hold for the layered architecture, and conditions VM.1 and VM.2 become

VL.1 All components of the intermediate architecture must be contained in the layered architecture.
VL.2 The connectors connected to the ports in the layered architecture must have the same stereotypes or more specific ones than in the intermediate architecture.

This final step could be carried out in a different way – resulting in a different final architecture – for other types of systems, e.g., when domain-specific languages are used.

7.6 Tool Support

The basis of our tool called *UML4PF* [30] is the Eclipse platform [1] together with its plug-ins EMF [2] and OCL [27]. Our UML-profiles described in Sections 7.3 and 7.4 are conceived as an eclipse plug-in, extending the EMF meta-model. We store all our OCL constraints (which formalise the validation conditions given in Section 7.5) in one file in XML-format. With these constraints, we check the validity and consistency of the different models we set up during the requirements analysis and architectural design phases. An overview of the context of our tool is provided in Figure 7.16. Gray boxes denote re-used components, whereas white boxes describe those components that we created.

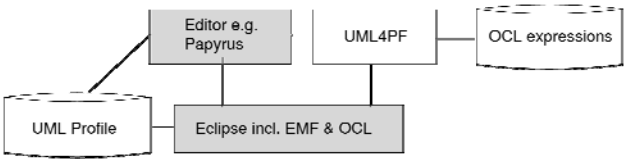

Fig. 7.16. Tool Realisation Overview

The functionality of our tool UML4PF comprises the following:

- It checks if the developed models are valid and consistent by using our OCL constraints.
- It returns the location of invalid parts of the model.
- It automatically generates model elements, e.g., it generates observed and controlled interfaces from association names as well as dependencies with stereotype ‹‹isPart›› for all domains and statements being inside a package in the graphical representation of the model.

The graphical representation of the different diagram types can be manipulated by using any EMF-based editor. We selected Papyrus UML [3], because it is available as an Eclipse plug-in, open-source, and EMF-based. Papyrus stores the model (containing requirements models and architectures) with references to the UML-profiles in one XML-file using EMF. The XML format created by EMF allows developers to exchange models between several UML tools. The graphical representation of the model is stored in separate file. Since UML4PF is based on EMF, it inherits all strengths and limitations of this platform. To use Papyrus with UML4PF to develop an architecture, developers have to draw the context diagram and the problem diagrams (see Section 7.3). Then they can proceed with deriving the specification, after UML4PF has generated the necessary model elements. Next, the requirements models are automatically checked with UML4PF.

Re-using model elements from the requirements models, developers create the architectures as described in Section 7.5. After each step, the model can be automatically validated. UML4PF indicates which model elements are not used correctly or which parts of the model are not consistent. Figure 7.13 shows a screenshot of UML4PF. As can be seen below the architectural diagram, several kinds of diagrams are available for display. When selecting the OCL validator, the validation conditions are checked, and the results are displayed as shown at the bottom of the figure. Fulfilled validation conditions are displayed in green, violated ones in red.

All in all, we have defined about 80 OCL validation conditions, including 17 conditions concerning architectural descriptions. The time needed for checking only depends on EMF and is about half a second per validation condition. The influence of the model size on the checking time is less than linear. About 9800 lines of code have been written to implement UML4PF.

The tool UML4PF is still under development and evaluation. Currently it is used in a software engineering class at the University Duisburg-Essen with about 100 participants. In this class, the problem frame approach and the method for architectural design described in this chapter are taught and applied to the development of a web application. The experience gained from the class will be used to assess (and possibly improve) the user-friendliness of the tool.

Moreover, UML4PF will be integrated into the tool WorkBench of the European network of excellence NESSoS (see http://www.nessos-project.eu/). With this integration, we will reach a wider audience in the future. Finally, the tool is available for download at http://swe.uni-due.de/en/research/tool/index.php.

7.7 Related Work

Since our approach heavily relies on the use of patterns, our work is related to research on problem frames and architectural styles. However, we are not aware of similar methods that provide such a detailed guidance for developing software architectures, together with the associated validation conditions.

Lencastre et al. [23] define a meta-model for problem frames using UML. Their meta-model considers Jackson's whole software development approach based on context diagrams, problem frames, and problem decomposition. In contrast to our meta-model, it only consists of a UML class model without OCL integrity constraints. Moreover, their approach does not qualify for a meta-model in terms of MDA because, e.g., the class Domain has subclasses Biddable and Given, but an object cannot belong to two classes at the same time (cf. Figures 5 and 11 in [23]).

Hall et al. [17] provide a formal semantics for the problem frame approach. They introduce a formal specification language to describe problem frames and problem diagrams. However, their approach does not consider integrity conditions.

Seater et al. [24] present a meta-model for problem frame instances. In addition to the diagram elements formalised in our meta-model, they formalise requirements and specifications. Consequently, their integrity conditions ("wellformedness predicate") focus on correctly deriving specifications from requirements. In contrast, our meta-model concentrates on the structure of problem frames and the different domain and phenomena types.

Colombo et al. [14] model problem frames and problem diagrams with SysML [26]. They state that "*UML is too oriented to software design; it does not support a seamless representation of characteristics of the real world like time, phenomena sharing [...]*". We do not agree with this statement. So far, we have been able to model all necessary means of the requirements engineering process using UML.

Charfi et al. [8] use a modelling framework, *Gaspard2*, to design high-performance embedded systems-on-chip. They use model transformations to move from one level of abstraction to the next. To validate that their transformations were performed correctly, they use the OCL language to specify the properties that must be checked in order to be considered as correct with respect to Gaspard2. We have been inspired by this approach. However, we do not focus on high-performance embedded systems-on-chip. Instead, we target general software development challenges.

Choppy and Heisel give heuristics for the transition from problem frames to architectural styles. In [12], they give criteria for choosing between architectural styles that could be associated with a given problem frame. In [13], a proposal for the development of information systems is given using *update* and *query* problem frames. A component-based architecture reflecting the repository architectural style is used for the design and integration of the different system parts. In [9], the authors of this paper propose architectural patterns for each basic problem frameproposed by Jackson [20]. In a follow-up paper [10], the authors show how to merge the different sub-architectures obtained according to the patterns presented in [9], based on the relationship between the subproblems. Hatebur and

Heisel [19] show how interface descriptions for layered architectures can be derived from problem descriptions.
Barroca et al. [4] extend the problem frame approach with *coordination* concepts. This leads to a description of *coordination interfaces* in terms of *services* and *events* together with required properties, and the use of *coordination rules* to describe the machine behaviour.
Lavazza and Del Bianco [21] also represent problem diagrams in a UML notation. They use component diagrams (and not stereotyped class diagrams) to represent domains. Jacksons interfaces are directly transformed into used/required classes (and not observe and control stereotypes that are translated in the architectural phase). In a later paper, Del Bianco and Lavazza [22] suggest enhance problem frames with scenarios and timing.
Hall, Rapanotti, and Jackson [18] describe a formal approach for transforming requirements into specifications. This specification is then transformed into the detailed specifications of an architecture. We intentionally left out deriving the specification describing the dynamic behaviour within this chapter and focus on the static aspects of the requirements and architecture.

7.8 Conclusion and Perspectives

We have shown how software architectures can be derived in a systematic way from problem descriptions as they are set up during the requirements analysis phase of software development. In particular, our method builds on information that is elicitated when applying (an extension of) the problem frame approach. The method consists of three steps, starting with a simple initial architecture. That architecture is gradually refined, resulting in a final layered architecture. The refinement is guided by patterns and stereotypes. The method is independent of system characteristics – it works e.g., for embedded systems, for web-applications, and for distributed systems as well as for local ones. Its most important advantages are the following:

- The method provides a systematic approach to derive software architectures from problem descriptions. Detailed guidance is given in three concrete steps.
- Validation conditions for each step help to increase the quality of the results. These conditions can be checked automatically.
- The subproblem structure can be exploited for setting up the architecture.
- Most interfaces can be derived from the problem descriptions.
- Only one model is constructed containing all the different development artifacts. Therefore, traceability between the different models is achieved, and changes propagate to all graphical views of the model.
- Frequently used technologies are taken into account by stereotypes. The stereotype hierarchy can be extended for new developments.
- Stereotypes guide the introduction of new components.
- Adapters can be generated automatically (based on stereotypes).

- The application components use high-level phenomena from the application domain. Thus, the application components are independent of the used technology.
- Re-use of components is supported.

The method presented in this chapter can be extended to take other software development artefacts into account. For example, sequence diagrams describing the externally visible behaviour of machine domains can be used to derive behavioural descriptions of the architectural components. In the future, we will extend our approach to support the development of design alternatives according to quality requirements, such as performance or security, and to support software evolution. On the long run, the method can also be extended to cover further phases of the software development lifecycle.

Acknowledgments We would like to thank our anonymous reviewers for their careful reading and constructive comments.

References

[1] Eclipse - An Open Development Platform (2008) May 2008. http://www.eclipse.org/.
[2] Eclipse Modeling Framework Project (EMF) (2008) May 2008. http://www.eclipse.org/modeling/emf/.
[3] Papyrus UML Modelling Tool (2010) Jan 2010. http://www.papyusuml.org/.
[4] L. Barroca, J. L. Fiadeiro, M. Jackson, R. C. Laney, and B. Nuseibeh (2004) Problem frames: A case for coordination. In Coordination Models and Languages, Proc. 6th International Conference COORDINATION, pages 5–19.
[5] L. Bass, P. Clements, and R. Kazman (1998) Software Architecture in Practice. Addison-Wesley, first edition.
[6] P. Bertrand, R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde (1998) GRAIL/KAOS: an environment for goal driven requirements engineering. In ICSE'98 - 20th International Conference on Software Engineering.
[7] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos (2004) Tropos: An agent oriented software development methodology. Autonomous Agents and Multi-Agent Systems, 8(3):203–236.
[8] A. Charfi, A. Gamatié, A. Honoré, J.-L. Dekeyser, and M. Abid (2008) Validation de modèles dans un cadre d'IDM dédié à la conception de systèmes sur puce. In 4èmes Jounées sur l'Ingénierie Dirigée par les Modèles (IDM 08.
[9] C. Choppy, D. Hatebur, and M. Heisel (2005) Architectural patterns for problem frames. IEE Proceedings – Software, Special Issue on Relating Software Requirements and Architectures, 152(4):198–208.
[10] C. Choppy, D. Hatebur, and M. Heisel (2006) Component composition through architectural patterns for problem frames. In Proc. XIII Asia Pacific Software Engineering Conference (APSEC), pages 27–34. IEEE.
[11] C. Choppy, D. Hatebur, and M. Heisel (2010) Systematic architectural design based on problem patterns (technical report). Universität Duisburg-Essen.
[12] C. Choppy and M. Heisel (2003) Use of patterns in formal development: Systematic transition from problems to architectural designs. In Recent Trends in Algebraic Development Techniques, 16th WADT, Selected Papers, LNCS 2755, pages 205–220. Springer Verlag.
[13] C. Choppy and M. Heisel (2004) 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.

[14] P. Colombo, V. del Bianco, and L. Lavazza (2008) Towards the integration of SysML and problem frames. In IWAAPF ’08: Proceedings of the 3rd international workshop on Applications and advances of problem frames, pages 1–8, New York, NY, USA, ACM.
[15] I. Côté, D. Hatebur, M. Heisel, H. Schmidt, and I. Wentzlaff (2008) A systematic account of problem frames. In Proceedings of the European Conference on Pattern Languages of Programs (EuroPLoP), pages 749–767. Universitätsverlag Konstanz.
[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides (1995) Design Patterns – Elements of Reusable Object-Oriented Software. Addison Wesley, Reading.
[17] J. G. Hall, L. Rapanotti, and M. Jackson (2005) Problem frame semantics for software development. Software and System Modeling, 4(2):189–198.
[18] J. G. Hall, L. Rapanotti, and Michael A. Jackson (2008) Problem oriented software engineering: Solving the package router control problem. volume 34, April 2008.
[19] D. Hatebur and M. Heisel (2009) Deriving software architectures from problem descriptions. In Software Engineering 2009 - Workshopband, pages 383–302. GI.
[20] M. Jackson (2001) Problem Frames. Analyzing and structuring software development problems. Addison-Wesley,.
[21] L. Lavazza and V. D. Bianco (2006) Combining Problem Frames and UML in the Description of Software Requirements. Fundamental Approaches to Software Engineering.
[22] L. Lavazza and V. D. Bianco (2008) Enhancing Problem Frames with Scenarios and Histories in UML-based software development. Expert Systems - The Journal of Knowledge Engineering, 25(1).
[23] M. Lencastre, J. Botelho, P. Clericuzzi, and J. Araújo (2005) A meta-model for the problem frames approach. In WiSME’05: 4th Workshop in Software Modeling Engineering.
[24] R. Seater, D. Jackson, and R. Gheyi (2007) Requirement progression in problem frames: deriving specifications from requirements. Requirements Engineering, 12(2):77–102.
[25] M. Shaw and D. Garlan (1996) Software Architecture. Perspectives on an Emerging Discipline. Prentice-Hall.
[26] SysML Partners. (2005) Systems Modeling Language (SysML) Specification. see http://www.sysml.org.
[27] "UML Revision Task Force"(2006) OMG Object Constraint Language: Reference. http://www.omg.org/docs/formal/06-05-01.pdf.
[28] "UML Revision Task Force" (2009) OMG Unified Modeling Language: Infrastructure. available at http://www.omg.org/docs/formal/09-02-04.pdf.
[29] "UML Revision Task Force". (2009) OMG Unified Modeling Language: Superstructure, available at http://www.omg.org/docs/formal/09-02-02.pdf.
[30] UML4PF (2010)
http://swe.uni-due.de/en/research/tool/index.php.
[31] E. Yu. (1997) Towards modelling and reasoning support for early-phase requirements engineering. In Proceedings of the 3rd IEEE Intern. Symposium on RE, pages 226 – 235.