Making Pattern- and Model-Based Software Development more Rigorous

Denis Hatebur[1,2] and Maritta Heisel[1]

[1] Universität Duisburg-Essen, Germany, Fakultät für Ingenieurwissenschaften, email: maritta.heisel@uni-due.de

[2] Institut für technische Systeme GmbH, Germany, email: d.hatebur@itesys.de

Abstract. Pattern-based and model-based software development approaches have a high potential to improve the quality of software. Patterns allow engineers to re-use established and proven development knowledge. Developing software by constructing a sequence of models provides engineers with various possibilities for validation, because the different development models are not independent of each other and hence can be checked for coherence.
We present a UML profile equipped with numerous OCL constraints that supports a pattern- and model-based software development process. The basis of the UML profile is a representation of problem frames, which are patterns supporting requirements analysis. OCL constraints provide a formal underpinning of the development process and allow one to perform semantic checks every time a new model is set up. Our approach is supported by a tool, called UML4PF. The tool is based on the Eclipse development environment, extended by an EMF-based UML tool, in our case, Papyrus. In this paper, we specifically focus on ensuring that problem frames are instantiated correctly. We illustrate our approach by the case study of an automatic teller machine.

1 Introduction

Software development with formal methods usually starts with a formal specification. Then, this specification is refined to code, possibly carrying out several refinement steps that must be proven correct. Another possibility is to annotate code with formally expressed assertions and prove the correctness of the code with respect to the assertions.

Today, patterns do not play a prominent role in such formal development processes. However, patterns are a very important means to reuse development knowledge. Therefore, they should also be integrated in formal development processes. Patterns are defined for all stages of the software development process. Requirements analysis can be supported by problem frames [17] and analysis patterns [10]. Coarse-grained design can make use of architectural styles [21]. Design patterns [11] are a well-known means to perform fine-grained design. Idioms [8] support programming, and test patterns [19] can be used in the testing phase.

Since the different artifacts of the software development process can be expressed as models, pattern- and model-based development fit very well together. The advantage of model-based development is that it is possible to check integrity conditions between the models. When such a check is performed by formal means, pattern- and model-based development processes can be carried out in a formal way.

In this paper, we present a pattern- and model-based requirements analysis process that is tool-supported and that allows one to check semantic integrity conditions that are expressed in the formal language OCL[1] (Object Constraint Language) [23]. We thus contribute to the goal of combining the use of patterns with formal development. That process is based on problem frames [17]. These are patterns that classify software development problems.

We have defined a UML profile that extends the UML meta-model [25]. It allows us to express the diagrams that are set up when performing requirements analysis with problem frames in UML notation. The defined OCL conditions provide a formal semantic underpinning of the problem frame approach. Automatically checking the constraints makes it possible to detect semantic errors in the requirements analysis process.

We have developed a tool, called UML4PF. With this tool, developers can draw different diagrams that have to be set up during the requirements engineering process. The diagrams are mapped to parts of a global model, and a graphical representation of this part. Every time a new diagram is finished, the developer may call the UML4PF validator. This causes the defined OCL conditions to be evaluated, based on the model information. If one of the conditions is not satisfied, a semantic error has been detected in one of the diagrams, or integrity conditions between two or more diagrams are violated. UML4PF also points out which condition is violated in which diagram(s), thus supporting the developer in locating and correcting the error.

Elements of the created model can be re-used in later development phases. We can also validate that the artifacts of later development steps, such as specification and architectural design, are consistent with the requirements engineering diagrams.

In the following, we introduce problem frames and the corresponding UML profile in Section 2. The tool UML4PF is described in Section 3. In Section 4, we show how to check that problem frames are correctly instantiated during a development[2]. Section 5 illustrates the approach by the case study of an automatic teller machine. Section 6 discusses related work. Finally, Sect. 7 concludes the paper with a summary, ongoing work, and directions for future research.

2 UML Profile for Problem Frames

Problem frames are a means to describe software development problems. They were introduced by Jackson [17], 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."*

Figure 1 shows a problem frame called *commanded behaviour* in UML notation. Informally, *there is some part of the physical world whose behaviour is to be controlled with commands issued by an operator. The problem is to build a machine that will accept the operator's commands and impose the control accordingly.* [17]. We describe problem frames using class diagrams extended by stereotypes (see Fig. 1). All elements of a problem frame diagram act as placeholders, which must be instantiated to represent concrete problems. Doing so, one obtains a problem description that belongs to a specific problem class.

[1] We have chosen OCL because it is part of UML, which is widely used and well equipped with tool support.

[2] Many other checks are defined that are not presented in this paper.

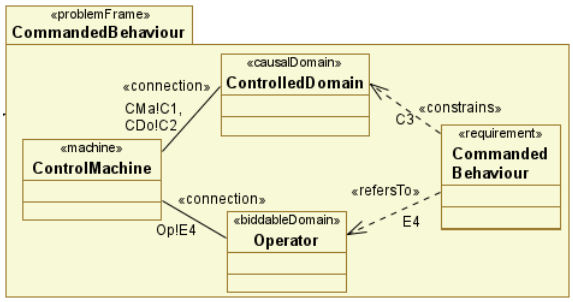

Fig. 1. *Commanded behaviour* problem frame using UML notation

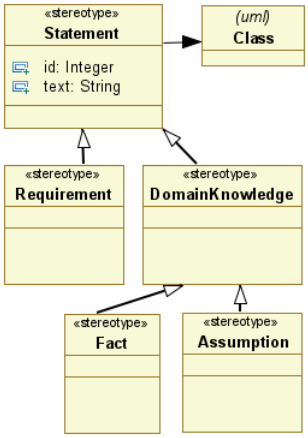

Fig. 2. Requirement stereotype inheritance structure

The class with the stereotype <<*machine*>> represents the software to be developed (possibly complemented by some hardware). The classes with domain stereotypes (e.g., <<*CausalDomain*>> or <<*BiddableDomain*>>) represent *problem domains* that already exist in the application environment.

In frame diagrams, *interfaces* connect domains, and they contain *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 Fig. 1 the notation Op!E4 means that the phenomena in the set E4 are controlled by the domain Operator. These interfaces are represented as associations, and the name of the associations contain the phenomena and the domain controlling the phenomena.

The associations can be replaced by interface classes, whose operations correspond to phenomena. The interface classes are either controlled or observed by the connected domains, represented by dependencies with the stereotypes <<*controls*>> or <<*observes*>>. Each interface can be controlled by at most one domain. A controlled interface must be observed by at least one domain, and an observed interface must be controlled by exactly one domain.

Problem frames substantially support developers in analyzing problems to be solved. They show what domains have to be considered, and what knowledge must be described and reasoned about when analyzing the problem in depth. Developers must elicit, examine, and describe the relevant properties of each domain. These descriptions form the *domain knowledge*.

The domain knowledge consists of *assumptions* and *facts*. Assumptions are conditions that are needed, so that the *requirements* are accomplishable. Usually, they describe required user behavior. For example, it must be assumed that a user ensures not to be observed by a malicious user when entering a password. Facts describe fixed properties of the problem environment, regardless of how the machine is built.

Domain knowledge and requirements are special statements. A statement is modeled similarly to a SysML requirement [24] as a class with a stereotype. In this stereo-

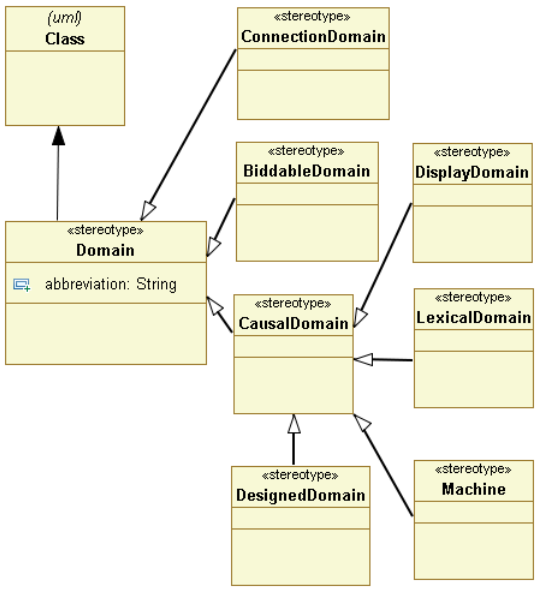

Fig. 3. Domain stereotypes in UML Profile

type a unique identifier and the statement text are contained as stereotype attributes. Fig. 2 shows the stereotype Statement that extends the metaclass Class of the UML metamodel.

When we state a requirement, we want to change something in the world with the machine to be developed. Therefore, each requirement constrains at least one domain. This is expressed by a dependency from the requirement to a domain with the stereotype *<<constrains>>*. Such a constrained domain is the core of any problem description, because it has to be controlled according to the requirements. Hence, a constrained domain triggers the need for developing a new software (the machine), which provides the desired control.

A requirement may refer to several domains in the environment of the machine. This is expressed by a dependency from the requirement to a domain with the stereotype *<<refersTo>>*. The referred domains are also given in the requirements description.

In Fig. 1, the Controlled Domain domain is constrained, because the Control Machine has the role to change it on behalf of user commands for achieving the required Commanded Behaviour.

Jackson distinguishes the domain types *biddable domains* that are usually people, *causal domains* that comply with some physical laws, and *lexical domains* that are data representations. The domain types are modeled by the stereotypes *<<BiddableDomain>>* and *<<CausalDomain>>* being subclasses of the stereotype *<<Domain>>*. A lexical domain (*<<LexicalDomain>>*) is modeled as a special case of a causal domain. 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. They are modeled as classes with the stereotype *<<ConnectionDomain>>*. Connection domains are, e.g., video cameras, sensors, or networks. This kind of modeling allows one to add further domain types,

such as <<*DisplayDomain*>> (introduced in [9]) being a special case of a causal domain. Figure 3 depicts the domain stereotypes defined in our UML Profile.

Other problem frames besides the commanded behavior frame are *required behaviour*, *simple workpieces*, *information display*, and *transformation* [17].

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 (see Fig. 5 for an example). Then, the problem is decomposed into subproblems. If 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, phenomena, and interfaces. The instantiated frame diagram is called a *problem diagram*.

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*>>, and <<*ProblemFrame*>>. These stereotypes extend (some of them indirectly) the meta-class Package in the UML meta-model.

Successfully fitting a problem to a given problem frame means that the concrete problem indeed exhibits the properties that are characteristic for the problem class defined by the problem frame. A problem can only be fitted to a problem frame if the involved problem domains belong to the domain types specified in the frame diagram. For example, the Operator domain of Fig. 1 can only be instantiated by persons, but not for example by some physical equipment like an elevator. Thus, an advantage of using problem frames in requirements engineering is that problems are mapped to well-known problem classes that are practically relevant. Moreover, when using problem frames, one can even hope for more than just a full comprehension of the problem at hand. Since problems fitting to a problem frame share common properties, their solutions will share common properties, too [5]. Thus, problem frames provide pattern-based support not only for problem comprehension, but also for problem solving. For each subproblem, a separate architecture can be developed as described in [5]. These can be merged in a systematic way, see [6].

3 Tool Support

We have developed a tool called UML4PF to support the requirements engineering process sketched in Section 2 as well as subsequent development steps, such as deriving software architectures from problem descriptions. After the developer has drawn some diagram(s) using some EMF-based editor, for example Papyrus UML [3], UML4PF provides him or her with the following functionality:

- It checks if the developed model is 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.

Figure 4 provides an overview of the context of our tool. Gray boxes denote re-used components, whereas white boxes describe those components that we created. Basis is the Eclipse platform [1] together with its plug-ins EMF [2] and OCL [23]. These

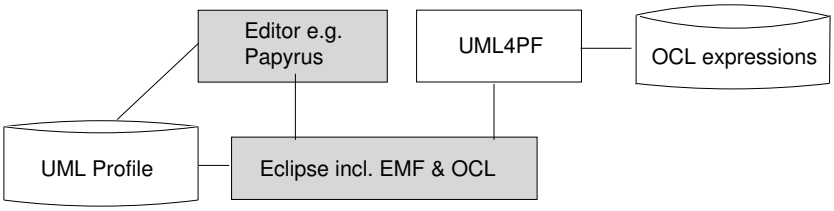

Fig. 4. Tool Realization Overview

plug-ins provide functions to query a model with OCL. Our UML-profile is conceived as an Eclipse plug-in, extending the EMF meta-model. We store the data in the profile in XMI-format. We store all our OCL constraints in one file in XML-format. They are directly checked using the OCL executor, which is part of EMF.

The graphical representation of the different diagram types can be manipulated by using any EMF-based editor. We selected Papyrus UML [3] as it is available as an Eclipse plug-in, open-source, and EMF-based. UML4PF provides additional windows in Eclipse to edit requirements and traceability links as an easy-to-use user interface. The requirements and traceability links are directly stored in the UML model. The graphical representation of the created UML elements is not necessary, but can be added later. The tool is an open source tool under development and is available on demand from the authors.

Listing 1.1 shows an example of an integrity condition. It formalizes the general fact that each statement constrains (see Fig. 2) constrains at least one domain. All classes in the model (Line 1) with the stereotype *<<Statement>>* (accessed by the EMF keyword getAppliedStereotypes) or a specialized statement subtype[3], e.g., *<<Requirement>>*) (Lines 2-5) are selected. For these classes, the dependencies of class (clientDependency) (Line 6) with the stereotype *<<constrains>>* are collected (Line 7). The number of 'constrains' for each class must be bigger than or equal to one (Line 8). In another OCL integrity condition, it is stated that all *<<constrains>>* dependencies must point to domains.

```
1 Class.allInstances()->select(
2 getAppliedStereotypes().name->includes('Statement') or
3 getAppliedStereotypes().general.name -> includes('Statement') or
4 getAppliedStereotypes().general.general.name ->
      includes('Statement') or
5 getAppliedStereotypes().general.general.general.name ->
      includes('Statement'))
6 ->forAll(clientDependency->collect(d |
7           d.oclAsType(Dependency).getAppliedStereotypes().name ->
                includes('constrains'))
8           ->count(true)>=1)
```

Listing 1.1. Statements have at least one Constrains Dependency

[3]The superclass can be accessed by the EMF keyword general. Since the keyword is not recursive, we need to address each of the 3 possible hierarchy levels explicitly.

4 Checking the Correct Instantiation of Problem Frames

This section, we present a number of OCL constraints that can be used to check if a given problem diagram is a correct instantiation of a given problem frame. Such checks are very important, because a software development problem only belongs to the problem class characterized by the problem frame if it really exhibits all characteristics required by the frame. Only then can the solution approaches associated with the problem frame by successfully applied.

We have also defined other OCL constraints (not presented in this paper) that concern the relation between context diagrams and problem diagrams as well as the consistency between problem diagrams and behavioral descriptions, expressed as sequence diagrams. Another paper [14] presents constraints for describing dependability requirements, such as confidentiality, integrity, and reliability. In this paper, we specifically focus on ensuring that problem frames are instantiated correctly.

For each problem diagram, we explicitly state which problem frame it instantiates by using a dependency with the stereotype *<<instanceOf>>*. The OCL expression of Listing 1.2 checks if the stereotype *<<instanceOf>>* is used correctly. To this end, all dependencies in the model (Line 1) with the stereotype *<<instanceOf>>* (accessed by the EMF keyword getAppliedStereotypes) (Line 2) are selected. For these dependencies (Line 3), the source and the target must be a package (checked by the EMF expression oclIsTypeOf(Package) (Lines 4 and 5), the source package has the stereotype *<<ProblemDiagram>>* (Line 6), and the target package has the stereotype *<<ProblemFrame>>* (Line 7).

```
1 Dependency.allInstances()
2 ->select(a |
      a.oclAsType(Dependency).getAppliedStereotypes().name ->
      includes('instanceOf') )
3 ->forAll(d |
4   d.oclAsType(Dependency).source->forAll(oclIsTypeOf(Package))
        and
5   d.oclAsType(Dependency).target->forAll(oclIsTypeOf(Package))
        and
6   d.oclAsType(Dependency).source.getAppliedStereotypes().name ->
        includes('ProblemDiagram') and
7   d.oclAsType(Dependency).target.getAppliedStereotypes().name ->
        includes('ProblemFrame'))
```

Listing 1.2. 'instanceOf'-Dependencies are from ProblemDiagrams to ProblemFrames

If a problem diagram correctly instantiates a problem frame, possible solutions defined for the problem frame can be reused for the concrete problem. For example, corresponding architectural patterns [5] can be applied.

For security-related problems (see, e.g., [15]), we are not allowed to add additional interfaces, whereas for other software development problems, additional elements are allowed to be added to the problem diagram. Therefore, we distinguish between two kinds of instances, namely *strict* and *weak* instances. In the OCL expressions, we only use the predicate weak.

We now present a set of conditions that should evaluate to true if a given problem diagram is a valid instantiation of a given problem frame. These OCL constraints are one of the contributions of this paper. Some of them we have derived from the informal explanations given by Jackson [17], for example conditions 1, 3, 5 and 7 given below. With these conditions (together with the rules given in [16]), we provide the problem frame approach with a formal semantic underpinning. Other conditions express general rules about correctly instantiating patterns, e.g., conditions 2, 3[4] and 6. All conditions are decidable, because they check semantic properties of problem descriptions that are expressed as syntactic properties of the corresponding UML models.

Our UML profile is not an exact match of the problem frame approach, but provides several enhancements. One of them is the distinction between weak and strict instantiations. Condition 5 states how a weak instance is distinguished from a strict one.

We do not claim that the integrity conditions we have defined so far are complete. On the contrary, it is easily possible to identify new conditions and incorporate them into UML4PF. In any case, it is impossible to come up with a set of semantic integrity conditions that is *sufficient* for the correctness of the defined models. However, the conditions constitute *necessary* conditions for the correctness of the defined models. Therefore, a violation of one of the conditions really indicates an error in the development.

For a given problem diagram to be a valid instantiation of a given problem frame, the following conditions should evaluate to true:

1. The domain types of the constrained domains in the problem frame are the same as in the problem diagram.
2. Each domain referred to by the requirement in the problem frame corresponds to a domain in the problem diagram (same domain types).
3. Each connection in the problem frame corresponds to a connection in the problem diagram, i.e., they connect same domain types.
4. For strict (i.e., non-weak) instances, each connection in the problem diagram corresponds to a connection in the problem frame, i.e., they connect the same domain types.
5. The domain types in problem diagrams and problem frames are consistent: the number of domains of each type in the problem frame is equal to the number of this type in the problem diagram. In case of a weak instance, the number of domains of each type in the problem frame is smaller than or equal to the number of this type in the problem diagram.
6. For strict instances, the direction of the interfaces (observed vs. controlled) is the same in the problem diagram and the problem frame. We allow that interfaces are left out.
7. Interfaces cannot be left out if they are controlled by the machine.

In the following, we present the OCL expressions checking a selection of these conditions.

Condition 1. In the OCL expression of Listing 1.3, all dependencies in the model (Line 1) with the stereotype $<<instanceOf>>$ (Line 2) are selected. For these dependencies (Line 3) the parts of the source (the problem diagram) being requirements (Lines 4 and 5) are selected. For these requirements, the dependencies with the stereotype $<<constrains>>$ are selected (Line 6). The target of these dependencies are the

[4] This condition combines a general instantiation rule with a problem-frame-specific rule.

constrained classes, and the bag of their stereotype names (Line 7) must be the same (Line 8) as the bag of stereotype names of constrained domains in the problem frame (Lines 9-13).

```
1  Dependency.allInstances() ->
       select(getAppliedStereotypes().name->includes('instanceOf'))
2  ->forAll(
3    source.oclAsType(Package)
4    .clientDependency -> select(getAppliedStereotypes().name ->
         includes('isPart'))
5    .target -> select(getAppliedStereotypes().name ->
         includes('Requirement')).oclAsType(Class)
6    .clientDependency -> select(getAppliedStereotypes().name ->
         includes('constrains')).
7    target.getAppliedStereotypes().name
8    =
9    target.oclAsType(Package)
10   .clientDependency -> select(getAppliedStereotypes().name ->
         includes('isPart'))
11   .target -> select(getAppliedStereotypes().name ->
         includes('Requirement')).oclAsType(Class)
12   .clientDependency -> select(getAppliedStereotypes().name ->
         includes('constrains')).
13   target.getAppliedStereotypes().name
14 )
```

Listing 1.3. Constrained domain type in ProblemDiagrams and ProblemFrames

Condition 2 can be checked in a similar way.

Condition 3. Line 1 in the OCL expression of Listing 1.4 selects all dependencies with the stereotype <<*instanceOf*>>. For all such dependencies (Line 2), we select all associations (Line 3) connecting classes (Line 4) and all association whose ends (Lines 5 and 6) are part of the problem frame the <<*instanceOf*>>-dependency points to (Lines 7-12). For all such associations in the problem frame (Line 13), we select all associations (Line 14) connecting classes (Line 15) and all associations whose ends (Lines 16 and 17) are part of the problem diagram the <<*instanceOf*>>-dependency comes from (Lines 18-23). We verify in Line 24 that in the selected set of problem diagram associations, an association exists that connects classes with the same stereotypes (Line 25 and 26).

```
1  Dependency.allInstances() -> select(getAppliedStereotypes().name
       -> includes('instanceOf'))
2  ->forAll(inst_of_dep |
3    Association.allInstances()
4    ->select(endType ->forAll(oclIsTypeOf(Class)))
         .oclAsType(Association)
5    ->select(ass |
6      ass.oclAsType(Association).endType->forAll (ass_end |
7        inst_of_dep.oclAsType(Dependency)
8        .target.oclAsType(Package)
9        .clientDependency -> select(getAppliedStereotypes().name
             -> includes('isPart'))
```

```
10        .target -> select(oclIsTypeOf(Class)).oclAsType(Class) ->
              asSet()
11        -> includes(ass_end.oclAsType(Class))
12      )
13    )->forAll(ass_in_pf |
14      Association.allInstances()
15      ->select(endType ->forAll(oclIsTypeOf(Class)))
            .oclAsType(Association)
16      ->select(ass |
17        ass.oclAsType(Association).endType->forAll (ass_end |
18          inst_of_dep.oclAsType(Dependency)
19          .source.oclAsType(Package)
20          .clientDependency -> select(getAppliedStereotypes().name
                -> includes('isPart'))
21          .target->select(oclIsTypeOf(Class)).oclAsType(Class) ->
                asSet()
22          ->includes(ass_end.oclAsType(Class))
23        )
24      )->exists(ass_in_pd |
25        ass_in_pd.endType.oclAsType(Class)
              .getAppliedStereotypes().name
26        -> includesAll (ass_in_pf.endType.oclAsType(Class)
              .getAppliedStereotypes().name)
27      )
28    )
29  )
```

Listing 1.4. Connections in ProblemDiagrams and ProblemFrames are consistent

Condition 4 can be checked in a similar way.

Condition 5. Line 1 in the OCL expression in Listing 1.5 selects all dependencies with the stereotype <<*instanceOf*>>. For these dependencies (Line 2), we define the boolean variable weak as the value of the attribute weak of the dependency (Lines 3-5), we define the bag pf_domain as all domains of the problem frame the dependency points to (Lines 6-10), and we define the bag pd_domains as all domains of the problem diagram (Lines 11-15).

```
1  Dependency.allInstances() ->
       select(getAppliedStereotypes().name->includes('instanceOf')
       )
2  ->forAll( inst_of_dep |
3    let weak: Boolean =
4      inst_of_dep.getValue(inst_of_dep.oclAsType(Dependency)
           .getAppliedStereotypes()
          ->select(name->includes('instanceOf')) ->asSequence()
          ->first(),'weak').oclAsType(Boolean)
5    in
6    let pf_domains: Bag(Class) =
7      inst_of_dep.target.oclAsType(Package)
8      .clientDependency ->
           select(getAppliedStereotypes().name->includes('isPart'))
```

```
9       .target -> select(oclIsTypeOf(Class))
            ->reject(getAppliedStereotypes().name
            ->includes('Requirement')).oclAsType(Class)
10    in
11    let pd_domains: Bag(Class) =
12      inst_of_dep.source.oclAsType(Package)
13      .clientDependency ->
            select(getAppliedStereotypes().name->includes('isPart'))
14      .target -> select(oclIsTypeOf(Class))
            ->reject(getAppliedStereotypes().name
            ->includes('Requirement')).oclAsType(Class)
15    in
16      pf_domains->forAll(domains |
17          domains.getAppliedStereotypes().name ->forAll(stn |
18            (pf_domains->select(getAppliedStereotypes().name
                ->includes(stn))->size() =
19            pd_domains->select(getAppliedStereotypes().name
                ->includes(stn))->size())
20            or (weak and
21              (pf_domains->select(getAppliedStereotypes().name
                    ->includes(stn))->size() <=
22              pd_domains->select(getAppliedStereotypes().name
                    ->includes(stn))->size())
23            )
24          )
25      )
```

Listing 1.5. Domain Types in Problem Diagrams and Problem Frames are Consistent

Using these definitions, we validate for all stereotype names of all problem frame domains (stn, Lines 16-17) that the number of problem frame domains with the stereotype stn is equal to the number of problem diagrams domains with this stereotype name (Lines 18-19), or for weak dependencies (Line 20) that the number of problem frame domains with the stereotype stn is smaller than or equal to the number of problem diagrams domains with this stereotype name (Lines 21-22).

5 Case Study

In this section, demonstrate how checking our OCL constraints helps developers in detecting and eliminating errors in the models they develop. As an example, we use a simplified automatic teller machine (ATM).

The intended environment of the ATM is described using a context diagram as depicted in Fig. 5. It contains the ATM as the machine to be built. In the 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). The Customers can

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

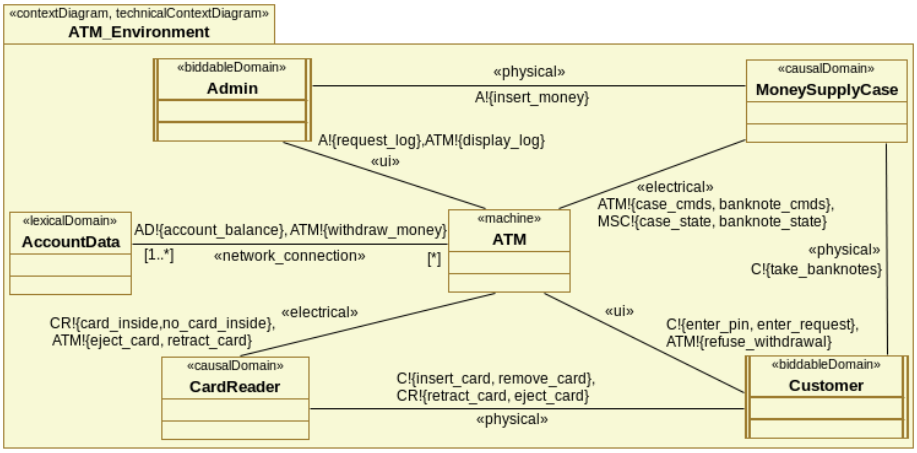

Fig. 5. ATM Context Diagram

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 stereotypes from the $<<domain>>$ stereotype, e.g., the Customer is biddable and the AccountData is lexical. The connections are marked with specializations of the stereotype $<<connection>>$, e.g., a user interface ($<<ui>>$) between Customer and ATM, and a physical connection ($<<physical>>$) between Customer and CardReader.

We consider one subproblem, treating the requirement: *The money supply case supplies the banknotes as requested and retracts the banknotes if the customer does not take them.* An initial problem diagram for this requirement is given in Fig. 6. It is stated to be a (strict) instance of the *required behaviour* problem frame.

Checking the OCL constraints given in Section 4 on our ATM model shows that the validation constraint given in Listing 1.2 is true and that our model also satisfies Condition 2. Conditions 4, 5, and 6 are not satisfied, because the additional domain Customer with its interfaces is introduced. The problem depicted in Fig. 6 is not a security problem, and the Customer should be just included in the problem diagram to describe the relevant context completely. Therefore, we decide to state that the instance is only weak. After this modification, conditions 1, 3, and 7 are still not satisfied. Condition 1 is not true, because the problem diagram contains no constrained class with the stereotype $<<CausalDomain>>$. Condition 3 is not true, because the problem frame connects the machine with a causal domain, whereas in the problem diagram, the machine is connected to a connection domain. To fulfill Conditions 1 and 3, we replace the stereotype $<<ConnectionDomain>>$ with the stereotype $<<CausalDomain>>$. Condition 7 is not fulfilled, because the problem diagram contains no interfaces controlled by the machine. To solve this problem we add MCC!{put_banknote_to_case, open_case, close_case, retract_banknotes_from_case} to the machine interface. The corrected problem diagram is depicted in Fig. 7

The complete case study consists of 4 problem diagram being instances of problem frames, 12 different domains and 33 associations. More than 50 OCL constraints were

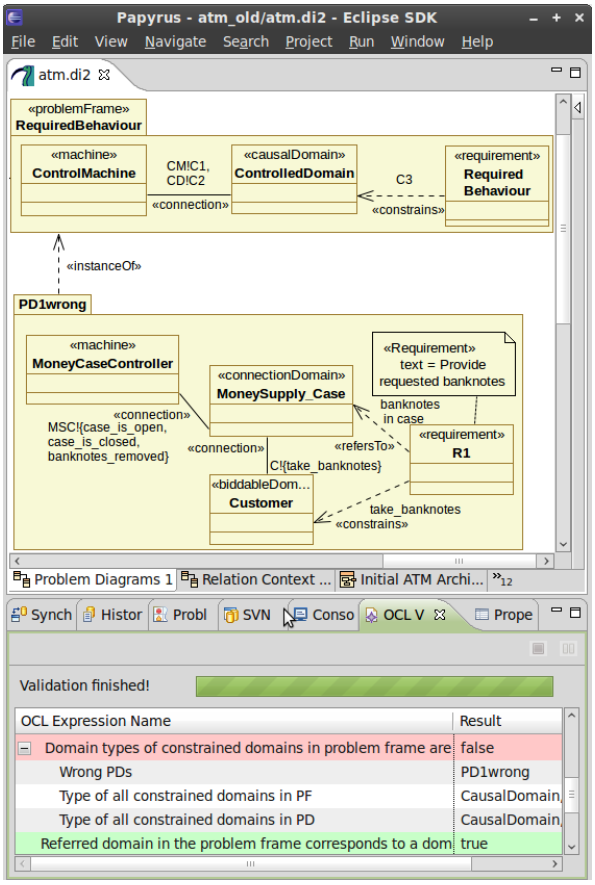

Fig. 6. Erroneous ATM Problem Diagram

checked using our tool, which takes about 30 seconds on a standard computer. As a final result, the ATM model has been successfully validated. Figure 6 shows a screen-shot of a view on the ATM model in Eclipse with our plug-in.

6 Related Work

Lencastre et al. [18] 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 (c.f. Figs. 5 and 11 in [18]).

Hall et al. [13] 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. [20] present a meta-model for problem frame instances. In addition to the diagram elements formalized in our meta-model, they formalize requirements and

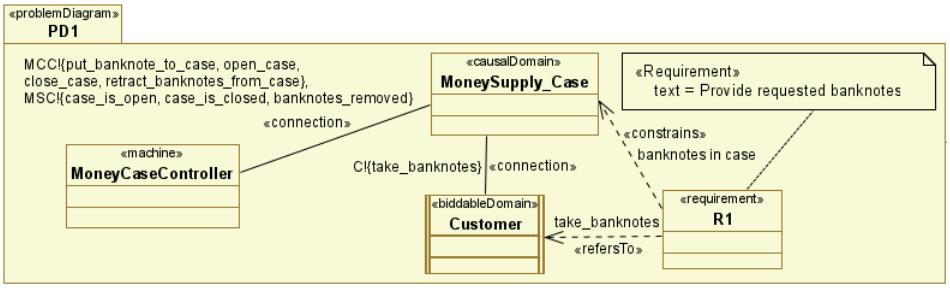

Fig. 7. ATM Control Card Reader Problem Diagram

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.

We agree with Haley [12] on adding cardinality to standard problem frames to enhance the detailing of shared phenomena at the interfaces. In contrast to Haley though, we do not extend the problem frames notation by introducing a new notational element. We adopt the means provided by UML to annotate problem frames in our meta-model instead.

Van Lamsweerde [27] considers the relationships between problem worlds and machine solutions. He makes a distinction between different statement subtypes. In our profile we cover a subset of these statements. Furthermore, he introduces *Satisfaction Arguments*.

Charfi et al. [4] use a modeling framework called *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 have been correctly performed, 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.

Colombo et al. [7] model problem frames and problem diagrams with SysML [22]. 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.

Other important UML profiles are SysML [22] for system engineering and MARTE [26] for model-driven development of Real Time and Embedded Systems (RTES). The UML profile for MARTE (in short MARTE) provides support for specification, design, and verification/validation stages.

7 Conclusions and Future Work

We have shown how a pattern- and model-based requirements engineering process can be equipped with formal elements that allow developers to detect and correct errors in their models. We achieved this by means of a UML profile that allows one to express

the different models being developed during the process in UML. The patterns (in particular, problem frames) can also be expressed with the profile. In this way, one can state conditions that check if a problem frame has been instantiated correctly. Besides these conditions, many other integrity conditions can be expressed in OCL. The approach is tool-supported, which is needed for its practical applicability. In particular, our contributions include the following:

- We have shown how formal and pattern-based software development can be combined.
- We provide a formal underpinning of the problem frame approach.
- We provide tool support for the problem frame approach. This tool support concerns the detection of semantic errors in the requirements engineering process.
- With the defined UML profile, we have provided a basis for a seamless model-and pattern-based development process that covers not only requirements analysis, but also specification and architectural design.
- The defined UML profile can easily be extended to cover not only several phases of software development, but also the specific treatment of dependability requirements [14] and other quality requirements.

Currently, we are augmenting our process to cover also the design phase of the software development process. We have already augmented our profile by architectural elements, and we have defined a number of OCL constraints checking the coherence of problem descriptions (i.e., context and problem diagrams) and architectural diagrams. Moreover, we have taken first steps to support software evolution. In particular, we are introducing traceability links to trace requirements to artifacts developed later, e.g. components in the software architecture.

In summary, our approach has the potential to make software development more rigourous and less error-prone, because semantic integrity conditions can be checked as soon as a new model is set up. Moreover, the use of patterns can be integrated in a natural way.

In the future, we plan to extend our tool to support the identification of missing and interacting requirements. In the long run, we aim to cover all phases of the software development process.

References

1. Eclipse - An Open Development Platform, May 2008. http://www.eclipse.org/.
2. Eclipse Modeling Framework Project (EMF), May 2008. http://www.eclipse.org/modeling/emf/.
3. Papyrus UML Modelling Tool, Jan 2010. http://www.papyusuml.org/.
4. A. Charfi, A. Gamatié, A. Honoré, J.-L. Dekeyser, and M. Abid. 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)*, 2008.
5. C. Choppy, D. Hatebur, and M. Heisel. Architectural patterns for problem frames. *IEE Proceedings – Software, Special Issue on Relating Software Requirements and Architectures*, 152(4):198–208, 2005.
6. C. Choppy, D. Hatebur, and M. Heisel. Component composition through architectural patterns for problem frames. In *Proc. XIII Asia Pacific Software Engineering Conference (APSEC)*, pages 27–34. IEEE, 2006.

7. P. Colombo, V. del Bianco, and L. Lavazza. 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, 2008. ACM.
8. J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.
9. I. Côté, D. Hatebur, M. Heisel, H. Schmidt, and I. Wentzlaff. A systematic account of problem frames. In *Proceedings of the European Conference on Pattern Languages of Programs (EuroPLoP 2007)*. Universitätsverlag Konstanz, 2008.
10. M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison Wesley, 1997.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, 1995.
12. C. B. Haley. Using problem frames with distributed architectures: A case for cardinality on interfaces. *The Second International Software Requirements to Architectures Workshop (STRAW'03)*, May 2003.
13. J. G. Hall, L. Rapanotti, and M. Jackson. Problem frame semantics for software development. *Software and System Modeling*, 4(2):189–198, 2005.
14. D. Hatebur and M. Heisel. A UML profile for requirements analysis of dependable software. In E. Schoitsch, editor, *Proc. SAFECOMP 2010*, LNCS. Springer, 2010. To appear.
15. D. Hatebur, M. Heisel, and H. Schmidt. A pattern system for security requirements engineering. In B. Werner, editor, *Proceedings of the International Conference on Availability, Reliability and Security (AReS)*, IEEE Transactions, pages 356–365. IEEE, 2007.
16. D. Hatebur, M. Heisel, and H. Schmidt. A formal metamodel for problem frames. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, volume 5301, pages 68–82. Springer Berlin / Heidelberg, 2008.
17. M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.
18. M. Lencastre, J. Botelho, P. Clericuzzi, and J. Araújo. A meta-model for the problem frames approach. In *WiSME'05: 4th Workshop in Software Modeling Engineering*, 2005.
19. G. Meszaros. *XUnit Test Patterns, Refactoring Test Code*. Addison-Wesley, 2007.
20. R. Seater, D. Jackson, and R. Gheyi. Requirement progression in problem frames: deriving specifications from requirements. *Requirements Engineering*, 12(2):77–102, 2007.
21. M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
22. SysML Partners. *Systems Modeling Language (SysML) Specification*, 2005. see http://www.sysml.org.
23. "UML Revision Task Force". *OMG Object Constraint Language: Reference*, May 2006. http://www.omg.org/docs/formal/06-05-01.pdf.
24. "UML Revision Task Force". *OMG Systems Modeling Language (OMG SysML)*, November 2008. http://www.omg.org/spec/SysML/1.1/.
25. "UML Revision Task Force". *OMG Unified Modeling Language: Superstructure*, February 2009. http://www.omg.org/docs/formal/09-02-02.pdf.
26. "UML Revision Task Force". *UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE)*, November 2009. http://www.omg.org/docs/formal/formal/2009-11-02.pdf.
27. A. van Lamsweerde. From worlds to machines. In *A Tribute to Michael Jackson*. Lulu Press, 2009.