Composing architectures based on architectural patterns for problem frames

Christine Choppy[1], Denis Hatebur[2,3], and Maritta Heisel[2]

[1] LIPN, Institut Galilée - Université Paris XIII, France, email: Christine.Choppy@lipn.univ-paris13.fr
[2] Universität Duisburg-Essen, Fachbereich Ingenieurwissenschaften, Institut für Medientechnik und Software-Engineering, Germany, email: maritta.heisel@uni-duisburg-essen.de
[3] Institut für technische Systeme GmbH, Germany, email: d.hatebur@itesys.de

Abstract. The use of patterns is a promising way of developing high-quality software in a systematic way. 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. In a recent paper, we have defined architectural patterns corresponding to Jackson's problem frames.
To make use of problem frames, complex problems have to be decomposed into simple ones. The corresponding architectural patterns then provide solution structures for these simple problems. Now the question arises how to combine the solutions structures of the simple subproblems to obtain a solution structure for the complex problem. The present paper addresses this question.
Different subproblems of a complex problem can be related in various ways. They can be independent of each other, they can exclude each other, or they may have to be solved in a specific order. Such information can be used to combine the solutions structures of the subproblem to a solution structure of the overall problem.
In this paper, we present a pattern-based software development process using problem frames and the corresponding architectural patterns. In decomposing a complex problem into simple subproblems, the relationships between the subproblems are recorded explicitly. Based on this information, we give guidelines how to derive the software architecture for the overall problem from the software architectures of the simple subproblems.

1 Introduction

Pattern-orientation is a promising approach to software development. Patterns provide structuring concepts that are of invaluable help for problem understanding and system design, and are a means to reuse software development knowledge on different levels of abstraction. They classify sets of software development problems or solutions that share the same structure.

Patterns were introduced on the level of detailed object oriented design [10], and are now defined for different activities. *Problem Frames* [13] are patterns that classify software development *problems*. *Architectural styles* (or "architectural patterns") are patterns that characterize software architectures [19]. Patterns for further development phases include *design patterns*, *frameworks*, and *idioms* or "code patterns". Using patterns, we can hope to construct software in a systematic way, making use of a body of accumulated knowledge, rather than starting from scratch.

It is acknowledged that the first steps of software development are essential to reach the best possible match between the expressed requirements and the proposed software product, and to eliminate any source of error as early as possible. Therefore, we propose to use patterns starting from the requirements elicitation phase of the software development life-cycle, as advocated by Fowler [9] or Sutcliffe et al. [20, 21]. M. Jackson [13] proposes the concept of *problem frames* for presenting, classifying and understanding software development problems. A problem frame is a characterization of a class of problems in terms of their main components and the connections between these components. Once a problem is successfully fitted to a problem frame, its most important characteristics are known.

Gaining a thorough understanding of the problem to be solved is a necessary prerequisite for solving it. However, when using problem frames, one can even hope for more than just a full comprehension of the problem at hand. Since problem frames are patterns, they represent problem structures that occur repeatedly in practice. Hence, it is worthwhile to look for solution structures that match the problem structures represented by problem frames.

The construction of the solution of a software development problem should begin with the decision on the main structure of the solution, i.e., a decision on the software architecture. We exploit the knowledge gained in representing a problem as an instance of a problem frame in taking that decision. In [5], we define architectural patterns corresponding to Jackson's problem frames, taking into account the characteristics of the problems fitting to the given problem frame. The structure provided by an architectural pattern constitutes a concrete starting point for the process of constructing a solution to a problem that is represented as an instance of a problem frame.

Different subproblems of a complex problem can be related in various ways. They can be related sequentially, by alternative or they can be independent (parallel). Such information can be used to combine the solutions structures of the subproblem to a solution structure of the overall problem.

In this paper, we present a pattern-based software development process using problem frames and the corresponding architectural patterns. In decomposing a complex problem into simple subproblems, the relationships between the subproblems are recorded explicitly. Based on this information, we give guidelines on how to derive the software architecture for the overall problem from the software architectures and the component specifications of the simple subproblems.

Throughout this work, we use object-oriented notations, mostly from UML 2.0 [23]. Although our pattern-based software development process does not strictly depend on object-orientation, it works particularly well in an object-oriented setting.

The rest of the paper is organized as follows: after introducing the basic concepts of our work in Section 2, we briefly introduce the architectural patterns we developed for the various problem frames in Section 3. Then, we discuss related work in Section 4. Our pattern-based software development process is presented in Section 5 and illustrated by a case study in Section 6. In Section 7, we conclude with a discussion of our approach and directions for future research.

2 Basic Concepts

The patterns used in our development process are problem frames and architectural patterns. As a notation for our architectural patterns, we use composite structure diagrams of UML 2.0. In the following, we give brief descriptions of these three ingredients of our work. [1]

2.1 Problem Frames

Jackson [13] describes problem frames 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."

Solving a problem is accomplished by constructing a "machine" and integrating it into the environment whose behavior is to be enhanced.

For each problem frame a diagram is set up (see left-hand side of Fig. 1). Plain rectangles denote application domains (that already exist), rectangles with a double vertical stripe denote the machine domains to be developed, and requirements are denoted with a dashed oval. They are linked together by lines that represent interfaces, also called shared phenomena.

The following problems fit to the *Required Behaviour* 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.'

The corresponding frame diagram is shown on the left-hand side of Figure 1. The "`C`" in the frame diagram indicates that the *Controlled domain* must be causal. The machine is always a causal domain (so an explicit "`C`" is not needed). The notation "`CM!C1`" means that the causal phenomena `C1` are controlled by the Control machine `CM`. The dashed line represents a requirements reference, and the arrow shows that it is a *constraining* reference.

This problem frame is appropriate for *embedded systems*, where the machine to be developed is embedded in a physical environment that must be controlled. The communication between the machine and the physical environment takes place via *sensors* and *actuators*. Thus, only by virtue of sensors and actuators can there be shared phenomena between the machine and its environment. Sensors realize the phenomena `C2` of the frame diagram, i.e., the phenomena controlled by the environment but observable

[1] In the following, we will also use sequence diagrams and state machines. However, these notations are well-known and intuitive, and we will not explain them here.

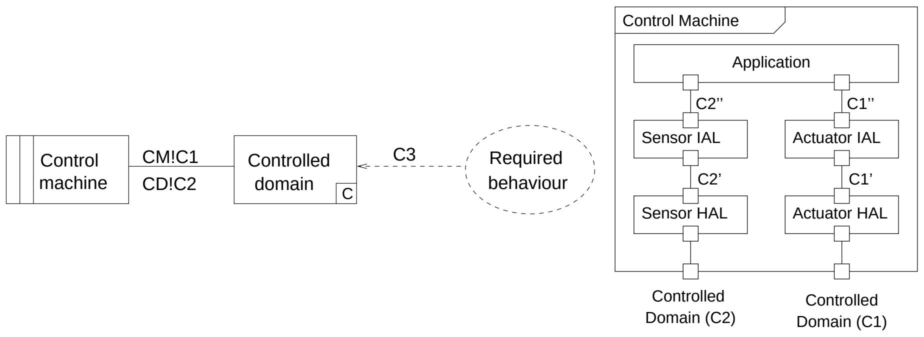


Fig. 1. Required Behaviour Frame Diagram and Architecture

by the machine. Actuators realize the phenomena `C1` of the frame diagram, i.e., the phenomena controlled by the machine and observable by the environment.

For example, we might want to build a machine that keeps the temperature of some liquid between given bounds. Then, the temperature of the liquid would be a shared phenomenon controlled by the environment. The corresponding sensor would be a thermometer. Another shared phenomenon would be the state of a burner. That state would be controlled by the machine, i.e., the machine is able to switch the burner on or off.

Jackson defines five basic problem frames, namely *Required Behaviour, Commanded Behaviour, Information Display, Workpieces* and *Transformation*. In order to use a problem frame, one must instantiate it, i.e., provide instances for its domains, interfaces and requirements.

2.2 Architectural Styles

According to Bass, Clements, and Kazman [2],

> “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 characterized by [2] (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. We use UML 2.0 composite structure diagrams (see Section 2.3) to represent architectural patterns as well as concrete architectures.

2.3 Composite Structure Diagrams

Composite structure diagrams [23] are a means to describe architectures (cf. Fig. 1). They contain named rectangles, called *parts*. Theses parts are components of the software. Each component may contain other (sub-) components. Atomic components can be described by state machines and operations for accessing internal data. Parts may have *ports*, denoted by small rectangles, and ports may have interfaces associated to them. Interfaces may be required or provided. Provided interfaces are denoted using the “lollipop” notation, and required interfaces using the “socket” notation. Figure 2 shows how interfaces in problem diagrams are transformed into interfaces in composite structure diagrams.

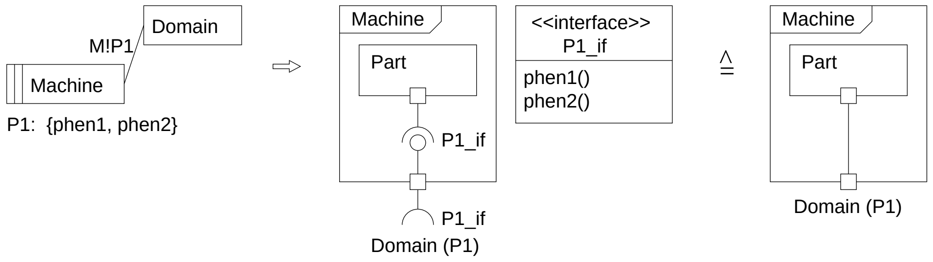


Fig. 2. Notation for Architectures

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. We use the structural view from UML 2.0 that describes the structure of the software at runtime. After that structure is fixed the interfaces need to be refined using sockets, lollipops and interface classes to describe the possible data flow. Then the corresponding active or passive class with its data and operations can be added for each component. Thereby the process-oriented and object-oriented views can be integrated seamlessly into the structural view. That approach and the corresponding process are described in [12].

3 Architectural Patterns for Problem Frames

The architectural patterns we have defined for the different problem frames in [5] take the characteristics of the respective problem frame into account. They are based on a *Layered* architecture, as shown on the right-hand side of Fig. 1.

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 [17]) 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 [3].

Note that the phenomena `C3` do not occur in the architecture[2], because they do not belong to the interface of the machine domain.

Thus, the architecture shown on the right-hand side of Fig. 1 represents an adequate structure for the `Control machine` of the left-hand side of Fig. 1. 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.

Of course, our 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 kind of (layered) architecture we propose has proven useful in practice (see for example [4, 12, 22]), 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. We have validated this kind of architecture in several industrial projects, dealing for example with smart cards, protocol converters, web/mail-servers, and real-time operating systems.

4 Related Work

A number of research activities deal with the use of patterns in the software development process. We consider here mainly those related with the use of problem frames, also in relationship with architectural styles.

Aiming to integrate problem frames in a formal development process, Choppy and Reggio [8] show how a formal specification skeleton may be associated with some problem frames. Choppy and Heisel show in [6, 7] that this idea is independent of concrete specification languages. In that work, they also give heuristics for the transition from problem frames to architectural styles. In [6], they give criteria for (i) helping to select an appropriate basic problem frame, and (ii) choosing between architectural styles that could be associated with a given problem frame.

[2] In the following, we use the word “architecture” instead of “architectural pattern” for reasons of readability. It is clear, however, that the components shown in the architectural diagrams have to be instantiated in order to obtain a concrete software architecture.

In [7], a proposal for the development of information systems is given using update or query problem frames. A component-based architecture reflecting the repository architectural style is used for the design and integration of the different system parts.

The approach developed by Hall, Rapanotti et al. [11, 18] is quite complementary to ours, since the idea developed there is to introduce architectural concepts into problem frames (introducing “AFrames”) so as to benefit from existing architectures. In [11], the applicability of problem frames is extended to include domains with existing architectural support, and to allow both for an annotated machine domain, and for annotations to discharge the frame concern. In [18], “AFrames” are presented corresponding to the architectural styles Pipe-and-Filter and Model-View-Controller (MVC), and applied to transformation and control problems.

Let us also mention Lavazza and Del Bianco [15] who do not use architectures, but provide a description of commanded and required behavior problem frames in UML-RT, focusing on active objects or “capsules” communicating through ports (defined by protocols). Moreover, they provide a real time version of OCL, called OTL.

Barroca et al. [1] extend the problem frame approach with *coordination* concepts. This leads to a description of *coordination interfaces* in terms of *services* and *events* (referred to respectively here as actuators and sensors) together with required properties, and the use of *coordination rules* to describe the machine behavior.

5 Software Development Process

In the following, we describe a pattern-based software development process. That process is based on problem frames [13] and the corresponding architectural patterns that we propose in [5]. We mostly use concrete object-oriented notations (often taken from UML [23]) to express the results of the different steps of the process. In principle, the process could be carried out using other notations, but the procedures we give below on how to execute the steps would have to be adjusted in that case.

The novelty of the process is that the relationships between the subproblems are expressed explicitly, and that these relationships are exploited when generating a global software architecture for the overall problem. Although Jackson [13] gives some hints on how to decompose problems into subproblems, there is no general procedure for constructing the solution of the overall problem from the solutions of the subproblems. The current paper proposes an approach on how to achieve that composition.

Our pattern-based software development process using problem frames and architectural patterns proceeds as follows: first, a context diagram showing the problem context is set up (for an example, see Figure 3). Then, the overall problem is decomposed into subproblems that should fit to existing problem frames. This decomposition can be achieved in various ways, for example by use-case decomposition, or by projection, as proposed by Jackson [13]. The decomposition results in a set of problem diagrams (that should be instantiated frame diagrams whenever possible) and the information how the different subproblems are related, expressed e.g. as a grammar. For each subproblem, a specification for the machine domain must be derived, thus addressing the frame concern. Each machine domain corresponding to a subproblem is then structured by instantiating the architectural patterns we have proposed in [5]. The instantiated patterns must afterwards be merged to obtain the architecture of the machine solving the overall problem. It ts the main contribution of the present paper to show how that composition can be performed in a systematic way, making use of the relations between the subproblems that were expressed during problem decomposition. Finally, the components of the combined architecture must be specified in more detail, and it must be shown that the combined architecture fulfils the specifications of all subproblems.

The process consists of twelve steps that we explain one by one. The steps that are the most important for the task of constructing the overall solution structure from the subproblem solution structures are Steps 3, 9, and 10.

1. Collect requirements and domain knowledge.

   Input An informal description of the task.

   Procedure The requirements (optative statements) have to be expressed, as well as knowledge about the environment in which the machine (i.e. the software system to be developed) has to operate (indicative statements). Whereas the requirements have to be achieved by constructing the machine, the domain knowledge expresses facts that are true no matter how the machine is built. (For a more details, see [24].)

   Output A set R of requirements, and a set D of domain knowledge statements. These can be expressed in natural language, or in semi-formal or formal notations.

   Validation The statements contained in R and D must be non-contradictory.

2. Draw a context diagram.

   Input An informal description of the task.

   Procedure We must identify all domains that are relevant to the problem at hand, and the phenomena that are shared by different domains.

   Output A context diagram containing all relevant domains and shared phenomena. (For a more details, see [13].)

   Validation The results of Steps 1 and 2 must be consistent, i.e., all domains and phenomena mentioned in R and D must be contained in the context diagram, and all domains and phenomena of the context diagram must be related to some element of R or D.

3. Decompose the problem into simple subproblems, and express the relations between the different subproblems. If possible, the subproblems should fit to known problem frames (or variants).

   Input Results of Steps 1 and 2.

   Procedure There are different possibilities to decompose a complex problem into subproblems. Jackson [13] proposes a parallel decomposition using projection, but a decomposition by use-cases (for an example, see [7]) or a top-down decomposition are also possible. Subproblems refer to related sets of requirements, and they should only constrain a single domain (otherwise, the subproblem is not simple but needs further decomposition).

   The following relationships between subproblems are possible: *parallel* subproblems are largely independent of one another, and the composed machine will have to treat the problems in parallel. *Sequential* subproblems have to be treated one after the other. *Alternative* problems are exclusive. Only one of them will have to be treated at a given time.

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 9 and 10). This is the challenge of the composition problem.

Output A set of problem diagrams, being mostly instantiated frame diagrams, and an expression of the subproblem relationships. 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).

Validation All requirements have to be captured, and each requirement must be assigned exactly to one subproblem, otherwise the requirement must be split. The problem diagrams must be consistent with the context diagram of Step 2. The following operations preserve consistency:

- leave out domains (with corresponding interfaces)
- combine several domains into one domain
- divide one domain
- reduce an interface between domains
- refine phenomena
- combine (i.e., abstract) phenomena

4. Derive a specification for each subproblem.

Input Results of Steps 1–3.

Procedure 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 domain knowledge. For more details, see [14].

Output A specification for each subproblem, expressed as a set of sequence diagrams. State invariants should be annotated for the domains in the environment of the machine.

Validation Specification and domain knowledge must be non-contradictory. The specification, together with the domain knowledge, must imply that the requirements are fulfilled. In performing that proof, the frame concern is addressed. The frame concern provides a structure for the correctness proof.

Additionally, the phenomena of the machine domain must be consistent with the signals in the sequence diagrams, i.e., they must have the same name, or a mapping must be created. All phenomena 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 of Step 3 describe.

5. Define an architecture for each subproblem.

Input Problem diagrams resulting from Step 3.

Procedure If a subproblem fits to a known problem frame, then a simple instantiation of the pattern we gave in [5] will suffice. If a subproblem is not an exact instance of a problem frame but a variant, then modifications of our architectural patterns will be necessary. If a subproblem is unrelated to any problem frame, then an appropriate architecture has to be developed from scratch.

Output A subproblem architecture for each subproblem, expressed as a composite structure diagram.

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

6. Specify the interface classes for all interfaces of all subproblem architectures.

   Input Results of Steps 3 and 5.

   Procedure For each interface contained in a subproblem architecture, the corresponding operations or signals, respectively, have to be defined, and provided and required interfaces must be distinguished.

   Output A set of interface classes.

   Validation All interfaces must be covered. The signals or operations in the interfaces classes must be the same as the signals in the sequence diagrams of Step 4.

7. Specify all components of all subproblem architectures.

   Input Results of Steps 4–6.

   Procedure For each component, its external behavior is expressed using sequence diagrams. For the application layer (cf. Fig. 1), it should be possible to re-use the specifications developed in Step 4. To reuse the specifications, the interface phenomena have to be adjusted according to the functionality of the IAL and the HAL. Moreover, to prepare for the next step, the sequence diagrams should be annotated with state invariants, as in Step 4.

   Output A set of sequence diagrams, annotated with state information.

   Validation All components must be covered. The signals in specification must be defined in the interfaces classes. The sequence diagrams for the components together must describe the same behavior as described in Step 4.

8. Define a state machine and the used data for each architectural component.

   Input Result of Step 7.

   Procedure Use the state information contained in the sequence diagrams to construct a state machine specifying the behavior of each architectural component. This step may seem redundant, because we have already developed a specification for each component using sequence diagrams. However, the sequence diagrams only show specific scenarios and are possibly incomplete. A state machine and the used data specify the overall behavior of the component in question and will later serve as the basis for the specification of the composed architecture and for the implementation. An approach to construct state machines from sequence diagram is described in [16]. The sequence diagrams, on the other hand, can be used for testing.

   The local data for each component can be defined using class diagrams.

Output A set of state machines and class diagrams.

Validation Each architectural component is covered, and each state machine is *complete*, i.e., each possible input signal (as specified in Step 6) 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 6).

9. Develop the global architecture of the machine to be developed by combination of the subproblem architectures.

   Input Relationships between subproblems as specified in Step 3, results of Steps 5 and 6.

   Procedure 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:

   (a) 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.

   (b) Two application components belong to subproblems being related sequentially or by alternative.
   Such components should be merged into one application component.

   (c) 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.

   (d) 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.

   (e) Two application components belong to parallel subproblems and do not share any interface phenomena.
   Such components should be kept separately.

   Output A composite structure diagram for the global architecture, i.e. the architecture of the machine solving the original problem, and a set of interface classes for the global architecture.

   Validation The global architecture must contain all components and interfaces of all subproblem architectures. It must be possible to map all signals in the external interfaces to the phenomena at the machine interfaces of the context diagram developed in Step 2.

10. Define state machines for all components of the global architecture that were merged from components of different subproblem architectures by merging their respective state machines.

Input Results of Steps 8 and 9.

Procedure According to the case distinction we made in Step 9, we proceed as follows:
- Case 9a. 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 9b. The composition can be achieved by using composite states. The connecting arcs between the sub-automata depend on the problem.
- Case 9c. 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 9d. The merge has to be performed manually.

Output A set of state machines.

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. All sequence diagrams of all subproblems for the component specified in Step 7 describe the same behavior as the corresponding state machine.

11. Specify operations and private data types.
12. Implement and test the software system.

As the last two steps are beyond the scope of this paper, we do not describe them here.

6 Case Study

We now illustrate the process by the case study of an automatic teller machine (ATM).

6.1 Requirements, Domain Knowledge – Step 1

The mission of an ATM is to provide customers with money, provided that they are entitled to withdraw the desired amount.

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.
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.

An example of domain knowledge is that the *Money Case* sends *banknotes_removed* when the *Customer* takes the banknotes.

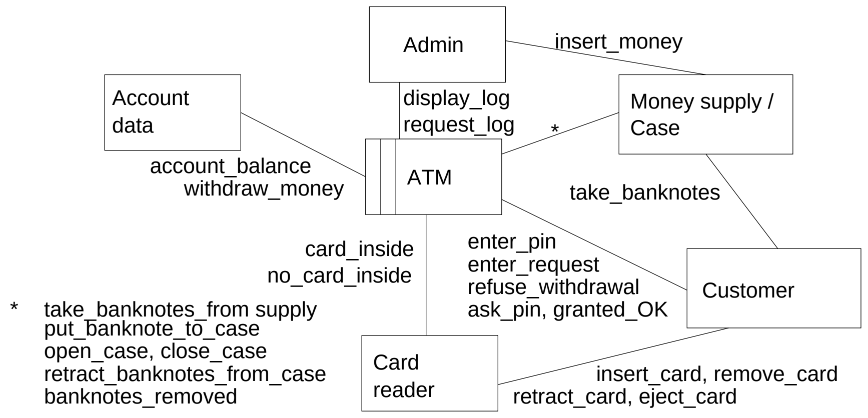


Fig. 3. Context Diagram for ATM Problem

6.2 Context Diagram - Step 2

Figure 3 shows the structure of the ATM problem context, where several domains and the corresponding shared phenomena are identified.

6.3 Subproblems - Step 3

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 4 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.

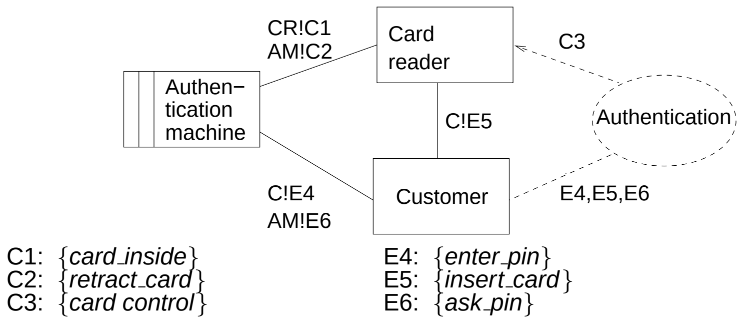
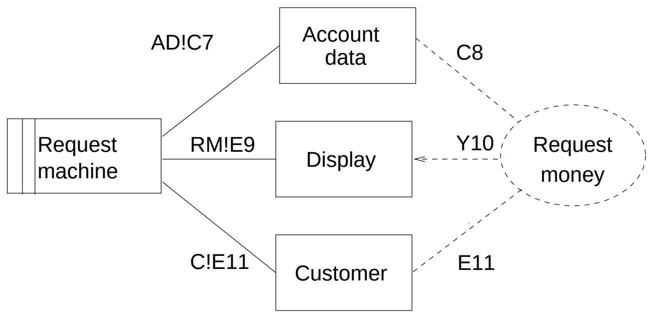


Fig. 4. Problem Diagram for Authenticate (commanded behavior variant)

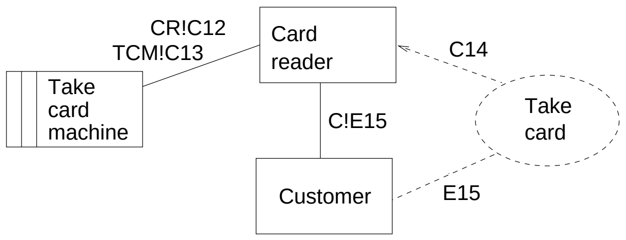
The problem diagram for *Request* (shown in Figure 5) describes requirement $R2$. It is a variant of the commanded information frame. This variant is called information display and described in [7].



C7: {*account_balance*}
C8: {*account_data*}
E9: {*granted_OK, refuse_withdrawal*}
Y10: {*withdrawal_possible*}
E11: {*enter_request*}

Fig. 5. Problem Diagram for Request (commanded information variant, information display)

Figure 6 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.

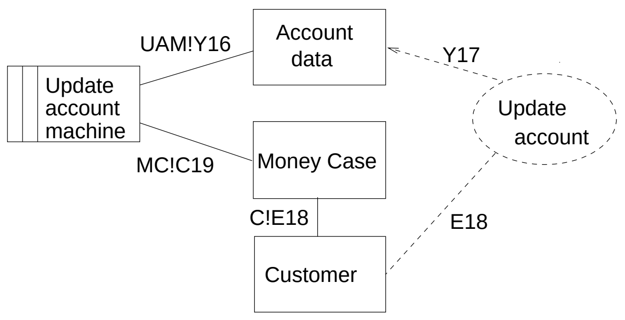


C12: {*card_inside*}
C13: {*eject_card, retract_card*}
C14: {*eject, retract*}
E15: {*take_card*}

Fig. 6. Problem Diagram for Take Card (required behavior)

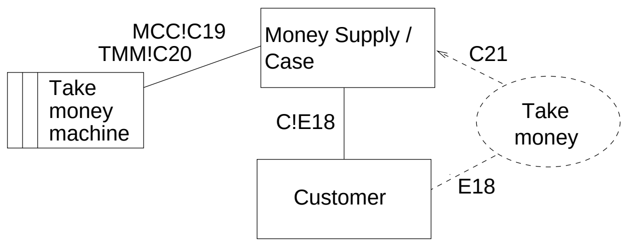
Figure 7 shows the problem diagram for *Update Account*, which is a variant of the *Workpieces* frame. The requirement $R4$ is described with this problem frame.

The problem diagram for *Take Money* (Figure 8) is a variant of the *Required Behaviour* problem frame (Figure 1), where we added the *Customer* and his/her connection with the *Money Case*. The requirement $R5$ is described with this problem frame.



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

Fig. 7. Problem Diagram for Update Account (Workpieces variant)



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*}

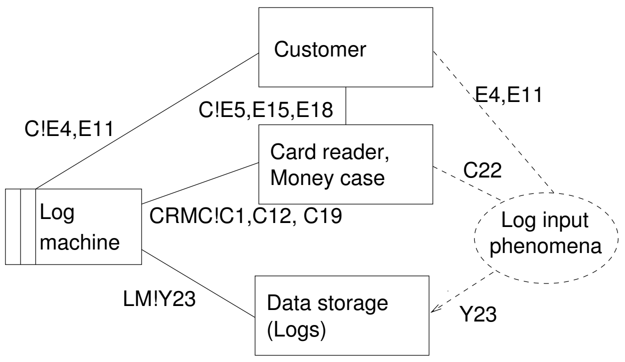
Fig. 8. Problem Diagram for Take Money (required behavior variant)

The problem diagram for *Log* (cf. requirement $R6$) is shown in Figure 9. The workpieces problem frame is instantiated. The domains *Card reader*, *Money case*, and *Customer* in the problem diagram represents the *User* in the problem frame.

The problem diagram for *Display Log* (shown in Figure 10) describes requirement $R7$. It is a variant of the commanded information frame. This variant is called information display and described in [7].

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.

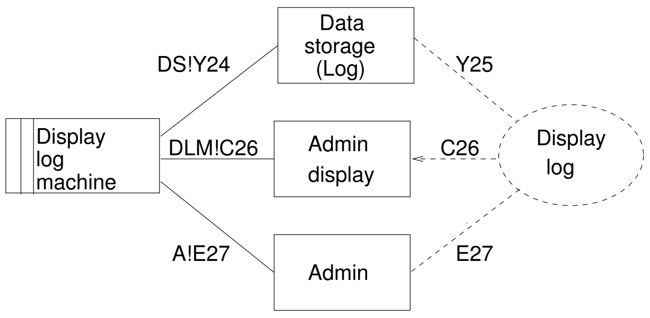
$< start >::= (< idle > || Log || DisplayLog)$
$< idle >::= (Authenticate < authenticated > | Authenticate < idle >)$
$< authenticated >::= (Request < granted > | Request < refused >)$
$< granted >::= (TakeCard < granted\_no\_card > | TakeCard < idle >)$
$< refused >::= TakeCardRefused < idle >$



Y22: {*card_reader_money_case_input_phenomena*}
E23: {*log_data*}

C1, C12 . . . as given in the other figures

Fig. 9. Problem Diagram for Log (Workpieces)



Y24: {*log_data*}
Y25: {*logged_input_phenomena*}
C26: {*log*}
E27: {*request_log*}

Fig. 10. Problem Diagram for Display Log commanded information variant, information display)

$< granted\_no\_card >::= (UpdateAccount||TakeMoney) < idle >$

The last line means that, once the card is removed and withdrawal is granted, both *UpdateAccount* and *TakeMoney* will take place in parallel, and then the idle state is reached.

6.4 Specification – Step 4

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 11. This diagram expresses that the card is retracted after 3 unsuccessful attempts. The *Customer* is authenticated if the valid PIN is entered.

The sequence diagram for the subproblem *Request* is shown in Figure 12. When an authenticated customer enters a request, his/her balance is checked and the access for the customer is granted or refused.

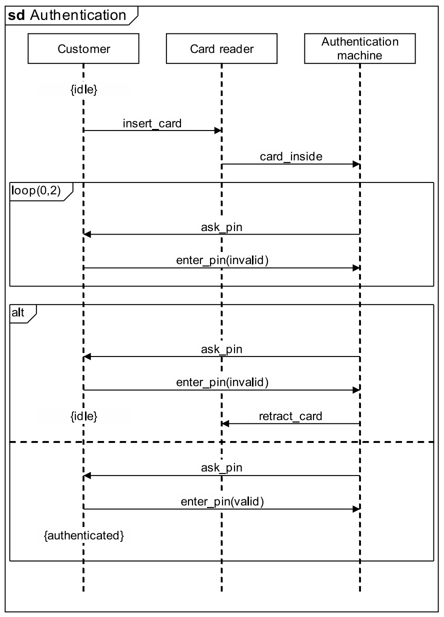


Fig. 11. Sequence Diagram for Authenticate

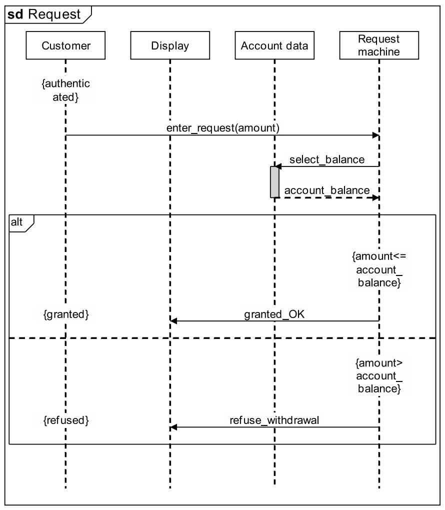


Fig. 12. Sequence Diagram for Request

The sequence diagrams for the subproblem *Take Card* are shown in Figures 13 and 14. 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. 14.

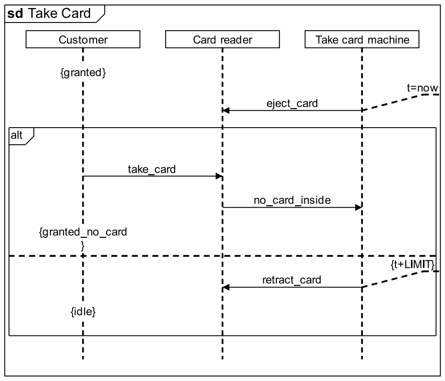


Fig. 13. 1st Sequence Diagram for Take Card

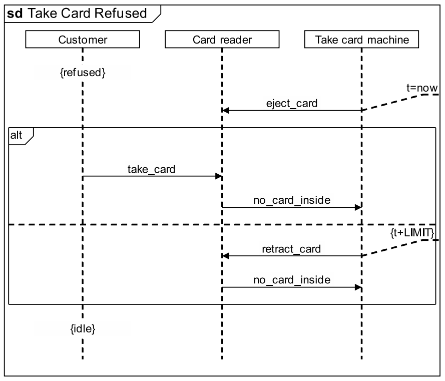


Fig. 14. 2nd Sequence Diagram for Take Card

The sequence diagram for the subproblem *Update Account* is shown in Figure 15. 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* is shown in Figure 16. 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 to the sequence shown in Fig. 15. This is possible, because both diagrams start with the same state invariant *granted_no_card* and only in Fig. 15 the state of the customer after this sequence is constraint.

The sequence diagram for the subproblem *Log* is shown in Figure 17. 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 17. 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.

6.5 Instantiated Architectural Patterns – Step 5

For each problem diagram, an architectural pattern from [5] is instantiated.

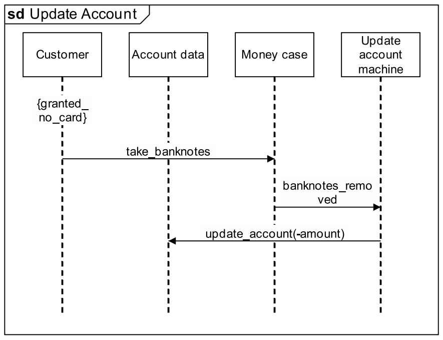


Fig. 15. Sequence Diagram for Update Account

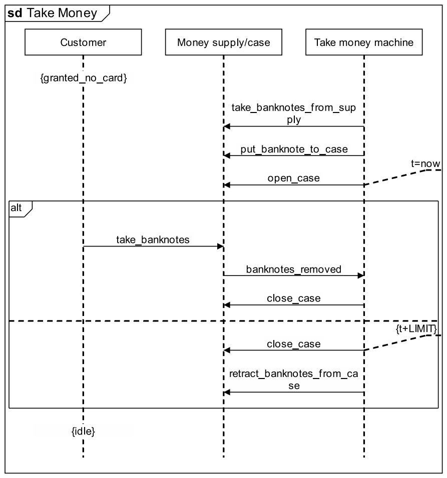


Fig. 16. Sequence Diagram for Take Money

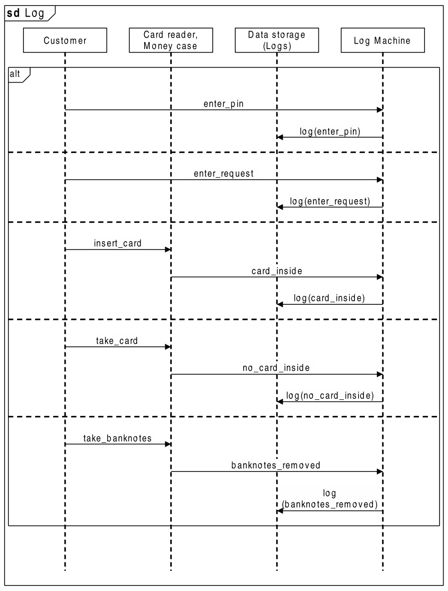


Fig. 17. Sequence Diagram for Log

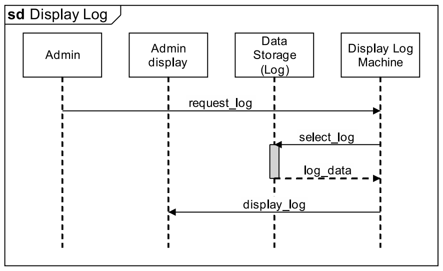


Fig. 18. Sequence Diagram for Display Log

The architecture for the subproblem *Authenticate* is shown in Fig. 19. It consists of an *Authentication Application*, a *Card In IAL*, a *Card Out IAL*, the corresponding HAL components, and a *User Interface*. The architecture for the subproblem *Request* is shown in Fig. 20. It consists of a *Request Application*, a *User Interface* and a *Data*

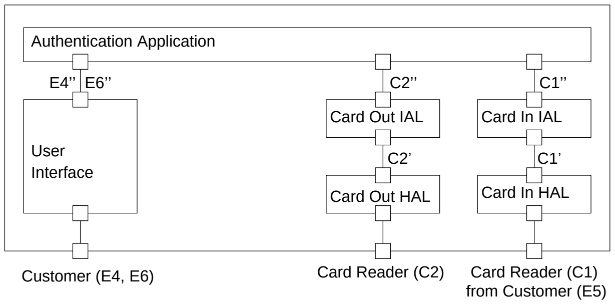


Fig. 19. Architecture for Authenticate

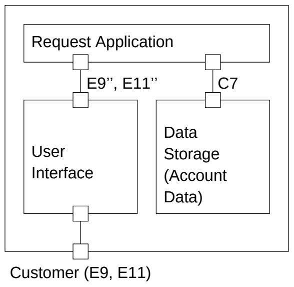


Fig. 20. Architecture for Request

Storage. The architecture for the subproblem *Take Card* (see Fig. 21) consists of the hardware abstraction layer and the interface abstraction layer for the Card, and a *Take Card Application*. The architecture for the subproblems *Update Account* is shown in

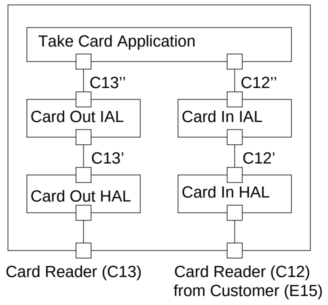
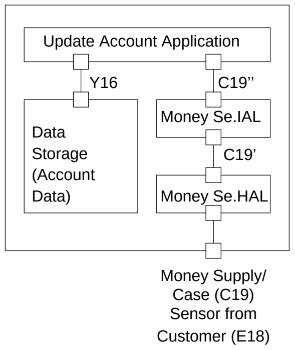


Fig. 21. Architecture for Take Card



Update Account Application
Y16
C19''
Data Storage (Account Data)
Money Se.IAL
C19'
Money Se.HAL
Money Supply/ Case (C19) Sensor from Customer (E18)

Fig. 22. Architecture for Update Account

Fig. 22. The architecture for the subproblems *Take Money* is shown in Fig. 23. The architecture for the subproblem *Log* consists of a *Log Application* and all components that handle input phenomena. It is shown in Fig.25. 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. 24).

6.6 Interface Classes – Step 6

As we use very abstract phenomena for the case study, the IAL and the HAL are trivial, and we obtain their interface phenomena simply by renaming the external phenomena. Phenomena controlled by the machine become provided interfaces, and phenomena controlled by the environment become required interfaces of the complete machine. A provided interface class is e.g. `C19` (see Figs. 7 and 22) with the method

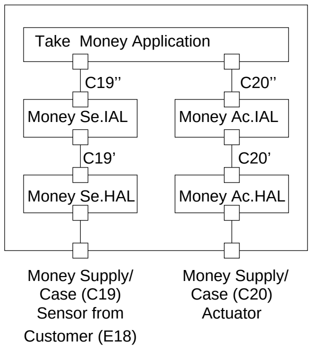


Fig. 23. Architecture for Take Money

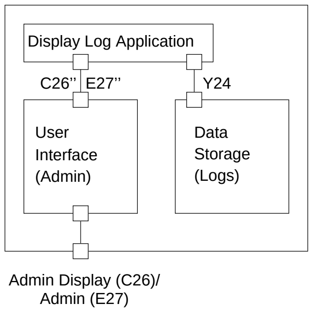


Fig. 24. Architecture for Display Log

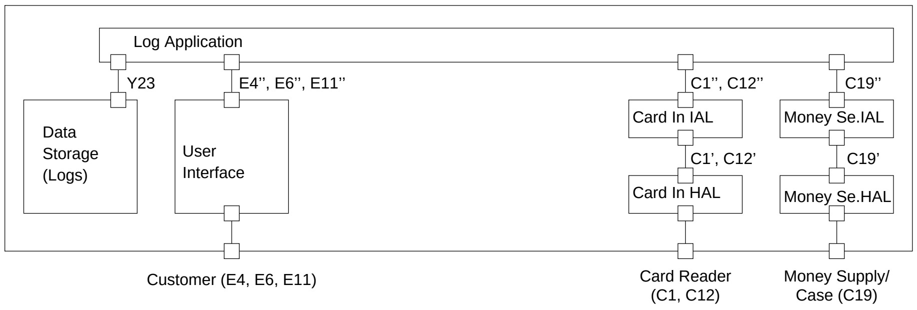


Fig. 25. Architecture for Log

banknotes_removed(). The interfaces of the application components are e.g. C19" with the method *banknotes_removed"()* or e.g. Y16 with the method *update_account(amount: Integer)*.

In the following tables some interface classes for the subproblem architectures are specified.

$\langle\langle interface\rangle\rangle$ 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"()

$\langle\langle interface\rangle\rangle$ E9”
granted_OK”()
refuse_withdrawal”()

$\langle\langle interface\rangle\rangle$ E11”
enter_request”(amount: Integer)

$\langle\langle interface\rangle\rangle$ C7
select_account”(): Integer

6.7 Sequence Diagrams for Components – Step 7

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”()*.

6.8 State Machines for Components – Step 8

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 26 and 27.

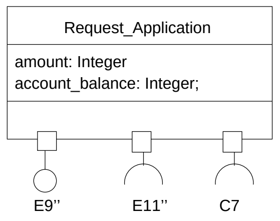


Fig. 26. Class Diagram for Request Application

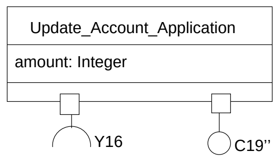


Fig. 27. Class Diagram for Update Account Application

Each sequence diagram constructed in Step 7 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 and exits with *failed* after 3 unsuccessful attempts (cf. Fig. 28) as specified in Fig. 11.

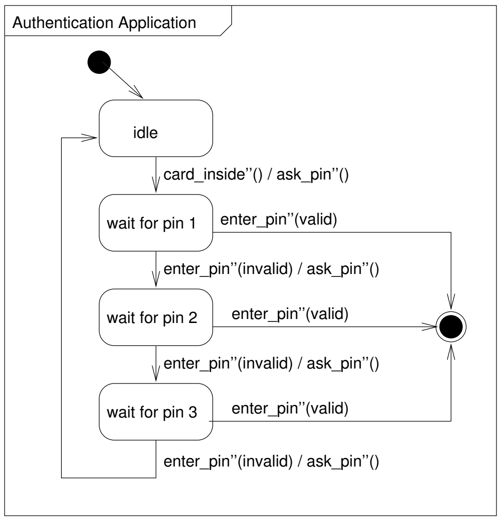

Authentication Application
idle
card_inside''() / ask_pin''()
wait for pin 1
enter_pin''(valid)
enter_pin''(invalid) / ask_pin''()
wait for pin 2
enter_pin''(valid)
enter_pin''(invalid) / ask_pin''()
wait for pin 3
enter_pin''(valid)
enter_pin''(invalid) / ask_pin''()

Fig. 28. State Machine for Authenticate Application

The state machine for the *Request Application* is shown in Fig. 29. It is consistent with the sequence diagrams in the Figures 12 and 14.

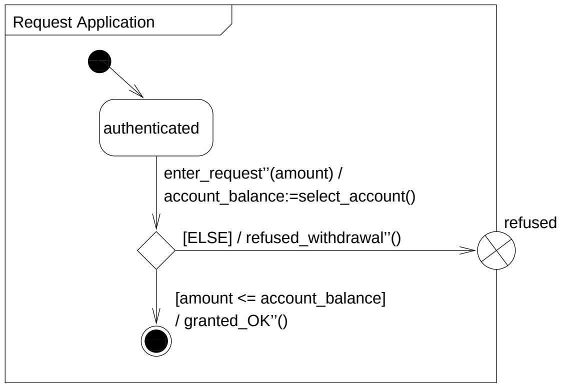

Request Application
authenticated
enter_request''(amount) /
account_balance:=select_account()
[ELSE] / refused_withdrawal''()
refused
[amount <= account_balance]
/ granted_OK''()

Fig. 29. State Machine for Request Application

The state machine shown in Fig. 30 requires a timer as defined in [12].

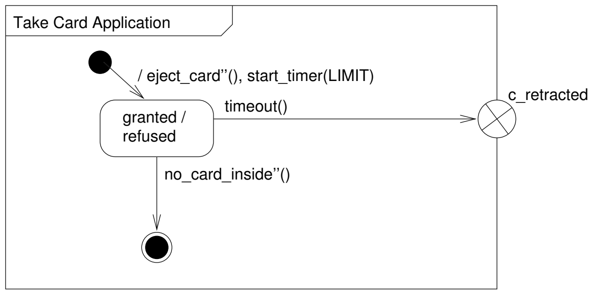


Fig. 30. State Machine for Take Card Application

The sequence diagram of Fig. 15 can be transformed into the state machine shown in Fig. 31.

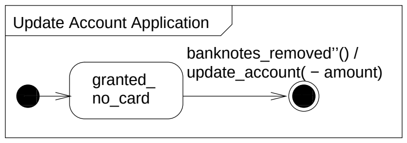


Fig. 31. State Machine for Update Account Application

The state machine for ejecting the requested amount of money and retracting this money if it was not taken within a certain time limit is shown in Fig. 32.

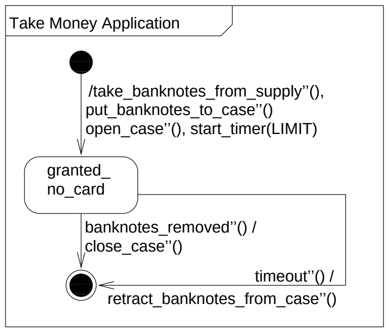

Take Money Application
/take_banknotes_from_supply''(),
put_banknotes_to_case''()
open_case''(), start_timer(LIMIT)
granted_
no_card
banknotes_removed''() /
close_case''()
timeout''() /
retract_banknotes_from_case''()

Fig. 32. State Machine for Take Money Application

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. 33

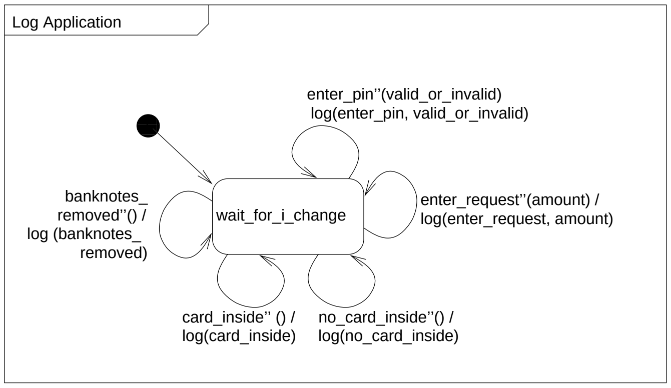


Fig. 33. State Machine for Log Application

The logged data can be requested by the *Admin*. This functionality is implemented in the state machine for *Display Log Application* (cf. 34).

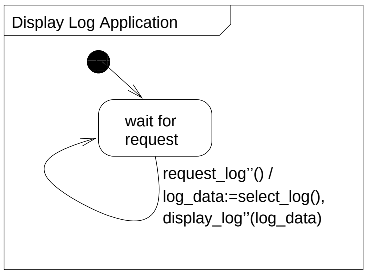


Fig. 34. State Machine for Display Log Application

Also the other components must be specified with state machines. As an example the statemachines for the *User Interface* components are presented in Figures 35, 36, and 37.

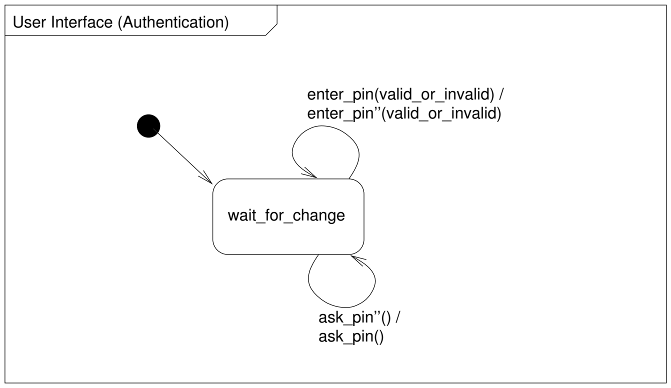

User Interface (Authentication)
enter_pin(valid_or_invalid) /
enter_pin''(valid_or_invalid)
wait_for_change
ask_pin''() /
ask_pin()

Fig. 35. State Machine for Authenticate User Interface

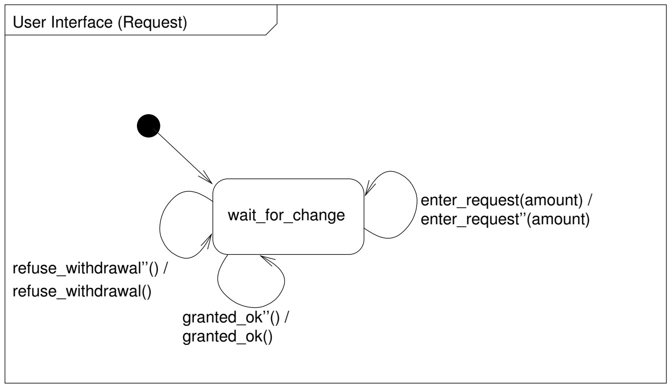


Fig. 36. State Machine for Request User Interface

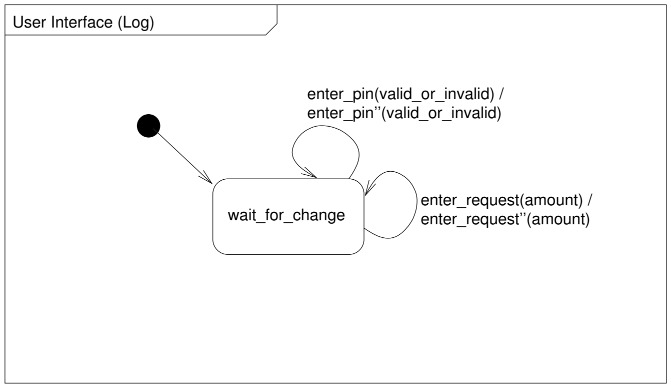


Fig. 37. State Machine for Log User Interface

6.9 Global Architecture – Step 9

The composed architecture for the ATM is shown in Figure 38. It shows that our patterns yield appropriate architectures for subproblems fitting to problem frames, and that these architectures can be combined in a modular way to obtain an architecture of the overall system according to the rules of Step 9. The following merges have been done:

The problem *Take Money* and the problem *Update Account* are parallel and share some input phenomena (cf. case 9d). We decided to merge the corresponding application components. The problem *Log* is related parallel to all other subproblems, sharing input phenomena (cf. case 9d). 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 9b). Therefore the corresponding applications are also merged. We call the resulting component *Main Application*. The problem *Display Log* is parallel and do not share any interface phenomena (cf. case 9e). Hence, the component *Display Log Application* is not merged. All components that are IALs or HALs (cf. case 9a) are merged with the components of the same name in the other subproblem architectures.

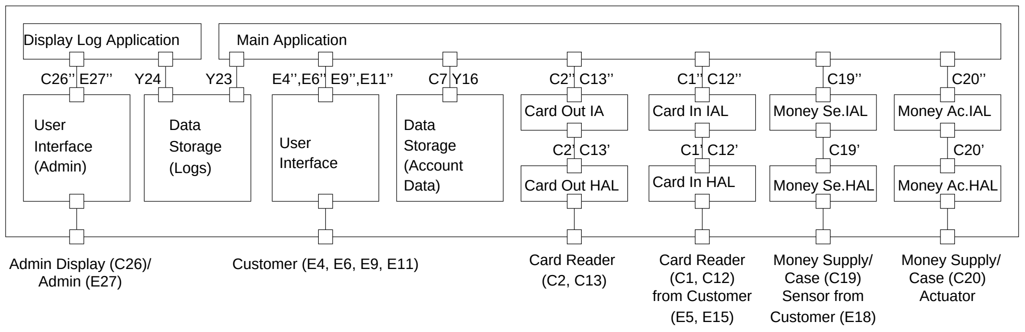


Fig. 38. Composed Architecture

6.10 Complete State Machines for all components – Step 10

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. 39).

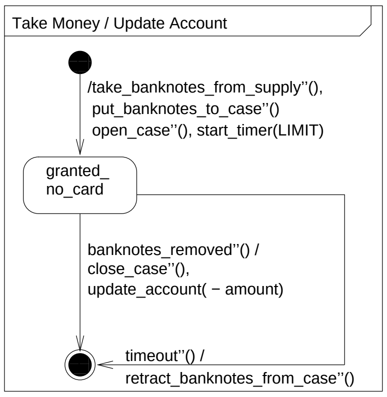

Take Money / Update Account
/take_banknotes_from_supply''(), put_banknotes_to_case''() open_case''(), start_timer(LIMIT)
granted_ no_card
banknotes_removed''() / close_case''(), update_account(− amount)
timeout''() / retract_banknotes_from_case''()

Fig. 39. Merged State Machine for Take Money and Update Account Application

The state machines for *Authentication Application*, *Request Application*, and *Take Money/Update Account* must then 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 40 shows the result of the merge.

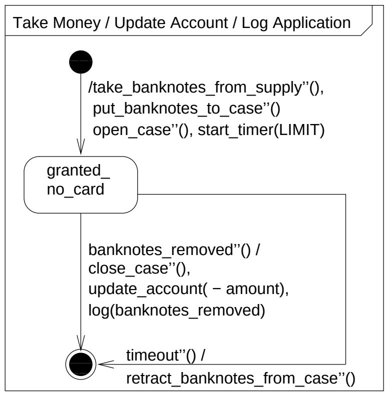
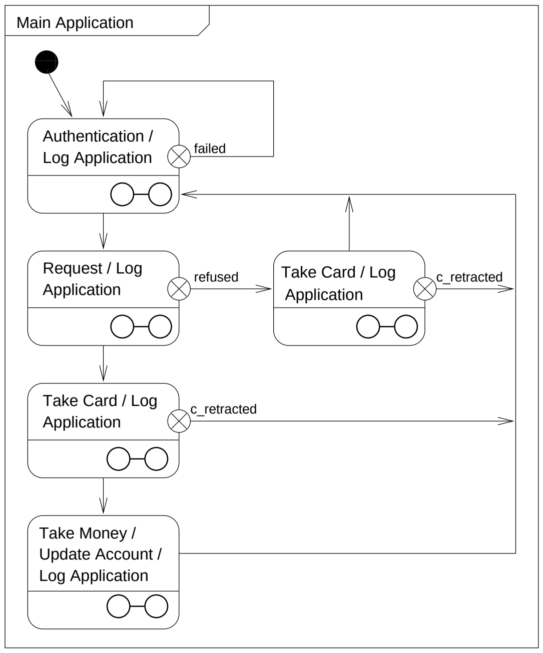

Take Money / Update Account / Log Application
/take_banknotes_from_supply''(), put_banknotes_to_case''() open_case''(), start_timer(LIMIT)
granted_ no_card
banknotes_removed''() / close_case''(), update_account(− amount), log(banknotes_removed)
timeout''() / retract_banknotes_from_case''()

Fig. 40. 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. 41). The resulting state machine exactly reflects the grammar describing the dependencies of the subproblems.



Main Application
Authentication / Log Application
failed
Request / Log Application
refused
Take Card / Log Application
c_retracted
Take Card / Log Application
c_retracted
Take Money / Update Account / Log Application

Fig. 41. State Machine for all Sequential and Alternative Problems

Then the state machines for the IALs, the HALs and the *User Interface*s must be merged. The merged state machine for the *User Interface* is shown in Figure 42.

7 Conclusions

In this paper we presented a (partial) development process from requirements elicitation to detailed design. This process is based on patterns provided by problem frames and architectural styles. The expression of the relationships between the subproblems is used to guide the composition of the designed components. The contributions of our approach are the following:

– Our process gives concrete guidance of how to use problem frames and architectural patterns, in connection with a model-based approach to software development, using various UML notations.
– We provide a systematic way of exploiting information on how a problem was decomposed into subproblems for constructing the overall solution to a problem

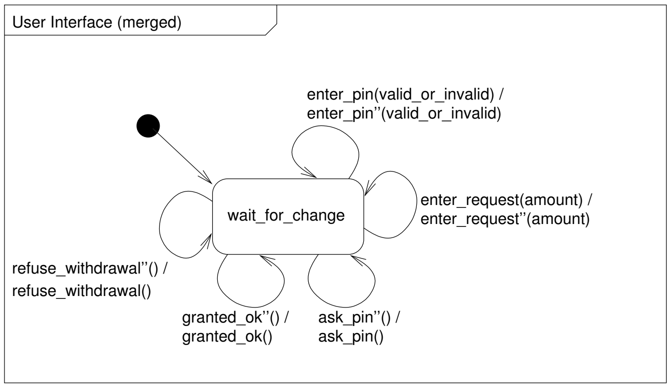


Fig. 42. Merged State Machine for the User Interface

from the solutions of its subproblems. For top-down decomposition, this may be simple; for use-case or parallel problem decomposition, however, it is not obvious how to obtain the overall solution from the solutions of the subproblems.

– The process results in detailed descriptions of the software components to be implemented and tested. The state machines (and data descriptions) are an appropriate basis for implementation, whereas the sequence diagrams provide scenarios against which the implemented software can be tested.
– Because of the extensive validation contained in our process, inconsistencies are found before starting the implementation.
– Because of the systematic problem decomposition and solution composition, our process can be used for large, realistic systems.

Although the work presented here is independent of any formal specification language, if desired, it would be possible to accompany the architectural descriptions with a formal specification development along the ideas of [6–8], and also to take into account properties as in [1] (cf. Section 4).

In the future, we intend to extend this work in several directions. First, we want to treat complex data structures in more detail. Second, since our approach aims at a guided and integrated use of several techniques and several patterns, we would like to explore how to integrate the use of design patterns in this development. Third, we intend to elaborate more on the later phases of software development. For example, we want to investigate how to generate code from the outputs of our process. Finally, we aim at tool support for our process, preferably integrating existing UML tools. Our long-term goal is to apply our process in industrial applications.

References

1. L. Barroca, J. L. Fiadeiro, M. Jackson, R. C. Laney, and B. Nuseibeh. Problem frames: A case for coordination. In R. D. Nicola, G. L. Ferrari, and G. Meredith, editors, *Coordination Models and Languages, 6th International Conference, COORDINATION 2004, Pisa, Italy, February 24-27, 2004, Proceedings*, pages 5–19, 2004.

2. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.
3. 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.
4. J. Cheesman and J. Daniels. *UML Components – A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.
5. C. Choppy, D. Hatebur, and M. Heisel. Architectural patterns for problem frames. *IEE Proceedings – Software, Special issue on Relating Software Requirements and Architecture*, 152(4):198–208, 2005.
6. C. Choppy and M. Heisel. Use of patterns in formal development: Systematic transition from problems to architectural designs. In M. Wirsing, R. Hennicker, and D. Pattinson, editors, *Recent Trends in Algebraic Development Techniques, 16th WADT, Selected Papers*, LNCS 2755, pages 205–220. Springer Verlag, 2003.
7. C. Choppy and M. 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.
8. C. Choppy and G. Reggio. Using CASL to Specify the Requirements and the Design: A Problem Specific Approach. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th WADT, Selected Papers*, LNCS 1827, pages 104–123. Springer Verlag, 2000. A complete version is available at `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio99a.ps`.
9. M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison Wesley, 1997.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, 1995.
11. J. G. Hall, M. Jackson, R. C. Laney, B. Nuseibeh, and L. Rapanotti. Relating Software Requirements and Architectures using Problem Frames. In *Proceedings of IEEE International Requirements Engineering Conference (RE'02)*, Essen, Germany, 9-13 September 2002.
12. M. Heisel and D. 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`.
13. M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.
14. 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.
15. L. Lavazza and V. D. Bianco. A UML-Based Approach for Representing Problem Frames. In K.Cox, J. Hall, and L. Rapanotti, editors, *Proc. 1st International Workshop on Advances and Applications of Problem Frames (IWAAPF)*. IEE Press, 2004.
16. N. Mansurov. Automatic synthesis of sdl from msc. Technical report, klocwork, Inc., 2003. `http://www.klocwork.com/company/downloads/WP_SDL_from_MSC.pdf`.
17. D. L. Parnas and J. Madey. Functional documents for computer systems. In *Science of Computer programming*, volume 25, pages 41–61, 1995.
18. 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.
19. M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

20. A. Sutcliffe. *The Domain Theory, Patterns for Knowledge and Software Reuse*. Addison Wesley, 2002.
21. A. Sutcliffe and N. Maiden. The Domain Theory for Requirements Engineering. *IEEE Transactions on Software Engineering*, 24(3):174–196, 1998.
22. A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992. TAN a 92:1 2.Ex.
23. UML Revision Task Force. *OMG UML Specification*. `http://www.uml.org`.
24. 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 available under http://www.research.att.com/˜pamela/ori.html#fre.