Automated Checking of Integrity Constraints for a Model- and Pattern-Based Requirements Engineering Method (Technical Report)

Isabelle Côté[1], Denis Hatebur[1,2], Maritta Heisel[1]

[1] University Duisburg-Essen, Faculty of Engineering, Department of Computer Science and Cognitive Science, Workgroup Software Engineering, Germany, {isabelle.cote, denis.hatebur, maritta.heisel}@uni-duisburg-essen.de
[2] Institut für technische Systeme GmbH, Germany, d.hatebur@itesys.de

Abstract. We present a new UML profile serving to support a pattern- and model-based requirements engineering method based on Jackson's problem frames. The UML profile allows us to express the different models being defined during requirements analysis using UML diagrams. In order to automatically perform semantic validations associated with the method, we provide integrity conditions, expressed as OCL constraints. These constraints concern single models as well as the coherence of different models. To provide tool support for the requirements engineering method, we have developed a tool called UML4PF. It is based on the Eclipse development environment, extended by an EMF-based UML tool, in our case, Papyrus. To demonstrate the applicability of our approach, we use the case study of a vacation rentals reservation system.

1 Introduction

Model-based development is a promising approach to develop high-quality software. This is because the development process consists of setting up a sequence of models, which cover different aspects of the software to be developed, but are not independent of each other. Thus, model-based development offers various possibilities to check the coherence of the developed models. Such checks not only concern syntactic properties, but also semantic ones.

Another concept to enhance the quality of software is to re-use previously acquired development knowledge, thus avoiding to re-invent the wheel each time a new software is built. *Patterns* are a means to represent such development knowledge. For each phase of software development, patterns are available. However, currently the use of patterns is not an integral part of model-based development. It is our aim to bring the two approaches (model- and pattern-based software development) together, thus exploiting the advantages of both of them.

In an earlier paper, we have defined a UML meta-model for *problem frames* [**?**]. Problem frames are patterns supporting requirements analysis [**?**]. We provided integrity conditions that allow one to check if a given problem frame is semantically valid. In this paper, we go further and support not only the definition, but also the use of problem frames. In particular, we have defined a UML

profile that allows us to represent the models being set up when analyzing software development problems according to the problem frame-approach in UML. In this way, the use of problem frames can be seamlessly integrated into a model-based software development process, and the models set up during requirements analysis can be re-used and referred to in later development phases.

For analyzing software development problems using problem frames, numerous integrity conditions can be identified. On the one hand, such conditions concern the semantic integrity of single models. For example, human users cannot be obliged by the software to perform certain actions. This means, the software is not allowed to constrain users. On the other hand, integrity conditions may concern the coherence of different models. For example, the simple software development problems obtained by problem decomposition must be valid instances of problem frames. Furthermore, they must correctly refer to the previously described application domain.

We express the identified integrity conditions as OCL constraints, and provide tool support for automatically checking them. Models can be checked immediately after they have been developed. This makes it possible to detect errors early in the development process.

In the following, we introduce the problem frames approach [**?**] and the corresponding UML profile (Sect. 2). Our tool UML4PF is described in Sect. 3. Section 4 describes a selection of the integrity conditions we have identified. Section 5 illustrates the approach by the case study of a web-based vacation rentals system. 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 [**?**], 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.* [**?**]. We describe problem frames using class diagrams extended by stereotypes. 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.

The UML-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 **problem 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 *O!E4* means that the phenomena in the set *E4* are controlled by the

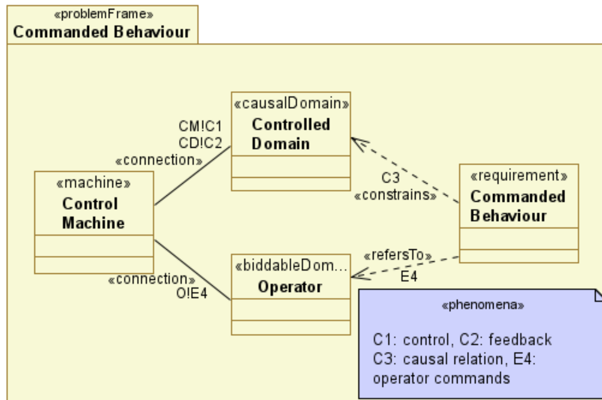

Fig. 1. *Commanded behaviour* problem frame using UML notation

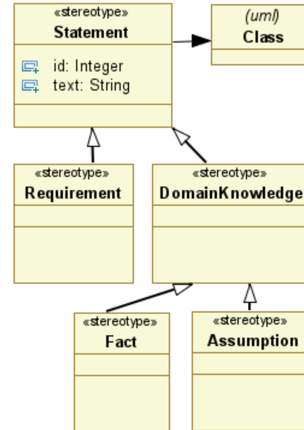

Fig. 2. Requirement stereotype inheritance structure

domain Operator. These interfaces are represented as associations in UML with the stereotype <<*connection*>> or a specialized stereotype, and the name of the associations contain the phenomena and the domain controlling the phenomena.

A association with the stereotype <<*connection*>> or a specialization can also be represented 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.[3]

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. 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 [**?**] as a class with a stereotype. In this stereotype a unique identifier and the statement text are contained as stereotype attributes. Figure 2 shows the new stereotype Statement that extends the meta-class Class of the UML meta-model.

When we state a requirement, we want to change something in the world with the machine to be developed. Therefore, each requirement constrains at

[3] Such conditions form integrity conditions for problem frames, which are implemented in UML4PF.

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.

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 domains referred to 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. This kind of modeling allows us to add further domain types, such as <<*DisplayDomain*>> (introduced in [**?**]) being 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. Connection domains are, e.g., video cameras, sensors, or networks. Whenever a connection domain is introduced, it is necessary to refine or concretize an association. For this purpose, we introduce the stereotypes <<*refines*>> and <<*concretizes*>>. We suggest to use the stereotype <<*refines*>> if a refinement relation is defined, e.g., Z-Refinement [**?**]. Otherwise, we suggest to use the stereotype <<*concretizes*>>.

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 does not take requirements into account (see Fig. 4 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**.

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.

Since the requirements refer to the *environment* in which the machine must operate, the next step consists in deriving a *specification* for the machine (see [**?**] for details). The specification describes the machine and is the starting point for its construction.

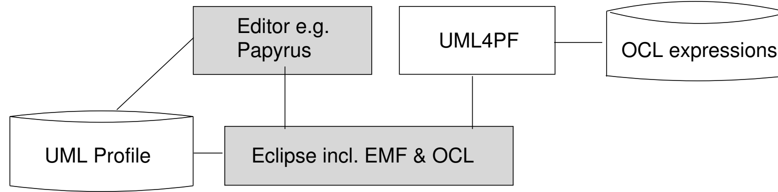

Fig. 3. Tool realization overview

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.

According to the UML superstructure specification [**?**], it is not possible that one UML element is part of several packages. Nevertheless, several UML tools allow one to put the same UML element into several packages within graphical representations. We want to make use of this information from graphical representations and add it to the model (using stereotypes of the profile). Thus, we have to relate the elements inside a package explicitly to the package. This can be achieved with a dependency stereotype <<*isPart*>> from the package to all included elements (e.g., classes, interfaces, comments, associations).

3 Tool Support

We have developed a tool called *UML4PF* to support the requirements engineering method sketched in Section 2 as well as subsequent development steps, such as deriving software architectures from problem descriptions. Figure 3 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 [**?**] together with its plug-ins EMF [**?**] and OCL [**?**]. With OCL it is possible to formally specify constraints over a given model. 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. Listing 1.1 shows an example of such a constraint. It formalizes the general fact that a model contains exactly one context diagram. In Listing 1.1, all packages in the model (line 1) having the stereotype <<*ContextDiagram*>> assigned (accessed by the EMF keyword getAppliedStereotypes in line 2) are selected. The number of packages in the selected set must then be equal to one (line 3).

```
1 Package.allInstances()
2 ->select(getAppliedStereotypes().name
      ->includes('ContextDiagram'))
3 ->size()=1
```

Listing 1.1. Only one context diagram in a project

The functionality of our tool UML4PF comprises the following:

– 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, as well as dependencies with stereotype <<*isPart*>> for all domains and statements being inside a package in the graphical representation of the model.

The graphical representation of the different diagram types can be manipulated by using any EMF-based editor. We selected Papyrus [**?**] 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. Requirements and traceability links are directly stored in the UML model.

4 Checking Integrity Constraints

In this section, we show how the different models can be checked using OCL-constraints. We have identified mainly two classes of constraints. The first class enables us to check the integrity and consistency of single diagrams. The second class allows us to check the consistency of different types of diagrams. The constraints given below in italics mark the selection of constraints given in detail in this paper[4]. The relevant diagram types covered in this paper are: context diagram, problem diagram and problem frame.

4.1 Checking Single Diagrams

We first give expressions that are related to all three diagram types, followed by expressions for context diagrams, and finally constraints that apply to both problem diagrams as well as problem frames.

OCL-Expressions related to all three diagram types:

1. A context diagram or problem diagram or problem frame must contain at least one machine domain.[5]
2. A context diagram/problem diagram/ problem frame only consists of allowed elements, e.g., classes and associations.
3. Domains contained in another domain must be of same type or of a sub-type.
4. Each connection connects at least two domains.
5. Each interface can be controlled by at most one domain.
6. A controlled interface must be observed by at least one domain.
7. Stereotypes <<*CausalDomain*>> (and sub-types) and <<*BiddableDomain*>> are not allowed together for one class.
8. Interfaces can only contain other interfaces.
9. Connection domains have at least one observed and exactly one controlled interface.
10. Each machine controls at least one interface.
11. Dependencies with the stereotypes <<*observes*>> and <<*controls*>> point from domains to interfaces.

[4] The complete presentation of the OCL constraints can be found in [**?**].
[5] In contrast to Jackson, we allow more than one machine domain to be able to model distributed systems.

```
1 Package.allInstances()->select(p |
2    p.oclAsType(Package).getAppliedStereotypes().name
       ->includes('ContextDiagram') or
3   p.oclAsType(Package).getAppliedStereotypes().name
       ->includes('ProblemDiagram') or
4   p.oclAsType(Package).getAppliedStereotypes().name
       ->includes('ProblemFrame')
5 )-> forAll (p |
6   p.clientDependency ->select(getAppliedStereotypes().name
       ->includes('isPart'))
7   .target->select(cd_elem |
8     cd_elem .oclIsTypeOf(Class) and cd_elem .oclAsType(Class)
         .getAppliedStereotypes().name ->includes('Machine')
9   )->size()>=1)
```

Listing 1.2. Context diagram/problem diagram/problem frame must contain at least one machine domain

Listing 1.2 contains the OCL-expression for **Condition 1**: It expresses that the context diagram must contain at least one machine. We first select all packages with the appropriate stereotype, i.e, <<*ContextDiagram*>>, <<*ProblemDiagram*>>, or <<*ProblemFrame*>> (lines 1 to 4). In this set of packages we collect all dependencies of this package (using clientDependency) and select those with the stereotype <<*isPart*>> (line 6). Using the target ends of the dependencies, we collect all elements of the package and select (line 7) those elements (cd_elem) being classes with the stereotype <<*Machine*>> (line 8). The size of the resulting bag must be greater than or equal to one (line 9).

```
1  Package.allInstances() ->select(p |
2  p.oclAsType(Package).getAppliedStereotypes().name
      ->includes('ContextDiagram') or
3  p.oclAsType(Package).getAppliedStereotypes().name
      ->includes('ProblemDiagram') or
4  p.oclAsType(Package).getAppliedStereotypes().name
      ->includes('ProblemFrame')).
5  clientDependency->select(getAppliedStereotypes().name
      ->includes('isPart')).target->forAll(oe |
6      (oe.oclIsTypeOf(Class) and
7          (oe.oclAsType(Class).getAppliedStereotypes().name
              ->includes('Domain') or
8          oe.oclAsType(Class).getAppliedStereotypes().general.
              name ->includes('Domain') or
9          oe.oclAsType(Class).getAppliedStereotypes().general.
              general.name ->includes('Domain') or
10         oe.oclAsType(Class).getAppliedStereotypes().general.
              general.general.name ->includes('Domain') or
11         oe.oclAsType(Class).getAppliedStereotypes().name
              ->includes('Statement') or
12         oe.oclAsType(Class).getAppliedStereotypes().general.
              name ->includes('Statement') or
```

```
13          oe.oclAsType(Class).getAppliedStereotypes().general.
                general.name ->includes('Statement') or
14          oe.oclAsType(Class).getAppliedStereotypes().general.
                general.general.name ->includes('Statement'))
15          ) or
16      oe.oclIsTypeOf(Interface) or
17      (oe.oclIsTypeOf(Association) and
18          (oe.oclAsType(Association).getAppliedStereotypes().name
               ->includes('connection') or
19          oe.oclAsType(Association).getAppliedStereotypes().
                general.name ->includes('connection') or
20          oe.oclAsType(Association).getAppliedStereotypes().
                general.general.name ->includes('connection') or
21          oe.oclAsType(Association).getAppliedStereotypes().
                general.general.general.name
               ->includes('connection'))
22          ) or
23      (oe.oclIsTypeOf(Dependency) and
24          (oe.oclAsType(Dependency).getAppliedStereotypes().name
               ->includes('refersTo') or
25          oe.oclAsType(Dependency).getAppliedStereotypes().name
               ->includes('constrains') or
26          oe.oclAsType(Dependency).getAppliedStereotypes().name
               ->includes('controls') or
27          oe.oclAsType(Dependency).getAppliedStereotypes().name
               ->includes('observes') or
28          oe.oclAsType(Dependency).getAppliedStereotypes().name
               ->includes('isPart') or
29          oe.oclAsType(Dependency).getAppliedStereotypes().name
               ->includes('replaces') or
30          oe.oclAsType(Dependency).getAppliedStereotypes().name
               ->includes('restricts') or
31          oe.oclAsType(Dependency).getAppliedStereotypes().name
               ->includes('modifies') or
32          oe.oclAsType(Dependency).getAppliedStereotypes().name
               ->includes('supplements'))
33          ) or
34      oe.oclIsTypeOf(Comment)
35  )
```

Listing 1.3. The packages only consist of allowed elements

All diagram types rely on the same basic notational elements (see **Condition 2**). However, not all existing notational elements are allowed to be used in the different diagram types. In a problem diagram, for instance, allowed elements are classes, interfaces, associations, dependencies, and comments. In the following, we show an OCL constraint expressing that for some of these elements, only a defined set of stereotypes is allowed. Not allowed in a problem diagram are, e.g. packages, components, or classes without any stereotype. This constraint is shown in Listing 1.3: First, we select all packages that are annotated with the stereotype <<*ProblemDiagram*>> (lines 1-3). Second, we select all the elements

being part of the package *el* (line 4) that satisfy the conditions for allowed elements, i.e., it is a class, an interface, an association, a dependency, or a comment (lines 5, 9, 10, 14, and 18).

– Classes (line 5) being part of the problem diagram package must have a stereotype <<*Domain*>> or a specialized domain stereotype. In line 6, we initialize the variable s, which is a set of type Stereotype, with the stereotypes applied for the class *el*. In line 7, we check that the name of the stereotype is 'Domain' or a sub-type of 'Domain'. Classes may also have the stereotype <<*Statement*>> or a sub-type such as <<*Requirement*>> (line 8).
– For interface classes (line 9) the usable stereotypes are not restricted.
– Any association (line 10) being part of the problem diagram package represents an interface. Therefore, it must have the stereotype <<*connection*>> or must be of a sub-type, e.g., <<*ui*>> for a user interface (lines 11-13).
– The included dependencies (line 14) must be (of the already described) stereotypes <<*observes*>>, <<*controls*>>, <<*refersTo*>>, <<*constrains*>>, or <<*isPart*>>, or of the stereotypes <<*concretizes*>>, <<*refines*>>, <<*replaces*>>, <<*restricts*>>, <<*modifies*>>, or <<*supplements*>> (lines 15-17). The latter stereotypes are not treated in this paper, as they are out of scope.
– For comments (line 18) the usable stereotypes are not restricted.

```
1  Class.allInstances() ->forAll(c |c.oclAsType(Class).member
2    ->forAll(p | (p.oclIsTypeOf(Property) and
         p.oclAsType(Property).type.oclIsTypeOf(Class)) implies
3      p.oclAsType(Property).type.getAppliedStereotypes().name
4        ->forAll(st_name_of_part |
5          (c.getAppliedStereotypes().name
             ->includes(st_name_of_part))  or
6          ( (
7            ( st_name_of_part='ConnectionDomain' or
8              st_name_of_part='BiddableDomain' or
9              st_name_of_part='CausalDomain' or
10             st_name_of_part='DisplayDomain' or
11             st_name_of_part='LexicalDomain' or
12             st_name_of_part='Machine' or
13             st_name_of_part='DesignedDomain' or
14             st_name_of_part='Domain'
15           ) and c.getAppliedStereotypes().name
               ->includes('Domain')
16         ) or
17         (
18           ( st_name_of_part='ConnectionDomain' or
19             st_name_of_part='DisplayDomain' or
20             st_name_of_part='LexicalDomain' or
21             st_name_of_part='Machine' or
22             st_name_of_part='DesignedDomain' or
23             st_name_of_part='CausalDomain'
24           ) and c.getAppliedStereotypes().name
               ->includes('CausalDomain')
```

```
25          ) or
26          (
27            ( st_name_of_part='LexicalDomain' or
28              st_name_of_part='Component' or
29              st_name_of_part='ReusedComponent' or
30              st_name_of_part='Machine'
31            ) and c.getAppliedStereotypes().name
                ->includes('Machine')
32          ) or
33          (
34            ( st_name_of_part='Component' or
35              st_name_of_part='ReusedComponent'
36            ) and c.getAppliedStereotypes().name
                ->includes('Component')
37          )
38        )
39      )
40    )
41  )
```

Listing 1.4. Domains being part of another domain must be of the same type or of a sub-type

To realize **Condition 3**, i.e. that domains being part of another domain must be of the same type or of a sub-type, we must do a case distinction according to the different domain types. Listing 1.4 shows the corresponding for the expression: Note that lexical domains may also be parts of a machine. In line 1, we get all class instances. For those instances we verify that all the class members being properties [6] have either the stereotype names of the contained class (accessed with type and getAppliedStereotypes().name) (lines 3 and 4) included in the list of stereotypes of the class (line 5), i.e. they have the same name or the list of names contain allowed sub-types of Domain (lines 7-15). We have to do this check for <<*CausalDomain*>> (lines 18-24) and <<*Machine*>> (lines 27-31), as well. In lines 34-36 we check that only allowed elements for domain are considered when deriving software architectures in the design step.

```
1  Association.allInstances() ->forAll(a |
       a.oclAsType(Association).memberEnd->size()>=2) and
2  Association.allInstances() ->select(
3    getAppliedStereotypes().name ->includes('connection') or
4    general.getAppliedStereotypes().name
         ->includes('connection') or
5    general.general.getAppliedStereotypes().name
         ->includes('connection') or
6    general.general.general.getAppliedStereotypes().name
         ->includes('connection')
7  ).endType->forAll(s |
8    s.getAppliedStereotypes().name ->includes('Domain') or
```

[6] In EMF, attributes, associations, aggregations and compositions are modeled as properties. Attributes, aggregations and compositions may have the type *Class*.

```
9   s.getAppliedStereotypes().general.name ->includes('Domain')
       or
10  s.getAppliedStereotypes().general.general.name
       ->includes('Domain')
11 )
```

Listing 1.5. Each connection connects at least two domains

Condition 4 is described in Listing 1.5. In line 1 we check that each association `a` has at least 2 association ends. Additionally, in line 2-6 we select all associations with the stereotype <<*connection*>> or a sub-type and check for these associations that connected elements have the stereotype <<*Domain*>> or a sub-type.

```
1 Interface.allInstances() ->forAll(i |
2   i.oclAsType(Interface).getRelationships() ->select(r |
3     r.oclAsType(Relationship).getAppliedStereotypes().name
        ->includes('constrains')
4   )->size()<=1)
```

Listing 1.6. Each interface can be controlled by at most one domain

Condition 5 is described in Listing 1.6. It checks for each interface in the model `i` (line 1) that the number of its relationships (i.e., dependencies) `r` (line 2) with the stereotype <<*constrains*>> (line 3) is smaller than or equal to 1 (line 4).

```
1 Interface.allInstances() ->forAll(i |
2   i.oclAsType(Interface).getRelationships() ->select(r |
3       r.oclAsType(Relationship).getAppliedStereotypes().name
          ->includes('constrains')) ->size()=1
4   implies
5   i.oclAsType(Interface).getRelationships() ->select(r |
6       r.oclAsType(Relationship).getAppliedStereotypes().name
          ->includes('observes')) ->size()>=1
```

Listing 1.7. A controlled interface must be observed by at least one domain

Condition 6 checks for each interface in the model `i` (Listing 1.7, line 1) that if this interface is constrained (lines 2 and 3) this interface is observed by at least one domain (lines 4-6).

```
1 Class.allInstances()->select(cl |
2 let st: Set(Stereotype) =
     cl.oclAsType(Class).getAppliedStereotypes() in
3   ( st.name->includes('BiddableDomain') or
4     st.general.name->includes('BiddableDomain')
5   ) and (
6     st.name->includes('CausalDomain') or
7     st.general.name->includes('CausalDomain') or
8     st.general.general.name->includes('CausalDomain')
9   ) )->size()=0
```

Listing 1.8. Domain cannot be Causal and Biddable

Not all combinations of stereotypes are permitted. For example, the stereotypes <<*CausalDomain*>> (or sub-types) and <<*BiddableDomain*>> are not allowed to be applied together on one class (see **Condition 7**). Hence, we must provide an OCL expression that checks whether this condition is fulfilled. Listing 1.8 depicts the corresponding OCL constraint, which expresses the following: in line 1, all the classes of the model are selected that satisfy the condition stated within the select-statement. In line 2, we gather the set of stereotypes for each class *cl* and assign it to the variable *st*. However, only those classes in *st* should be selected that have the stereotype <<*BiddableDomain*>> or a direct sub-type of <<*BiddableDomain*>> and the stereotype <<*CausalDomain*>> or a sub-type of <<*CausalDomain*>>. Unfortunately, it is not possible to iterate through the different inheritance hierarchies of stereotypes with EMF. Therefore, we must explicitly move to each level of inheritance (keyword *general*). As we currently have three hierarchy levels, we limit our constraints to this number (lines 3-8). Note that if new domain types are to be introduced, this limit may need to be adapted. In line 9, we finally check by comparing the size of the set to 0 whether *st* is empty.

```
1 Interface.allInstances().member
2   ->select(p |
        (p.oclIsTypeOf(Property))).oclAsType(Property).type
3   ->forAll(oclIsTypeOf(Interface))
```

Listing 1.9. Interfaces can only contain other interfaces

Condition 8 checks for each interface if it only contains other interface. In Listing 1.9, lines 1 and 2) all contained elements are selected. We check that the type of these elements is interface (line 3).

```
1 Class.allInstances()->select(oe | (
2 oe.oclAsType(Class).getAppliedStereotypes().
     name->includes('ConnectionDomain'))).oclAsType(Class).
     clientDependency.getAppliedStereotypes().name
     ->includes('observes')->count(true)>=1 and
3 Class.allInstances()->select(oe | (
4 oe.oclAsType(Class).getAppliedStereotypes().
     name->includes('ConnectionDomain'))).oclAsType(Class).
5 clientDependency.getAppliedStereotypes().name
     ->includes('controls')->count(true)>=1
```

Listing 1.10. Connection domains have at least one observed and exactly one controlled interface

Condition 9 checks that all display domains and connection domains (Listing 1.10, lines 1, 3 and 4) have at least one observed (line 2) and exactly one controlled interface (line 5).

```
1 self.allOwnedElements()->select(oe | oe.oclIsTypeOf(Class) and
2 oe.oclAsType(Class).getAppliedStereotypes().name
     ->includes('Machine') )->forAll(
```

```
3 oclAsType(Class).clientDependency.
      getAppliedStereotypes().name
      ->includes('controls')->count(true)>=1)
```

Listing 1.11. Each machine controls at least one interface

Condition 10 checks that each machine (Listing 1.11, lines 1 and 2) controls at least one interface (line 3).

```
 1 Dependency.allInstances()->select(a |
 2 a.oclAsType(Dependency).getAppliedStereotypes().name
      ->includes('observes')  or
 3 a.oclAsType(Dependency).getAppliedStereotypes().name
      ->includes('controls') ) ->forAll(d |
 4   d.oclAsType(Dependency).source.getAppliedStereotypes().name
        ->includes('Domain') or
 5   d.oclAsType(Dependency).source.getAppliedStereotypes().
        general.name->includes('Domain') or
 6   d.oclAsType(Dependency).source.getAppliedStereotypes().
        general.general.name->includes('Domain') )
 7 and
 8 Dependency.allInstances()->select(a |
 9 a.oclAsType(Dependency).getAppliedStereotypes().name
      ->includes('observes')  or
10 a.oclAsType(Dependency).getAppliedStereotypes().name
      ->includes('controls') ).target
11   ->forAll(oclIsTypeOf(Interface))
```

Listing 1.12. Dependencies with the stereotypes observes and controls point from domains to interfaces

Condition 11 checks that dependencies with the stereotypes <<*observes*>> and <<*controls*>> (Listing 1.12, lines 1-3 and 8-10) point from domains or sub-types (lines 4-6) to interfaces (line 11).

OCL-Expressions related to context diagrams:

12. Only one context diagram exists (see Listing 1.1).
13. Context diagrams do not contain requirements.

```
1 Package.allInstances()->select(p |
2 p.oclAsType(Package).getAppliedStereotypes().name
     ->includes('ContextDiagram') ).
3 clientDependency ->select(getAppliedStereotypes().name
     ->includes('isPart')).target->select(oe |
4 oe.oclIsTypeOf(Class) and
     oe.oclAsType(Class).getAppliedStereotypes().name
     ->includes('Requirement'))->size()=0
```

Listing 1.13. Context diagrams do not contain requirements

Condition 13 checks that context diagrams (Listing 1.13, lines 1 and 2) do not contain (line 3) requirements (line 4).

OCL-expressions related to problem diagrams/problem frames:

14. The requirements constrain no machine.
15. The requirements constrain no biddable domain.
16. Packages with the stereotype <<*ProblemDiagram*>> or <<*ProblemFrame*>> must contain at least one requirement.
17. Dependencies with the stereotypes <<*constrains*>> and <<*refersTo*>> point from statements (or sub-types) to domains (or sub-types).

```
1 Dependency.allInstances()
2   ->select(a |
        a.oclAsType(Dependency).getAppliedStereotypes().name
        ->includes('constrains'))
3 ->forAll(not (source.getAppliedStereotypes().name
                    ->includes('Requirement') and
4                target.getAppliedStereotypes().name
                    ->includes('Machine')
5                ) )
```

Listing 1.14. Requirement constrain no machine

Listing 1.14 illustrates the constraint for **Condition 14**: We first select all dependencies with the stereotype <<*constrains*>> (lines 1 and 2). We then check that there is no dependency that starts from <<*Requirement*>> and points to <<*Machine*>> (lines 3 and 4).

```
1 Dependency.allInstances()->select(
2 oclAsType(Dependency).getAppliedStereotypes().name
     ->includes('constrains'))->forAll(
3  source.getAppliedStereotypes()
      .name->includes('Requirement') implies
4  not target.getAppliedStereotypes().
      name->includes('BiddableDomain'))
```

Listing 1.15. Requirements constrain no biddable domain

Condition 15 checks that requirements constrain no biddable domain. Listing 1.15 shows for all dependencies (line 1) with the stereotype <<*constrains*>> (line 2) that if the the source of the dependency is a requirement (line 3), the target is not a biddable domain (line 4).

```
1 Package.allInstances()->select(p |
2 p.oclAsType(Package).getAppliedStereotypes().name
     ->includes('ProblemDiagram') or
3 p.oclAsType(Package).getAppliedStereotypes().name
     ->includes('ProblemFrame') ).
4 clientDependency ->select(getAppliedStereotypes().name
     ->includes('isPart')).target->select(oe |
5 oe.oclIsTypeOf(Class) and
     oe.oclAsType(Class).getAppliedStereotypes().name
     ->includes('Requirement'))->size()>=1
```

Listing 1.16. Packages with the stereotype ProblemDiagram or ProblemFrame must contain at least one requirement

Condition 16 checks that packages with the stereotype <<*ProblemDiagram*>> or <<*ProblemFrame*>> (Listing 1.16, lines 1-3) must contain (line 4) at least one requirement (line 5).

```
1  Dependency.allInstances()->select(a |
2  a.oclAsType(Dependency).getAppliedStereotypes().name
       ->includes('constrains')  or
3  a.oclAsType(Dependency).getAppliedStereotypes().name
       ->includes('refersTo') ) ->forAll(d |
4  d.oclAsType(Dependency).target.getAppliedStereotypes().name
       ->includes('Domain') or
5  d.oclAsType(Dependency).target.getAppliedStereotypes().
       general.name->includes('Domain') or
6  d.oclAsType(Dependency).target.getAppliedStereotypes().
       general.general.name->includes('Domain') )
7  and
8  Dependency.allInstances()->select(a |
9  a.oclAsType(Dependency).getAppliedStereotypes().name
       ->includes('constrains')  or
10 a.oclAsType(Dependency).getAppliedStereotypes().name
       ->includes('refersTo') ) ->forAll(d |
11 d.oclAsType(Dependency).source.getAppliedStereotypes().name
       ->includes('Statement') or
12 d.oclAsType(Dependency).source.getAppliedStereotypes().
       general.name->includes('Statement') or
13 d.oclAsType(Dependency).source.getAppliedStereotypes().
       general.general.name->includes('Statement') )
```

Listing 1.17. Dependencies with the stereotypes constrains and refersTo point from statements (or sub-types) to domains (or sub-types)

Condition 17 checks that dependencies with the stereotypes <<*constrains*>> and <<*refersTo*>> (Listing 1.17, lines 1-3 and lines 8-10) point from statements or sub-types (line 4-6) to domains or sub-types (lines 11-13).

4.2 Checking Relationships between Diagrams

In this section, OCL-constraints for checking the consistency of two different diagram types are presented.

Relationship between context diagram and problem diagrams:

18. Each problem diagram machine is a part of the context diagram machine.
19. All subproblem diagrams are derived from the context diagram by means of decomposition operators. To check this condition, the following aspects must be considered:
    (a) each domain in each of the problem diagrams corresponds to a domain in the context diagram.
    (b) each connection in each of the problem diagrams corresponds to a connection in the context diagram.
    (c) each observed interface in each of the problem diagrams corresponds to an observed interface in the context diagram.

(d) each controlled interface in each of the problem diagrams corresponds to a controlled interfaces in the context diagram.
(e) each domain in the context diagram corresponds to at least one domain in one of the problem diagrams.
(f) each controlled or observed interface of the machine in the context diagram corresponds to at least one interface in one of the problem diagrams.

Hence, it is allowed to leave out domains in one problem diagram, but each domain in the context diagram must be considered in at least one problem diagram.

Listings 18.1-18.3 express that each machines in the problem diagrams is a part of a machine in the context diagram to realize **Condition 18**.

```
1 let m: Set(Class) =
2   Package.allInstances()->select(getAppliedStereotypes().name
        ->includes('ContextDiagram'))->asSequence()->first()
3       .clientDependency.target
4   ->select(getAppliedStereotypes().name ->includes('Machine')
        or
5     getAppliedStereotypes().general.name
          ->includes('Machine'))
6   .oclAsType(Class)->asSet()
7 in
```

Listing 18.1. Machines of Problem Diagrams are Part of the Machines in the Context Diagram: Get Context Diagram Machines

The set of machines m is (line 1) defined by selecting the package with the stereotype <<*ContextDiagram*>>. Since only one context diaram is allowed in the model, we can access this diagram by converting the selected bag of packages into a sequence and taking the first element (line 2). For this package containing the context diagram, we collect the targets of all dependencies (with clientDependency and target in line 3). These dependencies include all dependencies with the stereotype <<*isPart*>> as described in Section 3. To get the machines being part of the package, we select all classes with the stereotype <<*Machine*>> (lines 4 and 5) and all classes with a stereotype derived from the stereotype <<*Machine*>>. The superclass can be accessed by the EMF keyword general. The selected elements with these stereotypes are classes and we can convert the bag of elements into a set of classes (line 6).

```
8   m.oclAsType(Class).member
9   ->select(oclIsTypeOf(Property)).oclAsType(Property).type
10  ->union(
11    m.oclAsType(Class).member
12    ->select(oclIsTypeOf(Property)) .oclAsType(Property).type
13    ->select(oclIsTypeOf(Class)) .oclAsType(Class).member
14    ->select(oclIsTypeOf(Property)) .oclAsType(Property).type
15  )
16  ->select(oclIsTypeOf(Class)).oclAsType(Class)
```

```
17   ->select(getAppliedStereotypes().name ->includes('Machine')
         or getAppliedStereotypes().general.name
         ->includes('Machine')) ->asSet()
```

Listing 18.2. Machines of Problem Diagrams are Part of the Machines in the Context Diagram: Get Parts of Context Diagram Machines

For all machines m we collect all members (e.g., contained classes, connections, ports, operations, properties) (line 8). We select the properties and collect the type of the properties (elements connected with an aggregation or composition or attribute types) in line 9. To these elements we add the elements aggregated or composed indirectly (lines 10-15). We select the elements being classes with the stereotype <<*Machine*>> or a sub-type and remove double classes (with asSet in lines 16 and 17).

```
18   ->includesAll(
19       Package.allInstances()
             ->select(getAppliedStereotypes().name
             ->includes('ProblemDiagram'))
20       .clientDependency.target
21       ->select(getAppliedStereotypes().name
             ->includes('Machine') or
             getAppliedStereotypes().general.name
             ->includes('Machine'))
22       .oclAsType(Class)->asSet()
23   )
```

Listing 18.3. Machines of Problem Diagrams are Part of the Machine in the Context Diagram: Problem Diagram Machines a are Subset of Context Diagram Machines Parts

Listing 18.3 verifes that the machines in the problem diagram are a subset (using includesAll in line 18) of the set of machines being part of the context diagram machine determined in Listing 18.2. The problem diagram machines are retrieved by selecting all packages with the stereotype <<*ProblemDiagram*>> (line 19) and by using the dependencies (with the stereotype <<*isPart*>>) pointing to the classes with the stereotype <<*Machine*>> or a sub-type (lines 20-22). All subproblem diagrams are derived from the context diagram by means of the following decomposition operators (**Condition 19**):

- introduce connection domain
- remove connection domain
- combine domain
- split domain
- concretize interface
- abstract interface
- combine interface
- split interface
- leave out domain

```
1  let cd_domains: Set(Class) =
2    Package.allInstances()->select(getAppliedStereotypes().name
         ->includes('ContextDiagram'))->asSequence()->first()
3    .clientDependency.target
4    ->select(
5      getAppliedStereotypes().name ->includes('Domain') or
6      getAppliedStereotypes().general.name ->includes('Domain')
         or
7      getAppliedStereotypes().general.general.name
         ->includes('Domain') or
8      getAppliedStereotypes().general.general.general.name
         ->includes('Domain'))
9    .oclAsType(Class)->asSet()
10 let cd_contained_domains: Set(Class) =
11    cd_domains.member
12    ->select(oclIsTypeOf(Property)).oclAsType(Property).type
13   ->select(oclIsTypeOf(Class)).oclAsType(Class)->asSet()
14  in
15 let cd_combined_domains: Set(Class) =
16   Class.allInstances() ->select(cl |
17     let cl_members: Set(Class) =
18       cl.member
19       ->select(oclIsTypeOf(Property))
           .oclAsType(Property).type
20       ->select(oclIsTypeOf(Class)) .oclAsType(Class) ->asSet()
21     in
22       cl_members->exists(clm | cd_domains->includes(clm))
23   )
24 in
```

Listing 19.1. Problem diagram domains are consistent to context diagram domains: get context diagram domains

In the OCL-constraint for **Condition 19a** (given in Listing 19.1), we collect all domains that can be found at the end of a dependency in the context diagram and store them in the set cd_domains (lines 1-9). Note that we must use the EMF-keyword general to access the next inheritance level as the keyword is not defined recursively. As we have 4 possible hierarchy levels, we need to apply it three times (see line 8). We also collect the contained domains (cd_contained_domains, lines 10-13). A contained domain is a class related to a context diagram domain by aggregation, composition, or as an attribute type. Combined domains are collected, as well (cd_combined_domains, lines 15-22). A combined domain is a domain that contains at least one context diagram domain.

```
25 let cd_connection_ifs: Set(Interface)=
26   Package.allInstances()->select(getAppliedStereotypes().name
         ->includes('ContextDiagram'))->asSequence()->first()
27   .clientDependency.target
28   ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
         ->asSet()
29 in
```

```
30 let cd_contained_connection_ifs: Set(Interface)=
31   cd_connection_ifs.member
32   ->select(oclIsTypeOf(Property)).oclAsType(Property).type
33   ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
       ->asSet()
34 in
```

Listing 19.2. Problem diagram domains are consistent to context diagram domains: get context diagram connections

We mentioned earlier that it is possible to introduce connection domains by concretizing or refining an interface. To cover this case, it is necessary to have a look at the connections and to identify those dependencies that have the stereotypes <<*refines*>> and <<*concretizes*>>. To obtain these connections, we must collect all the interfaces of the context diagram. Listing 19.2 contains the corresponding OCL-expression. Basically, we follow the same principle as we did for the domains in lines 1-8 of Listing 19.1: we first collect all interfaces included in the context diagram and store them in cd_connection_ifs (lines 25-28). Second, we collect the interfaces contained in other interfaces (lines 30-33) and store them in cd_contained_connection_ifs.

```
35   Package.allInstances()->select(getAppliedStereotypes().name
        ->includes('ProblemDiagram'))
36   ->forAll(pd |
37     pd.clientDependency ->
          select(getAppliedStereotypes().name
         ->includes('isPart')).target
38     ->select(oclIsTypeOf(Class))
39     ->select(
40       getAppliedStereotypes().name ->includes('Domain') or
41       getAppliedStereotypes().general.name
            ->includes('Domain') or
42       getAppliedStereotypes().general.general.name
            ->includes('Domain') or
43       getAppliedStereotypes().general.general.general.name
            ->includes('Domain')).oclAsType(Class)
```

Listing 19.3. Problem diagram domains are consistent to context diagram domains: all domains in problem diagram

We have now selected all the relevant parts of the context diagram. Next, we must collect the domains contained in the problem diagrams to be able to check whether all those domains are existing in the context diagram. To get these domains, we re-use lines 2-9 of Listing 19.1 as lines 35-43 of this expression (see Expression 19.3) with two modifications: first, we replace ContextDiagram by ProblemDiagram and second, we omit the ->asSet() at the end of line 9. Instead, we start checking whether we are able to find matching pairs of domains or concretizes/refines-relations, respectively.

```
44 ->forAll(pd_domain |
45       cd_domains->includes(pd_domain) or
46       cd_contained_domains->includes(pd_domain) or
```

```
47        cd_combined_domains->includes(pd_domain)  or
48        let concr_ifs_of_pd_domain: Set(Interface) =
49          pd_domain.clientDependency->select(
50            getAppliedStereotypes().name
                ->includes('concretizes') or
51            getAppliedStereotypes().name ->includes('refines')
52          ).target.oclAsType(Interface)->asSet()
53        in
54          concr_ifs_of_pd_domain -> exists(if_pd |
55            cd_connection_ifs->includes(if_pd) or
56            cd_contained_connection_ifs->includes(if_pd)
57          ) ) )
```

Listing 19.4. Problem diagram domains are consistent to context diagram domains

Listing 19.4 illustrates the corresponding OCL-expression: for each problem diagram domain (pd_domain, line 44), we check if they are either:

- included directly in the context diagram (line 45) or
- domains contained in a context diagram domain(line 46) or
- combined domains (line 47) or
- connection domains (lines 48-56). A domain is considered to be a connection domain if in the set of interfaces concretized or refined by problem diagram domains (concr_ifs_of_pd_domain) an interface (if_pd, line 54) with the following property exists: the interface if_pd is included in either the set of context diagram interfaces (line 55) or in interfaces contained in context diagram interfaces (line 56). The set of interfaces concretized or refined by problem diagram domains (concr_ifs_of_pd_domain) is defined (line 47) using the problem diagram domains (pd_domain) and their dependencies (line 48) with the stereotype <<*concretizes*>> or <<*refines*>>.

```
1  let cd_domains: Set(Class) =
2    Package.allInstances()->select(getAppliedStereotypes().name
         ->includes('ContextDiagram'))->asSequence()->first()
3    .clientDependency.target
4    ->select(
5      getAppliedStereotypes().name->includes('Domain') or
6      getAppliedStereotypes().general.name->includes('Domain')
         or
7      getAppliedStereotypes().general.general.name
         ->includes('Domain') or
8      getAppliedStereotypes().general.general.general.name
         ->includes('Domain')).oclAsType(Class)->asSet()
9  in
10 let cd_ifs: Set(Interface) =
11   Package.allInstances()->select(getAppliedStereotypes().name
         ->includes('ContextDiagram')) ->asSequence()->first()
12   .clientDependency.target
13   ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
         ->asSet()
14 in
```

```
15 let cd_contained_ifs: Set(Interface) =
16   cd_ifs.member
17   ->select(oclIsTypeOf(Property)).oclAsType(Property).type
18   ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
       ->asSet()
19 in
20 let cd_concr_ifs: Set(Interface)=
21   cd_ifs.clientDependency->select(
22     getAppliedStereotypes().name->includes('concretizes') or
23     getAppliedStereotypes().name->includes('refines')
24   ).target
25   ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
       ->asSet()
26 in
27 let cd_concr_concr_ifs: Set(Interface)=
28   cd_concr_ifs.clientDependency->select(
29                 getAppliedStereotypes().name
                       ->includes('concretizes') or
30                 getAppliedStereotypes().name->includes('refines')
31   ).target
32   ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
       ->asSet()
33 in
34 let cd_conn_doms: Set(Class) =
35   Class.allInstances()->select(cl |
36     cl.clientDependency
37     ->select(
38         getAppliedStereotypes().name->includes('concretizes')
               or
39         getAppliedStereotypes().name->includes('refines'))
40     ->select(target->asSequence()->first().
           oclIsTypeOf(Interface))
41     ->exists(
42       cd_ifs->includes(target.oclAsType(Interface)
             ->asSequence()->first()) or
43       cd_contained_ifs->includes(target.oclAsType(Interface)
             ->asSequence()->first())
44     ))->asSet()
45 in
46 let cd_con_dom_if: Set(Interface) =
47   cd_conn_doms.clientDependency->select(
48                 getAppliedStereotypes().name
                       ->includes('observes') or
49                 getAppliedStereotypes().name
                       ->includes('controls')
50   ).target
51   ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
       ->asSet()
52 in
53   Package.allInstances()->select(getAppliedStereotypes().name
       ->includes('TechnicalContextDiagram') or
```

```
54    getAppliedStereotypes().name  ->includes('ProblemDiagram'))
55   ->forAll(pd_tcd |
56     pd_tcd.clientDependency ->
           select(getAppliedStereotypes().name
           ->includes('isPart')).target
57     ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
           ->asSet()
58     ->forAll(pd_if |
59           cd_ifs->includes(pd_if) or
60           cd_contained_ifs->includes(pd_if) or
61           cd_concr_ifs->includes(pd_if) or
62           cd_concr_concr_ifs->includes(pd_if) or
63           cd_con_dom_if->includes(pd_if) or
64           let pd_concr_ifs: Set(Interface)=
65             pd_if.clientDependency->select(
66                         getAppliedStereotypes().name
                               ->includes('concretizes') or
67                         getAppliedStereotypes().name
                               ->includes('refines')
68             ).target
69             ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
                   ->asSet()
70           in
71             pd_concr_ifs -> exists(if_pd |
                   cd_ifs->includes(if_pd))
72           or
73           let pd_concr_domains: Set(Interface)=
74             pd_if.clientDependency->select(
75                         getAppliedStereotypes().name
                               ->includes('concretizes') or
76                         getAppliedStereotypes().name
                               ->includes('refines')
77             ).target
78             ->select(oclIsTypeOf(Class)).oclAsType(Interface)
                   ->asSet()
79           in
80             pd_concr_domains-> exists(if_pd |
                   cd_ifs->includes(if_pd))
81     )
82   )
```

Listing 20. Subproblems derived from context diagram by means of decomposition operators: connections in problem diagram consistent to connections in context diagram

With **Condition 19b** we check that each connection in each of the problem diagrams corresponds to a connection in the context diagram. We have to consider the operators *introduce connection domain*, *split domain*, *concretize interface*, *abstract interface*, *combine interface* and *split interface*.

To verify this condition, we define the set of domains in the context diagram (cd_domain, Listing 20, lines 1-9), the set of interface in the context di-

agram (cd_ifs, lines 10-13), the set of interfaces contained in these interfaces (cd_contained_ifs, lines 15-18), the set of interfaces concretizing or refining interfaces of the context diagram (cd_concr_ifs, lines 20-25), the set of interfaces concretizing or refining these interfaces (cd_concr_concr_ifs, lines 27-32), and the set of interface controlled or observed by a connection domain (cd_con_dom_if, lines 34-44). With these definitions, we check for each interface being part of each problem diagram and each technical context diagram[7] (lines 53-58) the following conditions:

- the context diagram includes the considered interface (line 59),
- cd_contained_ifs includes the considered interface (line 60),
- cd_concr_ifs includes the considered interface (line 61),
- cd_concr_concr_ifs includes the considered interface (line 62),
- cd_con_dom_if includes the considered interface (line 63),
- interfaces conretized or refined by a problem diagram interface are part of the context diagram (lines 64-71), or
- domains conretized or refined by a problem diagram interface are part of the context diagram (lines 73-80).

```
1  let cd_domains: Set(Class) =
2    Package.allInstances()->select(getAppliedStereotypes().name
         ->includes('ContextDiagram'))->asSequence()->first()
3    .clientDependency.target
4    ->select(
5            getAppliedStereotypes().name->includes('Domain') or
6            getAppliedStereotypes().general.name
                 ->includes('Domain') or
7            getAppliedStereotypes().general.general.name
                 ->includes('Domain') or
8            getAppliedStereotypes().general.general.general.
                 name->includes('Domain'))
9      .oclAsType(Class)->asSet()
10 in
11 let cd_domains_obs_if: Set(Interface) =
12   cd_domains.clientDependency
13   ->select(getAppliedStereotypes().name->includes('observes'))
14   .target.oclAsType(Interface)->asSet()
15 in
16 let cd_contained_domains: Set(Class) =
17   cd_domains.member
18   ->select(oclIsTypeOf(Property)).oclAsType(Property).type
19   ->select(oclIsTypeOf(Class)).oclAsType(Class)->asSet()
20 in
21 let cd_contained_domains_obs_if: Set(Interface) =
22   cd_contained_domains.clientDependency
23   ->select(getAppliedStereotypes().name->includes('observes'))
24   .target.oclAsType(Interface)->asSet()
25 in
26 let cd_combined_domains: Set(Class) =
```

[7] Special context diagram.

```
27    Class.allInstances()
28    ->select(
29      getAppliedStereotypes().name->includes('Domain') or
30      getAppliedStereotypes().general.name ->includes('Domain')
          or
31      getAppliedStereotypes().general.general.name
          ->includes('Domain') or
32      getAppliedStereotypes().general.general.general.
          name->includes('Domain')
33    )
34    ->select(combined_domain_proposal |
35        let combined_domain_proposal_parts : Set(Class) =
36          combined_domain_proposal.member
37            ->select(oclIsTypeOf(Property) and
                oclAsType(Property).type.oclIsTypeOf(Class)).
                oclAsType(Property).type
38            .oclAsType(Class)->asSet()
39        in
40          cd_domains->includesAll(combined_domain_proposal_parts)
              and
41          combined_domain_proposal_parts->size(<>0
42    ).oclAsType(Class)->asSet()
43  in
44  let cd_combined_domains_obs_if: Set(Interface) =
45    cd_combined_domains.clientDependency
46    ->select(getAppliedStereotypes().name->includes('observes'))
47    .target.oclAsType(Interface)->asSet()
48  in
49  let cd_ifs: Set(Interface)=
50    Package.allInstances()->select(getAppliedStereotypes().name
        ->includes('ContextDiagram'))->asSequence()->first()
51    .clientDependency.target
52    ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
        ->asSet()
53  in
54  let cd_contained_ifs: Set(Interface)=
55    cd_ifs.member
56    ->select(oclIsTypeOf(Property)).oclAsType(Property).type
57    ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
        ->asSet()
58  in
59    Package.allInstances()->select(getAppliedStereotypes().name
        ->includes('TechnicalContextDiagram') or
60    getAppliedStereotypes().name->includes('ProblemDiagram'))
61    ->forAll(pd_tcd |
62      pd_tcd.clientDependency ->
          select(getAppliedStereotypes().name
          ->includes('isPart')).target
63      ->select(oclIsTypeOf(Class))
64      ->select(
65            getAppliedStereotypes().name->includes('Domain') or
```

```
66          getAppliedStereotypes().general.name
                ->includes('Domain') or
67          getAppliedStereotypes().general.general.name
                ->includes('Domain') or
68          getAppliedStereotypes().general.general.general.
                name->includes('Domain')).oclAsType(Class)
69    ->forAll(pd_domain |
70        let pd_domain_obs_if: Set(Interface) =
71          pd_domain.clientDependency
72          ->select(getAppliedStereotypes().name
                ->includes('observes'))
73          .target.oclAsType(Interface)->asSet()
74        in
75          cd_domains_obs_if->includesAll(pd_domain_obs_if) or
76          cd_contained_domains_obs_if
                ->includesAll(pd_domain_obs_if) or
77          cd_combined_domains_obs_if
                ->includesAll(pd_domain_obs_if) or
78          (
79            cd_ifs->includesAll(
80                pd_domain.clientDependency->select(
81                  getAppliedStereotypes().name
                      ->includes('concretizes') or
82                  getAppliedStereotypes().name
                      ->includes('refines'))
83                .target->select(oclIsTypeOf(Interface)).
                    oclAsType(Interface)
84            ) and
85            pd_domain.clientDependency->exists(
86                getAppliedStereotypes().name
                    ->includes('observes') and
87                pd_tcd.clientDependency
                    ->select(getAppliedStereotypes().name
                    ->includes('isPart')).target
88                ->select(oclIsTypeOf(Interface))
89                ->includesAll(target
                    ->select(oclIsTypeOf(Interface))
                    .oclAsType(Interface))
90            )
91          ) or (
92            cd_contained_ifs->includesAll(
93                pd_domain.clientDependency->select(
94                  getAppliedStereotypes().name
                      ->includes('concretizes') or
95                  getAppliedStereotypes().name
                      ->includes('refines'))
96                .target->select(oclIsTypeOf(Interface))
                    .oclAsType(Interface)
97            ) and
98            pd_domain.clientDependency->exists(
```

```
99                 getAppliedStereotypes().name
                       ->includes('observes') and
100                pd_tcd.clientDependency
                       ->select(getAppliedStereotypes().name
                       ->includes('isPart')).target
101                ->select(oclIsTypeOf(Interface))
102                ->includesAll(target
                       ->select(oclIsTypeOf(Interface))
                       .oclAsType(Interface))
103            )
104          )
105      )
106  )
```

Listing 21. Subproblems derived from context diagram by means of decomposition operators: observed interfaces in problem diagrams are consistent with observed interfaces in context diagram

Condition 19c covers that observed interface in each of the problem diagrams corresponds to an observed interface in the context diagram. We have to consider the operators *introduce connection domain*, *split domain*, *concretize interface*, *abstract interface*, *combine interface* and *split interface*. To verify this condition, we define the set of domains in the context diagram (cd_domains, Listing 21, lines 1-9), the set of interfaces observed by these domains (cd_domains_obs_if, lines 11-14), the set of domains contained in the context diagram domains (cd_contained_domains, lines 16-19), the set of interfaces observed by these domains (cd_contained_domains_obs_if, lines 21-24), the set of domains combined form context diagram domains (cd_combined_domains, lines 26-42), the set of interfaces observed by these domains (cd_combined_domains_obs_if, lines 44-47), the set of interfaces in the context diagram (cd_ifs, lines 49-52), the set of interfaces being contained in these interfaces (cd_ifs, lines 54-57). With these definitions, we check for each domain being part of each problem diagram and each technical context diagram (lines 59-69) the following conditions. First, we define the set of observed interfaces of problem diagram domains (pd_domains_obs_if) in lines 70-73 and second, we check that:

- the context diagram includes the same observed interfaces (cd_domains_obs_if) as the problem diagram (pd_domains_obs_if) directly (line 75),
- cd_contained_domains_obs_if includes the considered interface (line 76),
- cd_combined_domains_obs_if includes the considered interface (line 77),
- interfaces concretized or refined by a problem diagram interface are part of the context diagram (lines 79-89), or
- interfaces contained in the problem diagram interface are part of the context diagram (lines 92-102).

```
1  let cd_domains: Set(Class) =
2    Package.allInstances()->select(getAppliedStereotypes().name
         ->includes('ContextDiagram'))->asSequence()->first()
3    .clientDependency.target
4    ->select(
```

```
5           getAppliedStereotypes().name ->includes('Domain') or
6           getAppliedStereotypes().general.name
                ->includes('Domain') or
7           getAppliedStereotypes().general.general.name
                ->includes('Domain') or
8           getAppliedStereotypes().general.general.general.name
                ->includes('Domain'))
9     .oclAsType(Class)->asSet()
10  in
11  let cd_domains_cont_if: Set(Interface) =
12    cd_domains.clientDependency
13    ->select(getAppliedStereotypes().name
          ->includes('controls'))
14    .target.oclAsType(Interface)->asSet()
15  in
16  let cd_contained_domains: Set(Class) =
17    cd_domains.member
18    ->select(p | (p.oclIsTypeOf(Property) and
          p.oclAsType(Property).isComposite())).oclAsType(Property)
          .type.oclAsType(Class)->asSet()
19  in
20  let cd_contained_domains_cont_if: Set(Interface) =
21    cd_contained_domains.clientDependency
22    ->select(getAppliedStereotypes().name
          ->includes('controls'))
23    .target.oclAsType(Interface)->asSet()
24  in
25  let cd_combined_domains: Set(Class) =
26    Class.allInstances()
27    ->select(
28      getAppliedStereotypes().name ->includes('Domain') or
29      getAppliedStereotypes().general.name ->includes('Domain')
            or
30      getAppliedStereotypes().general.general.name
            ->includes('Domain') or
31      getAppliedStereotypes().general.general.general.name
            ->includes('Domain')
32    )
33    ->select(combined_domain_proposal |
34          let combined_domain_proposal_parts : Set(Class) =
35            combined_domain_proposal.member
36              ->select(p | (p.oclIsTypeOf(Property) and
                    p.oclAsType(Property).isComposite()))
                    .oclAsType(Property).type
37              .oclAsType(Class)->asSet()
38          in
39            cd_domains->includesAll(combined_domain_proposal_parts)
                and
40            combined_domain_proposal_parts->size()<>0
41    ).oclAsType(Class)->asSet()
42  in
```

```
43 let cd_combined_domains_cont_if: Set(Interface) =
44   cd_combined_domains.clientDependency
45   ->select(getAppliedStereotypes().name
       ->includes('controls'))
46   .target.oclAsType(Interface)->asSet()
47 in
48 let cd_ifs: Set(Interface)=
49   Package.allInstances()->select(getAppliedStereotypes().name
       ->includes('ContextDiagram'))->asSequence()->first()
50   .clientDependency.target
51   ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
       ->asSet()
52 in
53   Package.allInstances()->select(getAppliedStereotypes().name
       ->includes('TechnicalContextDiagram') or
54   getAppliedStereotypes().name ->includes('ProblemDiagram'))
55   ->forAll(pd |
56     pd.clientDependency ->
         select(getAppliedStereotypes().name
         ->includes('isPart')).target
57     ->select(oclIsTypeOf(Class))
58     ->select(
59           getAppliedStereotypes().name ->includes('Domain') or
60           getAppliedStereotypes().general.name
               ->includes('Domain') or
61           getAppliedStereotypes().general.general.name
               ->includes('Domain') or
62           getAppliedStereotypes().general.general.general.name
               ->includes('Domain')).oclAsType(Class)
63     ->forAll(pd_domain |
64         let pd_domain_cont_if: Set(Interface) =
65           pd_domain.clientDependency
66           ->select(getAppliedStereotypes().name
               ->includes('controls'))
67           .target.oclAsType(Interface)->asSet()
68         in
69           cd_domains_cont_if ->includesAll(pd_domain_cont_if)
               or
70           cd_contained_domains_cont_if
               ->includesAll(pd_domain_cont_if) or
71           cd_combined_domains_cont_if
               ->includesAll(pd_domain_cont_if) or
72           (
73             cd_ifs->includesAll(
74                 pd_domain.clientDependency->select(
75                   getAppliedStereotypes().name
                       ->includes('concretizes') or
76                   getAppliedStereotypes().name
                       ->includes('refines'))
77                 .target->select(oclIsTypeOf(Interface))
                     .oclAsType(Interface)
```

```
78             ) and
79             pd_domain.clientDependency->exists(
80                 getAppliedStereotypes().name
                       ->includes('controls') and
81                 pd.clientDependency ->
                       select(getAppliedStereotypes().name
                       ->includes('isPart')).target
82                 ->select(oclIsTypeOf(Interface))
83                 ->includesAll(target
                       ->select(oclIsTypeOf(Interface))
                       .oclAsType(Interface))
84             )
85           )
86
87       )
88   )
```

Listing 22. Subproblems derived from context diagram by means of decomposition operators: controlled interfaces in problem diagrams are consistent with controlled interfaces in context diagram

Condition 19d states that each controlled interface in each of the problem diagrams corresponds to a controlled interface in the context diagram. This is expressed in Listing 22 in the same way as in Listing 21.

```
1  let pd_domains: Set(Class) =
2    Package.allInstances()->select(getAppliedStereotypes().name
         ->includes('ProblemDiagram'))
3    .clientDependency.target
4    ->select(
5          getAppliedStereotypes().name ->includes('Domain') or
6          getAppliedStereotypes().general.name
               ->includes('Domain') or
7          getAppliedStereotypes().general.general.name
               ->includes('Domain') or
8          getAppliedStereotypes().general.general.general.name
               ->includes('Domain'))
9      .oclAsType(Class)->asSet()
10 in
11 let pd_contained_domains: Set(Class) =
12   pd_domains.member
13   ->select(p | (p.oclIsTypeOf(Property) and
         p.oclAsType(Property).isComposite())).oclAsType(Property)
         .type.oclAsType(Class)->asSet()
14 in
15 let connection_domains: Set(Class) =
16   Class.allInstances()->select(
17         getAppliedStereotypes().name ->includes('Domain') or
18         getAppliedStereotypes().general.name
               ->includes('Domain') or
19         getAppliedStereotypes().general.general.name
               ->includes('Domain') or
```

```
20             getAppliedStereotypes().general.general.general.name
                   ->includes('Domain'))
21    ->select(
22             clientDependency.getAppliedStereotypes().name
                   ->includes('refines')   or
23             clientDependency.getAppliedStereotypes().name
                   ->includes('concretizes') )
24    ->select(clientDependency->exists(target
          ->forAll(oclIsTypeOf(Interface))))->asSet()
25 in
26     Package.allInstances()->select(getAppliedStereotypes().name
          ->includes('ContextDiagram') )->asSequence()->first()
27    ->forAll(cd |
28       cd.clientDependency ->
             select(getAppliedStereotypes().name
            ->includes('isPart')).target
29       ->select(oclIsTypeOf(Class))
30       ->select(
31             getAppliedStereotypes().name ->includes('Domain') or
32             getAppliedStereotypes().general.name
                   ->includes('Domain') or
33             getAppliedStereotypes().general.general.name
                   ->includes('Domain') or
34             getAppliedStereotypes().general.general.general.name
                   ->includes('Domain')).oclAsType(Class)
35       ->forAll(cd_dom |
36           pd_domains->includes(cd_dom) or
37           pd_contained_domains->includes(cd_dom) or
38           let cd_contained_doms: Set(Class) =
39               pd_domains.member
40               ->select(p | (p.oclIsTypeOf(Property) and
                     p.oclAsType(Property).isComposite()))
                     .oclAsType(Property).type
41               ->select(oclIsTypeOf(Class))
42               ->select(
43                     getAppliedStereotypes().name
                           ->includes('Domain') or
44                     getAppliedStereotypes().general.name
                           ->includes('Domain') or
45                     getAppliedStereotypes().general.general.name
                           ->includes('Domain') or
46                     getAppliedStereotypes().general.general.
                           general.name->includes('Domain'))
                           .oclAsType(Class)->asSet()
47           in
48             pd_domains->includesAll(cd_contained_doms)
49           or
50           connection_domains->includes(cd_dom)
51       )
52    )
```

Listing 23. Subproblems derived from context diagram by means of decomposition operators: each domain in context diagram consistent to at least one domain in problem diagrams

Condition 19e describes that each domain in the context diagram corresponds to at least one domain in one of the problem diagrams. We consider the operators *remove connection domain*, *combine domain*, and *split domain*. To verify this condition, we define the set of domains in all problem diagrams (pd_domains, Listing 23, lines 1-9), the set of domains contained in these domains(pd_contained-_domains, lines 11-13), and the set of connection domains (domain concretizing or refining an interface) (connection_domains, lines 15-24). With these definitions, we check for each domain being part of the context diagram (lines 26-35) the following conditions:

- it is included in one of the problem diagrams (line 37),
- it is included in the set of domains contained in problem diagram domains (line 38),
- it is combined from at least one problem diagram domains (line-39-49), or
- it is a connection domain (line 50).

```
1  let pd_ifs: Set(Interface)=
2    Package.allInstances()
         ->select(getAppliedStereotypes().name
         ->includes('ProblemDiagram'))
3    .clientDependency.target
4    ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
         ->asSet()
5  in
6  let pd_concr_ifs: Set(Interface)=
7    pd_ifs.clientDependency->select(
8                  getAppliedStereotypes().name
                       ->includes('concretizes') or
9                  getAppliedStereotypes().name
                       ->includes('refines')
10   ).target
11   ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
         ->asSet()
12 in
13 let pd_contained_ifs: Set(Interface) =
14   pd_ifs.member
15   ->select(p |
         (p.oclIsTypeOf(Property))).oclAsType(Property).type
16   ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
         ->asSet()
17 in
18 let connection_domains: Set(Class) =
19   Class.allInstances() ->select(
20             getAppliedStereotypes().name ->includes('Domain') or
21             getAppliedStereotypes().general.name
                   ->includes('Domain') or
```

```
22            getAppliedStereotypes().general.general.name
                  ->includes('Domain') or
23            getAppliedStereotypes().general.general.general.name
                  ->includes('Domain'))
24    ->select(
25            clientDependency.getAppliedStereotypes().name
                  ->includes('refines')  or
26            clientDependency.getAppliedStereotypes().name
                  ->includes('concretizes') )
27    ->select(clientDependency->exists(target
          ->forAll(oclIsTypeOf(Interface))))->asSet()
28 in
29 let connection_domain_ifs: Set(Interface) =
30    connection_domains.clientDependency
31    ->select(
32            getAppliedStereotypes().name ->includes('refines')
                  or
33            getAppliedStereotypes().name
                  ->includes('concretizes') )
34    .target.oclAsType(Interface)
35    .getRelationships() ->select(oclIsTypeOf(Association))
          .oclAsType(Association).endType
          ->select(oclIsTypeOf(Interface)) .oclAsType(Interface)
          ->asSet()
36 in
37    Package.allInstances()
          ->select(getAppliedStereotypes().name
          ->includes('ContextDiagram') ) ->asSequence() ->first()
38    ->forAll(cd |
39       cd.oclAsType(Package).clientDependency ->
             select(getAppliedStereotypes().name
            ->includes('isPart')).target
40       ->select(oclIsTypeOf(Class)).oclAsType(Class)
41       ->select(
42            getAppliedStereotypes().name ->includes('Machine')
                  or
43            getAppliedStereotypes().general.name
                  ->includes('Machine'))
44       .clientDependency -> select(
45            getAppliedStereotypes().name ->includes('observes')
                  or
46            getAppliedStereotypes().name ->includes('controls'))
47       .target.oclAsType(Interface) ->asSet()
48       -> forAll(cd_machine_if |
49           pd_ifs->includes(cd_machine_if) or
50           pd_concr_ifs->includes(cd_machine_if) or
51           pd_contained_ifs->includes(cd_machine_if) or
52           connection_domain_ifs->includes(cd_machine_if) or
53           let cd_machine_contained_ifs: Set(Interface) =
54             cd_machine_if.member
```

```
55          ->select(p |
                (p.oclIsTypeOf(Property))).oclAsType(Property)
                .type
56          ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
                ->asSet()
57        in
58          pd_ifs->includesAll(cd_machine_contained_ifs)
59        or
60        let cd_machine_concr_if: Set(Interface) =
61          cd_machine_if.clientDependency->select(
62              getAppliedStereotypes().name
                  ->includes('concretizes') or
63              getAppliedStereotypes().name
                  ->includes('refines')
64          ).target
65          ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
                ->asSet()
66        in
67          cd_machine_concr_if->exists(i | pd_ifs->includes(i))
68        or
69        let cd_machine_connecting_domain: Set(Class) =
70          cd.oclAsType(Package).clientDependency ->
                select(getAppliedStereotypes().name
                ->includes('isPart')).target
71          ->select(oclIsTypeOf(Class)).oclAsType(Class)
72          ->select(clientDependency->select(
73                getAppliedStereotypes().name
                    ->includes('observes') or
74                getAppliedStereotypes().name
                    ->includes('controls')
75            ).target->includes(cd_machine_if)
76          ).oclAsType(Class) ->asSet()
77        in
78          connection_domains
                ->includesAll(cd_machine_connecting_domain)
79    )
80  )
```

Listing 24. Subproblems derived from context diagram by means of decomposition operators: controlled/observed interfaces of the machine in the context diagram consistent to controlled/observed interfaces in at least one problem

Each (observed and controlled) interface of the machines in the context diagram must be considered in at least on problem diagram (**Condition 19f**). We consider the operators *remove connection domain*, *combine domain*, *concretize interface*, *abstract interface*, *combine interface*, and *split interface*. To verify this condition, we define the set of interfaces in all problem diagrams (pd_ifs, Listing 24, lines 1-4), the set of interfaces refining or concretizing these interfaces (pd_concr_ifs, lines 6-11), the set of interfaces refining or concretizing these interfaces (pd_contained_ifs, lines 13-16), the set of connection domains (domains

concretizing or refining an interface) (connection_domains, lines 18-27), and the set of interfaces concretized or refined by connection domains (pd_contained_ifs, lines 29-36). With these definitions, we check for each interface controlled or observed by context diagram machines (lines 37-48) the following conditions:

- it is included in one of the problem diagrams directly (line 49),
- it is included in the set of interfaces refining or concretizing problem diagram interfaces (line 50),
- it is included in the set of interfaces contained in problem diagram interfaces (line 51),
- it is a connection domain (line 52),
- it is combined from at least one problem diagram interface (lines-53-58),
- it concretizes or refines a problem diagram interface (lines 60-67),
- it is an interface intoduced for a removed connection domain in the context diagram (lines 69-78).

Relationship between problem diagram and problem frame:

20. The domain types of the constrained domains in the problem frame are the same as in the problem diagram.
21. Each domain referred to by the requirement in the problem frame corresponds to a domain in the problem diagram (same domain types).
22. Each connection in the problem frame corresponds to a connection in the problem diagram, i.e., they connect same domain types.
23. 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.
24. 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.
25. Interfaces cannot be left out if they are controlled by the machine.

```
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 25. Constrained domain type in ProblemDiagrams and ProblemFrames

Condition 20 describes that the domain types of the constrained domains in the problem frame are the same as in the problem diagram In the OCL-expression of Listing 25, 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 constrained classes, and the bag of their stereotype names (line 7) must be the same (line 8) as the bag of stereotype names of the 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('refersTo')).
 7    target.getAppliedStereotypes().name
 8    ->includesAll(target.oclAsType(Package)
 9      .clientDependency ->select(getAppliedStereotypes().name
            ->includes('isPart'))
10      .target ->select(getAppliedStereotypes().name
            ->includes('Requirement')).oclAsType(Class)
11      .clientDependency ->select(getAppliedStereotypes().name
            ->includes('refersTo')).
12      target.getAppliedStereotypes().name
13  ))
```

Listing 26. Referred to domain type in ProblemDiagrams and ProblemFrames

In **Condition 21** we state that each domain referred to by the requirement in the problem frame corresponds to a domain in the problem diagram (same domain types). In Listing 28 we check that each referred domain in the problem frame corresponds to a domain in the problem diagram. In this expression the string 'constrains' is replaced by 'refersTo'. We allow additional referred domains is a problem diagram. Therefore, the stereotype names of the referred classes in the problem diagram (line 7) must include all (line 8) names in the bag of stereotype names of referred domains in the problem frame (lines 9-13).

```
 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 27. Connections in ProblemDiagrams and ProblemFrames are consistent

Condition 22 describes that each connection in the problem frame corresponds to a connection in the problem diagram, i.e., they connect same domain types: line 1 in the OCL-expression of Listing 27 selects all dependencies with the stereotype <<*instanceOf*>>. For all those 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 (lines 25 and 26).

```
1  Dependency.allInstances()
       ->select(getAppliedStereotypes().name
       ->includes('instanceOf') )
2  ->forAll(inst_of_dep |
3    not
4      inst_of_dep.getValue(inst_of_dep.oclAsType(Dependency)
           .getAppliedStereotypes()
           ->select(name->includes('instanceOf'))
           ->asSequence()->first(),'weak').oclAsType(Boolean)
5    implies
6    Association.allInstances()
7    ->select(endType->forAll(oclIsTypeOf(Class)))
         .oclAsType(Association)
8    ->select(ass |
9    ass.oclAsType(Association).endType ->forAll (ass_end |
         inst_of_dep.oclAsType(Dependency)
10     .source.oclAsType(Package)
11     .clientDependency ->select(getAppliedStereotypes().name
           ->includes('isPart')).target->select(oclIsTypeOf(Class))
           .oclAsType(Class)->asSet()
12     ->includes(ass_end.oclAsType(Class)))
13   )->forAll(ass_in_pd |
14   Association.allInstances()
15   ->select(endType->forAll(oclIsTypeOf(Class)))
         .oclAsType(Association)
16   ->select(ass |
17   ass.oclAsType(Association).endType ->forAll (ass_end |
         inst_of_dep.oclAsType(Dependency)
18     .target.oclAsType(Package)
19     .clientDependency ->select(getAppliedStereotypes().name
           ->includes('isPart')).target->select(oclIsTypeOf(Class))
           .oclAsType(Class)->asSet()
20     ->includes(ass_end.oclAsType(Class)))
21   )->exists(ass_in_pf |
22       ass_in_pd.endType.oclAsType(Class)
             .getAppliedStereotypes().name
23       ->includesAll (
24         ass_in_pf.endType.oclAsType(Class)
               .getAppliedStereotypes().name
25         )
26     )
27   )
28 )
```

Listing 28. Connections in ProblemFrames and ProblemDiagrams are consistent

Also the opposite direction is considered for instances stated being not weak (**Condition 23**). In that case all connection in problem diagrams are checked to

correspond to a connection in the instantiated problem frame, i.e., they connect the same domain types. The OCL expression is similar to Listing 27; additional the conditions is only checked if the <<*instanceOf*>>-dependency is not weak, and the target and the source of the dependency are exchanged.

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

Listing 29. Domain Types in Problem Diagrams and Problem Frames are Consistent

In **Condition 24** the number of domains of each type in the problem frame is equal to the number of this type in the problem diagram needs to be checked: line 1 in the OCL-expression in Listing 29 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 pf_domains: Bag(Class) =
4      inst_of_dep.target.oclAsType(Package)
5      .clientDependency ->select(getAppliedStereotypes().name
          ->includes('isPart'))
6      .target ->select(oclIsTypeOf(Class))
          ->reject(getAppliedStereotypes().name
          ->includes('Requirement')).oclAsType(Class)
7    in
8    let pd_domains: Bag(Class) =
9      inst_of_dep.source.oclAsType(Package)
10     .clientDependency ->select(getAppliedStereotypes().name
          ->includes('isPart'))
11     .target ->select(oclIsTypeOf(Class))
          ->reject(getAppliedStereotypes().name
          ->includes('Requirement')).oclAsType(Class)
12   in
13   let pf_ifs: Set(Interface) =
14     inst_of_dep.target.oclAsType(Package)
15     .clientDependency ->select(getAppliedStereotypes().name
          ->includes('isPart'))
16     .target
          ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
          ->asSet()
17   in
18   let pd_ifs: Set(Interface) =
19     inst_of_dep.source.oclAsType(Package)
20     .clientDependency ->select(getAppliedStereotypes().name
          ->includes('isPart'))
21     .target
          ->select(oclIsTypeOf(Interface)).oclAsType(Interface)
          ->asSet()
22   in
23     pf_domains->select(getAppliedStereotypes().name
          ->includes('Machine'))
24     .clientDependency ->select(getAppliedStereotypes().name
          ->includes('controls'))
25     ->select(pf_ifs->includesAll(target.oclAsType(Interface)))
          ->asSet() ->size ()
26     =
27     pd_domains->select(getAppliedStereotypes().name
          ->includes('Machine'))
```

```
28       .clientDependency ->select(getAppliedStereotypes().name
             ->includes('controls'))
29       ->select(pd_ifs->includesAll(target.oclAsType(Interface)))
             ->asSet() ->size ()
30 )
```

Listing 30. Direction of interfaces in ProblemDiagrams and ProblemFrames is consistent

Condition 25 describes that interfaces cannot be left out if they are controlled by the machine. Listing 30 containes the OCL-expression for this: We check that the number of controlled interfaces of each domain in each problem frame with the stereotype <<*machine*>> (lines 23-25) is equal to (line 26) the number of controlled interfaces of each domain in each problem diagram with the stereotype <<*machine*>> (lines 27-29).

In this paper, we presented a total of `30` OCL-constraints (25 plus the 5 sub-constraints). `17` are used to check the integrity of a single diagram type and `13` to check the consistency of different diagram types. We additionally defined some constraints restricting the use of dependency stereotypes and checking the generated model elements. Until now, we have defined about 50 constraints covering requirements analysis.

(R01)	A staff member can make holiday offers available.
(R02)	A guest can browse available holiday offers.
(R03)	A guest can book available holiday offers, which then are reserved.
⋮	…
(R09)	A staff member can browse reserved holiday offers.

Table 1. Subset of requirements for Vacation Rentals

5 Case Study

We use a simple vacation rentals system based on a superset of the requirements given in Tab. 1 as case study to illustrate our approach. As described in Sect. 2, the intended environment of the vacation rentals system (VR) is described using a context diagram (see Fig. 4). It contains VacationRentals as the machine domain. In the environment, we can find the StaffMember responsible for creating and maintaining the holiday offers contained in HolidayOffer. The Guests can interact with the VacationRentals in the following way. They can browse available holiday offers (browseHolidayOffers) and book a holiday offer (bookHolidayOffer). Furthermore, StaffMembers and Guests can interact with VacationHome and Bank.

We use multiplicities to express, e.g., that several Guests can interact with the one VacationRentals. The fact that the HolidayOffers belong to VacationRentals is expressed by composition. The different domains are annotated with the appropriate stereotypes from the <<*domain*>> stereotype , e.g., Guest is biddable and HolidayOffer is lexical. The connections are marked with the appropriate stereotype <<*connection*>> or specializations of this stereotype, e.g., a user interface (<<*ui*>>) between Guest and VacationRentals. When we check the validation conditions relevant for the context diagram (see Sect. 4.1), all pass.

After that, we continue and decompose the overall problem into subproblems. One of the subproblems is given in Fig. 5. We see that we introduced the Webpage to display the output to the guest. After completing the decomposition, we check the validation conditions for problem diagrams (see Sect. 4.1). All conditions pass. Subsequently, we check the conditions ensuring that the context diagram and the problem diagrams are consistent (see Sect. 4.2). All conditions pass, except for

– each domain in the problem diagrams corresponds to a domain in the context diagram (Condition 19a) and
– each interface in the problem diagrams corresponds to an interface in the context diagram (Condition 19b)

The failure is indicated at the bottom of Fig. 4. Furthermore, we can see that in both cases the domain Webpage (and its corresponding interfaces) causes the failure. We know that the domain Webpage was introduced during the decomposition. It must therefore concretize the interface VR!{Invoice, results} between the machine VacationRentals and the domain Guest. However, this has been forgotten to indicate during the decomposition. In order to resolve this problem, we must do the following:

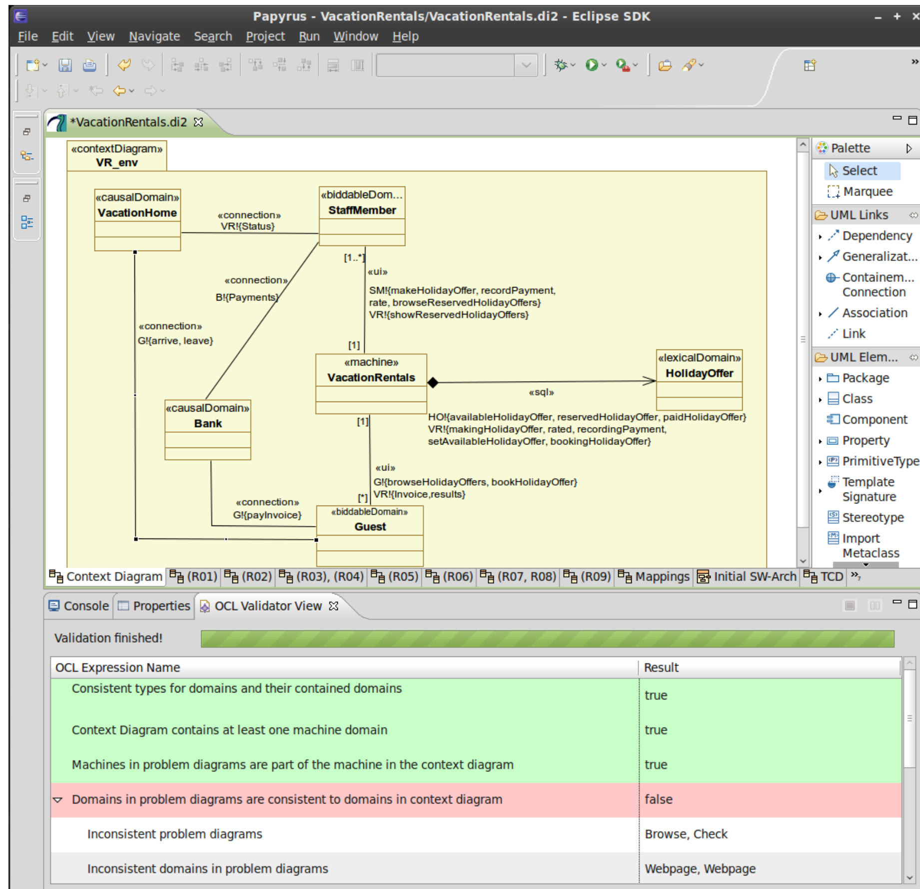

Fig. 4. Vacation Rentals: Context diagram and Validator (Screenshot)

- create an interface class, namely VR!{Invoice} and state that it belongs to the interface VR!{Invoice, results} (see left-hand side of Fig. 6)
  This is necessary, as the Webpage only addresses the phenomenon results of the interface VR!{Invoice, results}
- create a *concretizes*-dependency between the class Webpage and the interface VR!{results} (see right-hand side of Fig.6).

We re-check the conditions and see that now all conditions pass.

6 Related Work

Other authors have also worked on further developing the problem frame approach, mostly by providing a meta-model, defining a semantics, or adding notational elements.

Lencastre et al. [?] 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. Hence, the OCL integrity

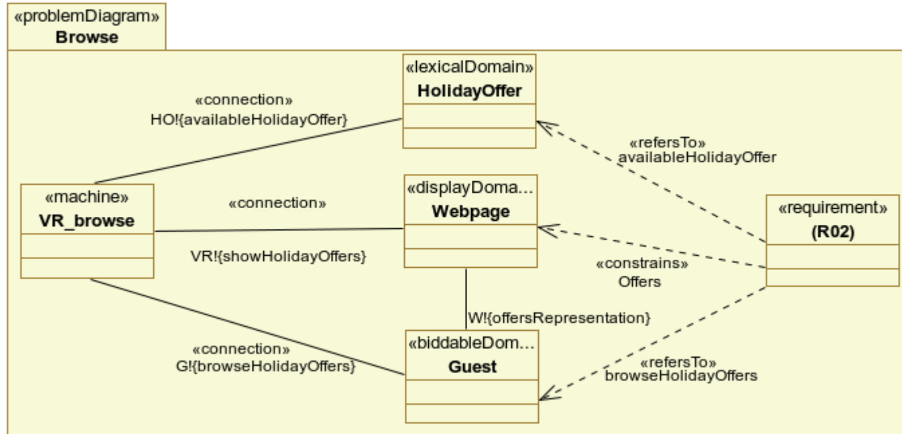

Fig. 5. Vacation Rentals: Problem diagram for R02 (Browse)

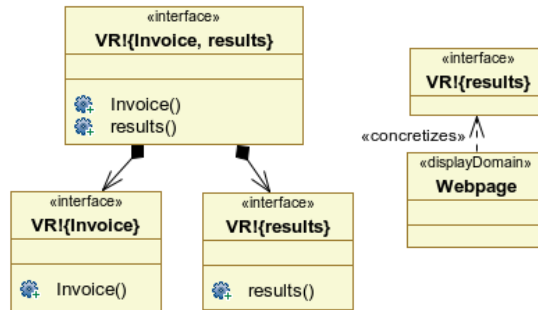

Fig. 6. Vacation Rentals: Mapping of interfaces

conditions of our meta-model are not considered in their meta-model. 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 [?]).

Hall et al. [?] provide a formal semantics for the problem frame approach. They introduce a formal specification language to describe problem frames and problem diagrams. As compared to our meta-model, their approach does not consider integrity conditions.

Seater et al. [?] present a meta-model for problem frame instances. In addition to the diagram elements formalized in our meta-model, they formalize requirements and specifications. Consequently, their integrity conditions ("well-formedness 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 [?] 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 [?] 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. [?] 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. [?] model problem frames and problem diagrams with SysML. 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.

We are not aware of any other tools supporting the work with problem frames on the semantic level, as does UML4PF.

7 Conclusion and Perspectives

We have shown how to seamlessly integrate patterns (in particular, problem frames) into a model-based software engineering process (in particular, the requirements analysis phase) using UML. This combination of model- and pattern-based development allows us to formally express numerous semantic integrity conditions on the developed models in OCL. An accompanying tool helps to draw the respective diagrams and allows developers to automatically check the integrity conditions. It helps to detect errors early in the development process and to maintain a coherent set of models at all times.

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 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.