Designing Architectures from Problem Descriptions by Interactive Model Transformation

Azadeh Alebrahim,
Isabelle Côté,
Maritta Heisel
Universität Duisburg-Essen,
Germany

Christine Choppy
LIPN, UMR CNRS 7030,
Université Paris 13,
France

Denis Hatebur
Institut für technische
Systeme GmbH,
Germany

ABSTRACT

We present a structured approach to systematically derive a software architecture from a given problem description based on problem frames and a description of the environment. Our aim is to re-use the elements of the problem descriptions in creating the architecture. The derivation is performed by transforming the problem description into an initial architecture, where each subproblem corresponds to a component. The transformation is supported by model transformation rules, formally specified as operations with pre- and postconditions. This specification serves as a blueprint for a tool supporting the architectural design. We illustrate our method by the example of a patient care system.

Keywords

Requirements analysis, architectural design, model transformation, UML, problem frames

1. INTRODUCTION

Model-based development is a promising approach to develop high-quality software. Its basic idea is to construct a sequence of models that are of an increasing level of detail and cover different aspects of the software development problem and its solution. We use UML diagrams to set up models for the different phases of the software development, and to express validation conditions and specifications of model transformations.

Our methodology provides detailed procedures for problem-based requirements analysis and architectural design. The requirements analysis process is based on patterns called *problem frames* [1]. The process for architectural design (see Sect. 4) describes a structured approach to create software architectures starting with problem descriptions derived by problem-based requirements analysis. Since we use UML to express problem as well as architectural descriptions, we achieve a seamless transition between the two phases.

We have implemented a tool called *UML4PF* that supports developers in performing requirements analysis using problem frames and in deriving software architectures from problem descriptions. The tool is based on the Eclipse framework and uses the plugins *Eclipse Modeling Framework* (EMF) and OCL. We defined two profiles extending the UML meta-model. 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 integrity and coherence of the different models, we have defined a number of integrity conditions expressed as OCL conditions. For more details on the tool, see [2].

This paper elaborates on an earlier work [3], which informally describes a general method for the systematic development of architectures from problem descriptions. There, the corresponding validation conditions are discussed in detail. In this paper, we define an interactive model transformation operator that allows us to transform a set of problem descriptions into an initial architecture. Such an operator is defined by pre- and postconditions expressed in OCL. Necessary user interaction corresponds to input parameters. This approach has the advantage that it can be automated gradually. The transformations are performed on the model level. In addition to the model information, a graphical representation of the model exists. In contrast to the model transformation, we do not present concrete rules for transforming the graphical representation.
The paper is structured as follows: The case study of a patient care system (PCS) is sketched in Sect. 2. Architectural descriptions and the corresponding UML profile are described in Sect. 3. In Sect. 4, we describe the process to derive software architectures from problem descriptions. Related work is discussed in Sect. 5, and we conclude in Sect. 6.

2. CASE STUDY

As a running example, we consider a *patient care system*, which displays the vital signs of patients to physicians and nurses, and controls an infusion flow according to previously configured rules. Table 1 lists the functional requirements of the PCS case study. The environment in which the software to be built (called *machine*) will operate is represented by a *context diagram*. The context diagram for the patient care system contains the biddable domains Patient and PhysiciansAndNurses, and causal domains for O2Sensor, HeartbeatSensor, InfusionPump, and Terminal. Figure. 1 shows the problem diagram for the requirement R_Config. R_Config constrains the domain PatientSettings since physicians and nurses can change the configuration of PatientSettings. The machine PatientCareSystem is stereotyped ≪machine≫.

For more information on the problem frame approach and our UML version of it, see [1, 3].

3. ARCHITECTURAL DESCRIPTIONS

Each context diagram contains a machine domain, representing the software to be developed. For this machine domain, we design an architecture that is described using composite structure diagrams [4]. In such a diagram, the components with their ports and the con-

Requirement	≪refersTo≫	≪constrains≫
R_WarnShow: The vital signs should be displayed, and an alarm should be raised if the vital signs exceed the limits.	Patient, PatientSettings	Terminal
R_Config: Physicians and nurses can change the configuration.	PhysiciansAndNurses	PatientSettings
R_Ctrl: The infusion flow is controlled according to the configured doses for the current vital signs.	Patient, PatientSettings	InfusionPump

Table 1: Requirements of Patient Care System

nectors between the ports are given. The components are another representation of UML classes. The ports are typed by a class that uses and realizes interfaces. 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 2.

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 specialization of the stereotype ≪Component≫. This must be recorded in case such a component is changed. Reused components may also be used in other projects.

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). For the architectural connectors, we allow the same stereotypes as for connections in context and problem diagrams, e.g. ≪ui≫ or ≪tcp≫. However, these stereotypes extend the UML meta-class Connector (instead of the meta-class Association).

4. DERIVING ARCHITECTURES INTERACTIVELY FROM PROBLEM DESCRIPTIONS

We now present the model transformations that serve to derive software architectures from problem descriptions: we derive an initial architecture. It contains one component for each subproblem. The purpose of this step is to collect the necessary information for the architectural design from the requirements analysis phase. The machines from the subproblem descriptions and domains being part of the machine in the context diagram become components in the initial architecture. This initial architecture is used to show the interrelation between the above-mentioned components. At this stage, the submachine components are not yet coordinated.

Based on the problem descriptions and the context diagram, the initial architecture can be created, using the model transformation operator *createInitialArchitecture* (see Listing 1), which takes the name of a machine domain as its input. The initial architecture consists of a composite structure diagram for the machine in the context diagram. In our example, the context diagram contains the machine domain, PatientCareSystem.

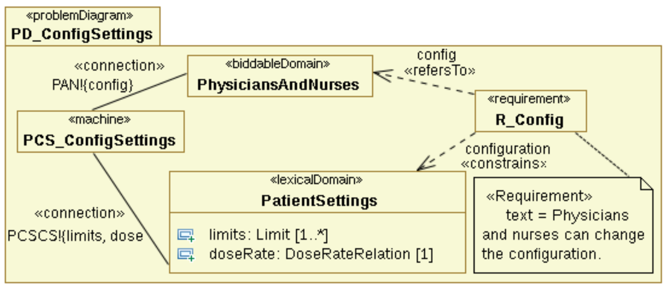

Figure 1: Problem Diagram "ConfigSettings" of Patient Care System

The following preconditions have to be fulfilled:

1. The machine domain with the name (<*nameOfMachineInCD*>) exists in the context diagram (see line 3 in Listing 1), e.g., machine 'PatientCareSystem' exists as machine domain in the context diagram.
2. Each problem diagram contains exactly one machine domain (line 4). A machine in a problem diagram (a so-called submachine) represents a fraction of the functionality the overall machine has to fulfill.

   In our example, Fig. 1 shows the machine domain PCS_ConfigSettings. The other components of the initial architecture shown in Fig. 2 correspond to the other subproblems given in Tab. 1.
3. The machine in the context diagram (<*nameOfMachineInCD*>) is composed of the machines in the problem diagrams (line 5), e.g., machine domain PCS_ConfigSettings is part of the machine domain in the context diagram.
4. The problem diagrams and the context diagram are consistent (line 6) (i.e., all domains and interfaces of the problem diagrams are related to elements in the context diagram). In our example, the problem diagram is consistent with the context diagram.

After the operation is executed, the resulting composite structure diagram fulfills the following postconditions:

1. External ports for each direct connection from the machine to another domain in the context diagram – not being part of the machine domain – as well as required and provided interfaces exist (see line 8 in Listing 1). Machine PatientCareSystem in our example has four external ports. Provided and required interfaces exist according to the observed and controlled interfaces in the context diagram. For example, a required interface PCS!{InfusionFlow} exists (see Fig. 2).
2. For each submachine related to the machine in the context diagram as well as for lexical domains being part of the machine (lines 9 - 12) (internal) components exist (line 13). These components are equipped with ports (line 14 and corresponding required (line 15) and provided interfaces (line 16) based on the controlled and observed interfaces in the problem diagrams and the domain types exist. We distinguish between lexical and non-lexical domain types.

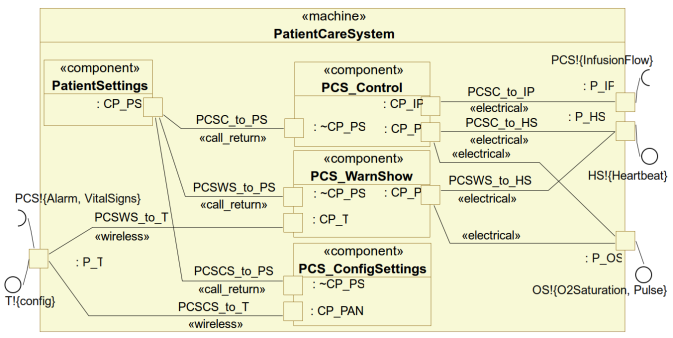

Figure 2: Initial architecture for the patient care system

The ports of these components are connected to the external ports according to the interfaces in the problem diagrams. The connectors have the same stereotype as their corresponding associations in the context diagram (line 17).

```
 1 createIntialArchitecture(nameOfMachineInCD: String)
 2 PRE:
 3 isMachineDomainInCD(nameOfMachineInCD) = true and
 4 problemDiagramsHaveOneMachine() = true and
 5 tcdMachineIsComposedOfPDMachine() = true and
 6 pdsConsistentToTcd() = true
 7 POST:
 8 externalPortsExist(nameOfMachineInCD)
 9 let machine_parts: Set(Class) =
10   collectSubmachines(nameOfMachineInCD) ->
                 union(collectRelevantLexDomains(
         nameOfMachineInCD))
11 in
12 machine_parts -> forAll(m_component |
13 ComponentExists(m_component) and
14 PortsExist(m_component, nameOfMachineInCD) and
15 RequiredInterfacesExist(m_component,
         nameOfMachineInCD)and
16 ProvidedInterfacesExist(m_component,
         nameOfMachineInCD)and
17 ConnectorsToExternalPortsExist(m_component,
         nameOfMachineInCD)) and
18 ConnectorsBetweenInternalPortsExist(m_component)
```

Listing 1: Specification of operation *createInitialArchitecture*

3. Connectors between the internal components exist (line 18). They are connected based on their counterpart associations in the context diagram. The connectors have the same stereotype as the associations in the context diagram.

Note that in order to keep the OCL expressions legible, we decided to use auxiliary expressions (see Listing 1) in the pre- and postconditions we present in this paper. For all such auxiliary expressions, we have defined corresponding OCL expressions.

5. RELATED WORK

Fujaba is a tool-suite for model-based software engineering and re-engineering. It is conceived as round-trip engineering tool for UML and Java [5]. Fujaba has several plug-ins, one of these is the plugin *MoTE/MoRTEn*. It is used for model-to-model transformations and synchronization. The transformations are based on triple graph grammars [6]. The transformations are modeled graphically. However, some constraints are not expressed graphically. Instead, Java-dependent expressions are used. Stölzel et al. [7] use the Dresden OCL Toolkit [8] to provide an independent constraint language for checking Fujaba's model transformations. We use the OCL implementation of EMF for both, consistency checks as well as transformation specifications. Therefore, we do not use graphical representations of the transformations but specify them through preconditions describing the state of the model *before* the transformation takes place and postconditions describing the state *after* the transformation is executed.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have shown how software architectures can be derived from problem descriptions in a systematic way. This systematic derivation takes place in the context of a requirements engineering process based on problem frames. The derivation method is described by transformation operations specified by pre- and postconditions. The operations are parameterized to express necessary design decisions.

Our tool is easily extensible with new transformation rules, for example, rules for pattern application or for taking non-functional aspects into account.

In summary, our contributions are the following:

- We have developed a detailed method to interactively derive software architectures from problem descriptions.
- We have defined UML profiles that make it possible to express problem descriptions as well as architectural designs as UML models. This allows a seamless integration of the requirements analysis and architectural design phases.
- We have provided formally specified transformation operations that allow software engineers to achieve the transition from problem descriptions to architectural designs. So far, the transformations have to be performed manually. However, the result can be checked by our tool.

We are currently working on extending our approach to be able to derive implementable architectures. The purpose of such an implementable architecture is to introduce coordination mechanisms between the different submachine components of the initial architecture and its external interfaces. In the future, we intend to implement all model transformations presented in this paper, so that they can be applied automatically. We plan to integrate the use of architectural patterns in our transformations. Furthermore, we want to extend our approach to support the development of design alternatives according to quality requirements, such as performance or security, and to support software evolution.

7. REFERENCES

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

[2] I. Côté, D. Hatebur, M. Heisel, and H. Schmidt, “UML4PF – a tool for problem-oriented requirements analysis,” in *Proc. Int. Conf. on Requirements Engineering (RE)*. IEEE, 2011.

[3] C. Choppy, D. Hatebur, and M. Heisel, “Systematic architectural design based on problem patterns,” in *Relating Software Requirements and Architectures*, P. Avgeriou, J. Grundy, J. Hall, P. Lago, and I. Mistrik, Eds. Springer, 2011, ch. 9, pp. 133–159.

[4] "UML Revision Task Force", *OMG Unified Modeling Language: Superstructure*, February 2009, available at http://www.omg.org/docs/formal/09-02-02.pdf.

[5] T. Klein, U. A. Nickel, J. Niere, and A. Zündorf, “From UML to Java and back again,” University of Paderborn, Tech. Rep., 1999.

[6] A. Schürr, “Specification of graph translators with triple graph grammars,” in *Proc. of the 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG ‘94)*. Springer, 1995.

[7] M. Stölzel, S. Zschaler, and L. Geiger, “Integrating OCL and model transformations in Fujaba,” 2006.

[8] Dresden OCL, “Dresden OCL Toolkit,” 2011, www.dresden-ocl.org.