A Formal Metamodel for Problem Frames

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

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

Abstract. Problem frames are patterns for analyzing, structuring, and characterizing software development problems. This paper presents a formal metamodel for problem frames expressed in UML class diagrams and using the formal specification notation OCL. That metamodel clarifies the nature of the different syntactical elements of problem frames, as well as the relations between them. It provides a framework for syntactical analysis and semantic validation of newly defined problem frames, and it prepares the ground for tool support for the problem frame approach.

1 Introduction

It is a widely accepted opinion in the software engineering community that reusing software development knowledge helps to avoid errors and to speed up the development of a software product. One promising attempt that enables software engineers to systematically construct software using a body of accumulated knowledge are *patterns*.

Patterns have been introduced on the level of detailed object oriented design [13]. Today, patterns are defined for different software development activities. *Problem Frames* [20] are patterns that classify software development *problems*. *Architectural styles* are patterns that characterize software architectures [8]. They are also called "architectural patterns". *Design Patterns* are used for finer-grained software design, while *frameworks* [11] are considered as less abstract, more specialized. Finally, *idioms* are low-level patterns related to specific programming languages [8], and are sometimes called "code patterns".

It is also acknowledged that the first steps of software development are essential for the success of a software development project, because it is important to eliminate any source of error as early as possible. We believe that it is of particular importance to use patterns already in the requirements analysis phase of the software development life-cycle, as is also advocated by, e.g., Fowler [12] and Sutcliffe et al. [28, 29].

Jackson [19, 20] proposes the concept of problem frames for presenting, classifying, and understanding software development problems. A problem frame is a characterization of a class of problems in terms of the considered requirement, their main components (*domains*), and the connections between these components (*interfaces*, consisting of *shared phenomena*). Once a problem is successfully fitted to a problem frame, its most important characteristics are known.

Jackson defines five basic problem frames, as well as some variants of them. These frames are distinguished in the number and characteristics of the involved domains. Jackson does not claim that these frames are complete in the sense that each software development problem can be decomposed in such a way that all derived subproblems fit to one of his problem frames.

Several new problem frames have been developed, such as the frames presented in [10], architectural frames (combinations of architectural styles and problem frames) [24], a dynamic and a static information frame [21], security problem frames [15, 16, 17], frames for database-related problems [9], and *HCI*Frames [25, 27], which focus on usability problems.

Currently, the notation and semantics of problem frames are not defined *rigorously*. Under these circumstances, it is difficult to decide whether a new problem frame contains errors or contradictions. Some problem frames cannot make sense, e.g., because they would require users to be constrained, which contradicts their domain characteristics.

In this paper, we present a *formal metamodel for problem frames*, which contributes to an unambiguous comprehension of the problem frame approach. This metamodel is expressed in a *Unified Modeling Language* (UML) [31] class model, and the formal specification notation *Object Constraint Language* (OCL) [30]. It clarifies the nature of the different constituents of problem frames, as well as the relations between them.

As a consequence of formalizing the syntax of problem frames and their syntactical elements, the metamodel practically helps designing new problem frames. We equipped the metamodel with a number of *integrity conditions*, that allow one to verify that a frame is valid according to the metamodel. Furthermore, the metamodel provides a basis for *tool support* for the problem frame approach, and it prepares the ground for integrating a problem analysis phase based on problem frames into software development processes.

In the following Sect. 2, we present the problem frames proposed by Jackson [20]. In Sect. 3, we introduce our formal metamodel for the problem frame approach. In Sect. 4, we show how tool support for the problem frame approach can be provided using our formal metamodel, and in Sect. 5, we create and check two sample problem frames with the tool. Section 6 discusses related work, and we conclude in Sect. 7.

2 Problem Frames

Problem frames are a means to describe software development problems. They were invented by Michael Jackson [20], who describes them as follows: *"A problem frame is a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirement."* Problem frames are described by *frame diagrams*, which basically consist of rectangles, a dashed oval, and different links between them (see frame diagram in Fig. 1). The task is to construct a *machine* that improves the behavior of the environment in which it is integrated in.

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

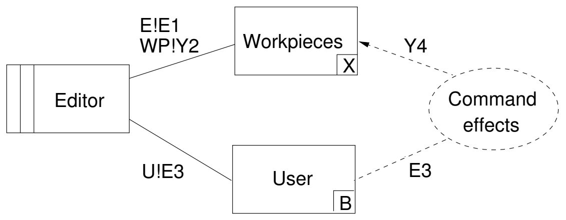

Fig. 1. *Simple workpieces* problem frame

Biddable domains "*A biddable domain usually consists of people. The most important characteristic of a biddable domain is that it's physical but lacks positive predictable internal causality. That is, in most situations it's impossible to compel a person to initiate an event: the most that can be done is to issue instructions to be followed.*"
Biddable domains are indicated by B, and they are always *given*.

Causal domains "*A causal domain is one whose properties include predictable causal relationships among its causal phenomena.*"
Often, causal domains are mechanical or electrical equipment. They are indicated with a C in frame diagrams. Their actions and reactions are predictable. Thus, they can be controlled by other domains. Causal domains can be *given* or *designed*.

Lexical domains "*A lexical domain is a physical representation of data – that is, of symbolic phenomena. It combines causal and symbolic phenomena in a special way. The causal properties allow the data to be written and read.*"
Lexical domains are indicated by X. They are used for data representation purposes. This type of domains can be *given* or *designed*.

A rectangle with a double vertical stripe denotes the machine to be developed, and *requirements* are denoted with a dashed oval. The connecting lines between domains represent interfaces that consist of *shared phenomena*. Shared phenomena may be events, operation calls, messages, and the like. They are observable by at least two domains, but controlled by only one domain, as indicated by an exclamation mark. For example, in Fig. 1 the notation U!E3 means that the phenomena in the set E3 are controlled by the domain User.

A dashed line represents a requirement reference, and an arrow indicates that the requirement *constrains* a domain. If a domain is constrained by the requirement, we must develop a machine, which controls this domain accordingly. In Fig. 1, the Workpieces domain is constrained, because the Editor changes it on behalf of user commands to satisfy the required Command effects.

Software development with problem frames proceeds as follows: first, the environment in which the machine will operate is represented by a *context diagram*. Like a frame diagram, a context diagram consists of domains and interfaces. However, a context diagram contains no requirements, and it is not shown which domain is in control of the shared phenomena. Then, the problem is decomposed into subproblems. If ever possible, the decomposition is done in such a way that the subproblems fit to given problem frames. To fit a subproblem to a problem frame, one must instantiate its frame diagram, i.e., provide instances for its domains, interfaces, and requirement. The instantiated frame diagram is called a *problem diagram*. For example, a requirement such as "Online forms have to be provided that can be filled out by web users" can be used for instantiating the workpieces frame of Fig. 1.

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 [22] for details). The specification describes the machine and is the starting point for its development.

3 A Formal Metamodel for Problem Frames

Based on Jackson's descriptions, we have developed a *formal metamodel for problem frames*. The benefits of such a metamodel are:

- Gain an *unambiguous comprehension* of the problem frame approach.
- *Clarify the syntax* of problem frames and their constituents.
- Provide a framework for *syntactical analysis and validation* of problem frames. This allows one to identify erroneous problem frames.
- Support the development of new problem frames.
- Provide a basis for *tool support* for the problem frame approach by an Eclipse [1] application (see Sect. 4 for details).
- Prepare the ground for integrating a problem analysis phase based on the problem frames approach into software development processes (see [16, 17] for details).

According to Jackson, the syntactical elements of a problem frame can be divided into several categories and subcategories. For example, a domain is either a biddable, causal, or lexical domain. For this reason, a formal method that supports object-orientation is appropriate to formalize problem frames and their syntactical elements. We use a UML [31] class model, and the formal specification notation OCL [30] to construct our formal metamodel for problem frames. The combination of a UML class model and a set of OCL expressions enables us to rapidly provide tool support for the problem frame approach.

In the following, we present the class model in Sect. 3.1, and the set of integrity conditions, expressed in OCL, in Sect. 3.2.

3.1 UML Class Model

In our class model, each syntactical element of a problem frame is represented as a class. Relations between the different elements are expressed using inheritance and associations. The associations are equipped with multiplicities, which express integrity conditions about the minimal or maximal number of occurrences of the syntactical elements in a problem frame. The association ends have names, so that OCL expressions about the objects that participate in an association can be specified (see Sect. 3.2).

Domains have **names** and **abbreviations**, which are used to define interfaces. Hence, the class Domain has the attributes name and abbreviation of type string. According to Jackson, domains are either **designed**, **given**, or **machine** domains. These facts are modeled by the boolean attibutes isGiven and isMachine in Fig. 2.

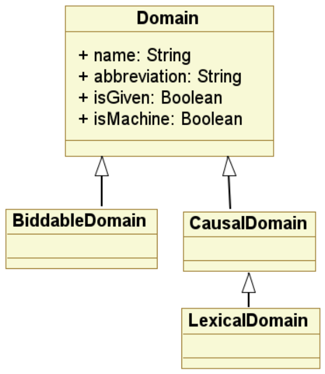

Fig. 2. Inheritance structure of different domain types

Jackson also distinguishes **biddable**, **causal**, and **lexical domains**. Therefore, we introduce the subclasses BiddableDomain, CausalDomain, and LexicalDomain of the class Domain. A lexical domain is a special case of a causal domain. This kind of modeling allows to add further domain types, such as *display domains* as introduced in [10].

Interfaces model connections between domains. The symbol "!" with a prefixed abbreviation of a domain name indicates that the domain *controls* certain **phenomena** contained in the interface. Therefore, we introduce the class Phenomenon, and its subclass InterfacePhenomenon in Fig. 3. Phenomena have **names**. Therefore, the class Phenomenon has an attribute name of type string. Jackson [20, 14] distinguishes **symbolic**, **causal**, and **event** phenomena. We model the different kinds of phenomena as an enumeration type PhenomenonType.

In frame diagrams, **interfaces** connect domains, and they contain phenomena. Therefore, we introduce the class Interface, which has associations to the classes InterfacePhenomenon and Domain. Interfaces connect (connects) at least two domains ([2..*]), and they contain (contains) at least one ([1..*]) phenomenon. Each domain is connected by at least one ([1..*]) interface (isConnectedBy), and each phenomenon is contained in exactly one ([1]) interface (isContainedInIf). An additional association between the class Domain and the class InterfacePhenomenon specifies that each interface phenomenon is controlled by (isControlledBy) exactly one ([1]) domain.

A problem frame contains a **requirement** that refers to certain domains via **requirement references**. A requirement **constrains** at least one domain. Hence, we introduce the classes Requirement, RequirementReference, and its subclass ConstrainingRequirementReference. Furthermore, we introduce associations between the class Requirement and the classes RequirementReference and ConstrainingRequirementReference. The latter two classes have also associations to the class Domain. A domain is either referred to (isReferredToByUsing) or constrained by (isConstrainedByUsing) a requirement reference or not ([0..1]). Each requirement reference is connected to (isConnectedToDomain) exactly one ([1]) domain. A requirement reference or a constraining requirement reference is connected to (isConnectedToReq) exactly one ([1]) require-

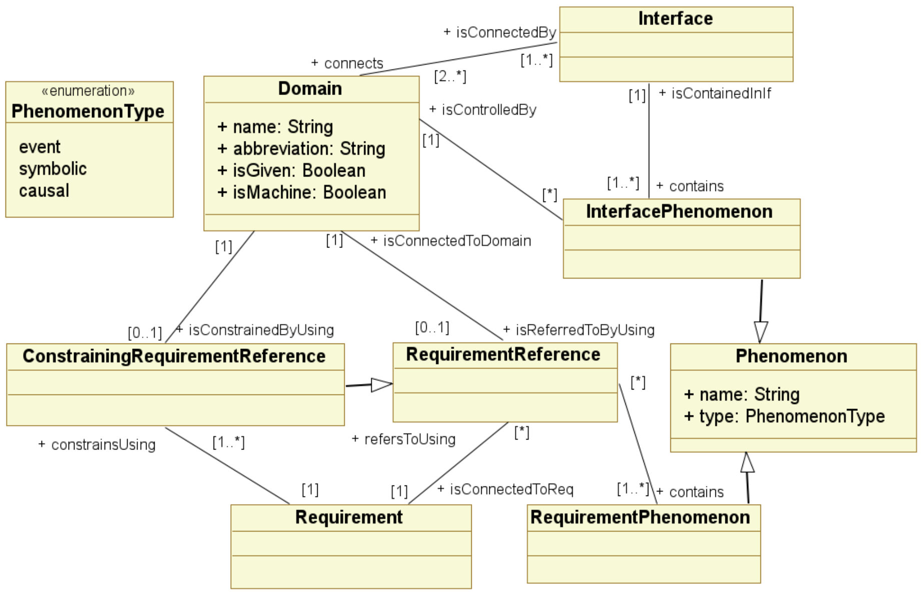

Fig. 3. Problem frames constituents and their relations

ment. A requirement refers to a domain using (refersToUsing) a requirement reference, and it constrains a domain using (constrainsUsing) at least one ([1..*]) constraining requirement reference.

Since each (constraining) requirement reference contains phenomena, we introduce a subclass RequirementPhenomenon of the class Phenomenon, which has an association to the class RequirementReference. A (constraining) requirement reference contains (contains) at least one ([1..*]) requirement phenomenon.

According to Jackson, a **problem frame** consists of domains, a requirement, interfaces, and requirement references. Hence, we introduce a class ProblemFrame in Fig. 4. A problem frame consists of one machine domain and at least one additional domain (association end domains with multiplicity [2..*]), exactly one requirement (association end requirement with multiplicity [1]), at least one interface (association end interfaces with multiplicity [1..*]), and at least one constraining requirement reference (association end requirementReferences with multiplicity [1..*]).

3.2 Integrity Conditions

The class model presented in the previous Sect. 3.1 makes it possible to define *integrity conditions* on problem frames and their syntactical elements, which we express in OCL. The class model with its associated integrity conditions constitutes a formal metamodel for the development of new problem frames, as well as for validating and analyzing given problem frames with respect to the metamodel.

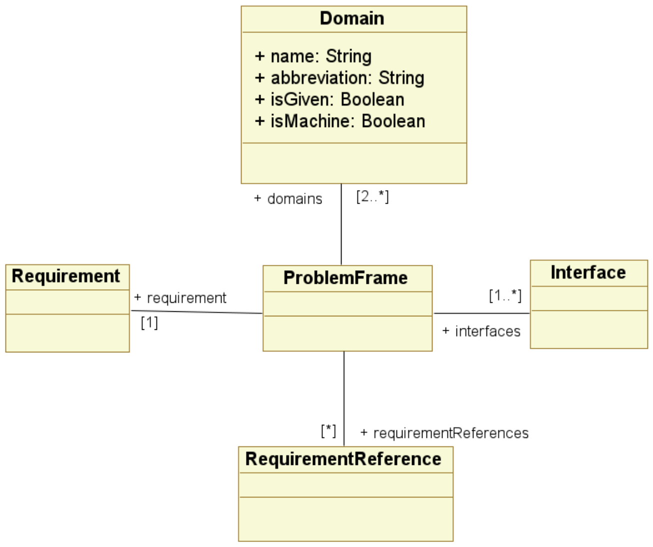

Fig. 4. A problem frame and its constituents

OCL is part of UML; it is a notation to describe constraints on object-oriented modeling artefacts such as class models. A constraint is a restriction on one or more values of an object-oriented model. For our metamodel, we make use of *invariants*, which are constraints that must always be fulfilled. Invariants are labeled with the keyword `inv`.

The `context` definition of an OCL expression specifies the model entity for which the OCL expression is defined. In our metamodel, this is usually a class defined in the class model depicted in Figs. 2, 3, and 4. In the following, a selection of the OCL integrity conditions are described, according to their context. The complete set of integrity conditions can be found in Appendix A. All integrity conditions have been checked using the *Octopus OCL Tool for Precise UML Specifications* [4]. The relatively high number of integrity conditions shows that the semantic integrity of problem frames is not trivial.

`context Domain`

A machine domain is always a designed domain. Therefore, a machine domain cannot be a given domain.

```
inv: self.isMachine implies not self.isGiven
```

`context BiddableDomain`

Biddable domains are always given domains.

```
inv: self.isGiven
```

A biddable domain cannot be constrained.

```
inv: self.isConstrainedByUsing->size() = 0
```

`context ProblemFrame`

A phenomenon contained in an interface is controlled by a domain connected by that interface.

```
inv: self.isContainedInIf.connects->includes(
  self.isControlledBy)
```

Each phenomenon contained in an interface is controlled by exactly one domain connected by that interface.

```
inv: self.contains->forAll(p: InterfacePhenomenon |
  self.connects->one(d: Domain |
   p.isControlledBy = d))
```

The machine domain is connected by an interface to a problem domain.

```
inv: self.interfaces->exists(i: Interface |
  self.domains->exists(d: Domain |
  self.domains->exists(m: Domain | m.isMachine and
  m.isConnectedBy->includes(i) and
  d.isConnectedBy->includes(i))))
```

The machine domain controls phenomena contained in an interface it is connected to.

```
inv: self.interfaces->exists(i: Interface |
  i.connects->exists(d: Domain | d.isMachine) and
  i.contains->exists(p: InterfacePhenomenon |
  p.isControlledBy.isMachine))
```

A domain is connected by interfaces, which are part of the problem frame.

```
inv: self.domains->forAll(d: Domain |
  self.interfaces->includesAll(d.isConnectedBy))
```

The names of domains must be unique.

```
inv: self.domains->forAll(d1: Domain |
  self.domains->forAll(d2: Domain |
  (d1.name = d2.name) implies (d1=d2)))
```

The requirement's constraining requirement references and requirement references are those contained in the problem frame.

```
inv: self.requirementReferences->includesAll(
  self.requirement.constrainsUsing)
```

4 Tool Support

Individual problem frames can be described as instantiations of the formal metamodel presented in Sect. 3. This formal metamodel constitutes the basis for the development of tool support for the problem frame approach.

We currently develop an Eclipse [1] application that supports generating newly defined problem frames for requirements analysis. This tool will be a rich graphical problem frame editor that allows one to draw and store problem frames, and will check the integrity conditions specified in the metamodel. We base this application on the open-source frameworks *Eclipse Modeling Framework* (EMF) [2], and *Graphical Editing Framework* (GEF) [3]. As an input for these frameworks, a metamodel specification in, e.g., *XML Metadata Interchange (XMI)* [6] (XML [7] storage format for UML diagrams) can be used. UML tools such as Papyrus UML [5] can generate these XMI files from the models presented in Figs. 2, 3, and 4. The XMI files serve as an input for EMF and GEF to automatically generate tool support for the problem frames approach. Additionally, for all integrity expressions from Appendix A, checks must be implemented or generated.

A a proof of concept, we have used the *Octopus OCL Tool for Precise UML Specifications* [4]. Octopus can check the syntax of the OCL constraints according to a corresponding UML class model. It can also import a class model from a tool supporting the XMI storage format. We used Octopus to check the OCL expressions presented in Appendix A, with the class diagram presented in Sect. 3.1, and to automatically generate a tool to create and edit instances of our formal metamodel for problem frames[1]. The generated tool can check whether a given problem frame is valid according to our formal metamodel[2]. It also generates a user interface prototype and an XML storage facility.

The approach of our formal metamodel and the automatic tool generation with Octopus (and also EMF and GEF) is comparable to the *model-driven architecture* (MDA) approach proposed by the *Object Management Group* (OMG). This approach is based on the metadata architecture including the XMI specification, called the *meta object facility*. A typical metamodels proposed by OMG is the UML metamodel, as it is described in the UML *superstructure* [31].

5 Formalizing and Checking Individual Problem Frames

We used the generated tool to check whether a given problem frame is valid according to our formal metamodel. With its user interface, all elements of a problem frame can be created and connected according to its graphical representation as an instance of the metamodel. The tool stores the instance in an XML file. In this XML file, the tool assigns an identification name to each instance of a class[3]. For each attribute of a class, the tool stores a value, and for each each navigable association end the tool stores the reference names of the connected class instances.

We validated our formal metamodel and the generated tool by instantiating Jackson's basic problem frames. The complete instance of the simple workpieces problem frame (see Fig. 1) can be found in [18]. The tool indeed shows that the instance is valid according to the formal metamodel, i.e., it is type correct and all OCL integrity conditions hold.

To demonstrate that the generated tool supports detecting errors, we consider the erroneous "problem frame" shown in Fig. 5. It is an incorrect variant of the simple workpieces problem frame shown in Fig. 1.

Figure 6 shows the domain *User* defined as a designed domain. According to the formalization of the class BiddableDomain in our metamodel, a biddable domain is always a given domain. Therefore, the generated tool finds an error when checking this OCL integrity condition.

[1] Download: http://swe.uni-duisburg-essen.de/en/members/schmidt/pftool/index.php

[2] For OCL expressions with `oclIsTypeOf` no code can be generated by the available version of Octopus.

[3] For better readability, we replaced the automatically assigned numbers by meaningful names in Fig. 6.

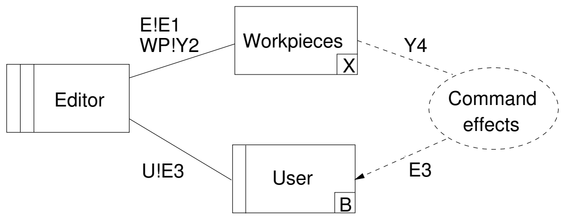

Fig. 5. Erroneous variant of the simple workpieces problem frame shown in Fig. 1

```
<instance id="id_U" class="PF.BiddableDomain">
        <attribute name="name" value="User" />
        <attribute name="abbreviation" value="U" />
        <attribute name="isGiven" value="false" />
        <attribute name="isMachine" value="false" />
        <attribute name="controls" idrefs=" id_e3" />
        <attribute name="isConnectedBy" idrefs=" id_i_e3" />
        <attribute name="isConstrainedByUsing" idref="id_crr" />
</instance>
```

Fig. 6. XML specification of the domain *User* of the simple workpieces problem frame shown in Fig. 1

The attribute isConstrainedByUsing has the value id_crr. According to the formalization of the class BiddableDomain in our metamodel, a biddable domain is never constrained. Therefore, the generated tool finds a further error when checking this OCL integrity condition.

The other parts of the erroneous “problem frame” shown in Fig. 5 are valid with respect to our formal metamodel.

6 Related Work

Not much work on providing the problem frame approach with a formal foundation can be found.

Lencastre et al. [23] define a metamodel for problem frames using UML. Their metamodel considers Jackson’s whole software development approach based on context diagrams, problem frames, and problem decomposition. In contrast to our metamodel, it only consists of a UML class model. Hence, the OCL integrity conditions of our metamodel are not considered in their metamodel. The approach does not qualify for a metamodel in terms of MDA since, e.g., the class Domain has subclasses Biddable and Given, but an object cannot belong to two classes at the same time (c.f. Fig. 5 and 11 in [23]).

Hall et al. [14] 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 metamodel, their approach does not consider integrity conditions.

Seater et al. [26] present a metamodel for problem frame instances. In addition to the diagram elements formalized in our metamodel, they formalize requirements and specifications. Consequently, their integrity conditions ("wellformedness predicate") focus on correctly deriving specifications from requirements. In contrast, our metamodel concentrates on the structure of problem frames and the different domain and phenomena types.

7 Conclusions and Perspectives

In this paper, we have presented a formal metamodel for problem frames, which contains a number of integrity conditions. The metamodel with its integrity conditions constitutes a framework for the development of new problem frames as well as for the semantic validation and syntax checking of individual problem frames with respect to the metamodel. Additionally, it contributes to an unambiguous comprehension of the problem frame approach.

We have shown that such a metamodel with its integrity conditions can be used to automatically generate a tool according to the model-driven architecture approach. We used the metamodel and the generated tool to instantiate Jackson's basic problem frames and to check our own variations of the frames (e.g., [9]).

In the future, we intend to further elaborate our approach in the following directions:

– Apply the metamodel to context diagrams and instantiations of problems frames.
– Consider instantiations of formally defined problem frames, and use these formal descriptions to support the decomposition of problems into subproblems and the composition of the solutions developed for the subproblems.
– Build a more mature graphical problem frame editor based on GEF and EMF.
– Integrate the problem frame approach with object-oriented problem analysis methods and software development processes.

In summary, we believe that with this paper, we have prepared the basis for a more rigorous understanding of problem frames, which can also contribute to further maturing and enhancing the problem frame approach in practice.

References

[1] Eclipse - An Open Development Platform, May 2008. http://www.eclipse.org/.
[2] Eclipse Modeling Framework Project (EMF), May 2008. http://www.eclipse.org/modeling/emf/.
[3] Graphical Editing Framework Project (GEF), May 2008. http://www.eclipse.org/gef/.
[4] OCL Tool for Precise UML Specifications (Octopus), May 2008. http://www.klasse.nl/octopus/.
[5] Papyrus UML, May 2008. http://www.papyrusuml.org.
[6] XMI - XML Metadata Interchange, May 2008. http://www.omg.org/docs/formal/05-09-01.pdf.

[7] XML - Extensible Markup Language, May 2008. http://www.w3.org/XML/.

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[9] C. Choppy and M. Heisel. Une approache à base de “patrons” pour la spécification et le développement de systèmes d’information. In *Proceedings Approches Formelles dans l’Assistance au Développement de Logiciels - AFADL’2004*, pages 61–76, 2004.

[10] I. Côté, D. Hatebur, M. Heisel, H. Schmidt, and I. Wentzlaff. A systematic account of problem frames. In *Proceedings of the European Conference on Pattern Languages of Programs (EuroPLoP)*. Universitätsverlag Konstanz, to be published in 2008.

[11] M. E. Fayad and R. E. Johnson. *Domain-Specific Application Frameworks*. John Wiley & Sons, 1999.

[12] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison Wesley, 1997.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, 1995.

[14] J. G. Hall, L. Rapanotti, and M. Jackson. Problem frame semantics for software development. *Software and System Modeling*, 4(2):189–198, 2005.

[15] D. Hatebur, M. Heisel, and H. Schmidt. Security Engineering using Problem Frames. In Müller, G., editor, *Proceedings of the International Conference on Emerging Trends in Information and Communication Security (ETRICS)*, LNCS 3995, pages 238–253. Springer-Verlag, 2006.

[16] D. Hatebur, M. Heisel, and H. Schmidt. A pattern system for security requirements engineering. In *Proceedings of the International Conference on Availability, Reliability and Security (AReS)*, pages 356–365. IEEE, 2007.

[17] D. Hatebur, M. Heisel, and H. Schmidt. A security engineering process based on patterns. In *Proceedings of the International Workshop on Secure Systems Methodologies using Patterns (SPatterns)*, pages 734–738. IEEE, 2007.

[18] D. Hatebur, M. Heisel, and H. Schmidt. A formal metamodel for problem frames (technical report). Technical report, 2008. http://swe.uni-duisburg-essen.de/en/members/schmidt/index.php.

[19] M. Jackson. *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, 1995.

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

[21] M. Jackson and D. Jackson. Problem decomposition for reuse. *Software Engineering Journal*, 11(1):19–30, 1996.

[22] M. Jackson and P. Zave. Deriving specifications from requirements: an example. In *Proceedings 17th Int. Conf. on Software Engineering, Seattle, USA*, pages 15–24. ACM Press, 1995.

[23] M. Lencastre, J. Botelho, P. Clericuzzi, and J. Araújo. A meta-model for the problem frames approach. In *WiSME’05: 4th Workshop in Software Modeling Engineering*, 2005.

[24] L. Rapanotti, J. G. Hall, M. Jackson, and B. Nuseibeh. Architecture-driven problem decomposition. In *Proceedings of the 2004 International Conference on Requirements Engineering (RE’04), Kyoto*. IEEE CS Press, 2004. http://mcs.open.ac.uk/mj665/ArchDrvn.pdf.

[25] H. Schmidt and I. Wentzlaff. Preserving software quality characteristics from requirements analysis to architectural design. In *Proceedings of the European Workshop on Software Architectures (EWSA)*, volume 4344/2006, pages 189–203. Springer-Verlag, 2006.

[26] R. Seater, D. Jackson, and R. Gheyi. Requirement progression in problem frames: deriving specifications from requirements. *Requirements Engineering*, 12(2):77–102, 2007.

[27] M. Specker and I. Wentzlaff. Exploring usability needs by human-computer interaction patterns. In *Proceedings of the 6th International Workshop on TAsk MOdels and DIAgrams*. Springer-Verlag, 2007.

[28] A. Sutcliffe. *The Domain Theory, Patterns for Knowledge and Software Reuse*. Addison Wesley, 2002.
[29] A. Sutcliffe and N. Maiden. The domain theory for requirements engineering. *IEEE Transactions on Software Engineering*, 24(3):174–196, 1998.
[30] UML Revision Task Force. *Object Constraint Language Specification*, May 2006. Object Constraint Language (OCL).
[31] UML Revision Task Force. *OMG Unified Modeling Language: Superstructure*, February 2007. http://www.omg.org/docs/formal/07-02-03.pdf.

A The Complete Set of Integrity Conditions

`context Domain`

A machine domain is always a designed domain. Therefore, a machine domain cannot be a given domain.

```
inv: self.isMachine implies not self.isGiven
```

A machine domain cannot be referred to by a requirement using a requirement reference.

```
inv: self.isMachine implies
  self.isReferredToByUsing->size() = 0
```

A machine domain cannot be constrained.

```
inv: self.isMachine implies
  self.isConstrainedByUsing->size() = 0
```

`context BiddableDomain`

Biddable domains are always given domains.

```
inv: self.isGiven
```

Machine domains are always causal domains. Hence, biddable domains cannot be machine domains.

```
inv: not self.isMachine
```

A biddable domain cannot be constrained.

```
inv: self.isConstrainedByUsing->size() = 0
```

`context LexicalDomain`

Lexical domains cannot be machine domains since they represent data.

```
inv: not self.isMachine
```

`context Interface`

Each phenomenon contained in an interface is controlled by exactly one domain connected by that interface.

```
inv: self.contains->forAll(p: InterfacePhenomenon |
  self.connects->one(d: Domain |
   p.isControlledBy = d))
```

`context InterfacePhenomenon`

A phenomenon contained in an interface is controlled by a domain connected by that interface.

```
inv: self.isContainedInIf.connects->includes(
  self.isControlledBy)
```

`context RequirementReference`

Machine domains cannot be referred to by requirements (other direction of association).

```
    inv: not(self.isConnectedToDomain.isMachine)
```

context ConstrainingRequirementReference

A machine domain cannot be constrained.

```
    inv: not (self.isConnectedToDomain.isMachine)
```

A biddable domain cannot be constrained (other direction of association).

```
    inv: not(self.isConnectedToDomain
      ->oclIsTypeOf(BiddableDomain))
```

context ProblemFrame

A problem frame has exactly one machine domain.

```
    inv: self.domains->one (d: Domain | d.isMachine)
```

A phenomenon contained in an interface is controlled by a domain connected by that interface.

```
    inv: self.isContainedInIf.connects->includes(
      self.isControlledBy)
```

The machine domain is connected by an interface to a problem domain.

```
    inv: self.interfaces->exists(i: Interface |
      self.domains->exists(d: Domain |
      self.domains->exists(m: Domain | m.isMachine and
      m.isConnectedBy->includes(i) and
      d.isConnectedBy->includes(i))))
```

The machine domain controls phenomena contained in an interface it is connected to.

```
    inv: self.interfaces->exists(i: Interface |
      i.connects->exists(d: Domain |
      d.isMachine) and
      i.contains->exists(p: InterfacePhenomenon |
      p.isControlledBy.isMachine))
```

Domains are connected by an interface to another problem domain.

```
    inv: self.domains->forAll(d: Domain |
      self.interfaces->exists(i: Interface |
      d.isConnectedBy->includes(i)))
```

A domain can only control phenomena contained in those interfaces that are connected to it.

```
    inv: self.domains->forAll(d: Domain |
      d.controls->forAll(p: InterfacePhenomenon |
      d.isConnectedBy->exists(i: Interface |
      i.contains->includes(p))))
```

A domain is connected by interfaces, which are part of the problem frame.

```
    inv: self.domains->forAll(d: Domain |
      self.interfaces->includesAll(d.isConnectedBy))
```

A domain can be constrained by constraining requirement references, which are part of the problem frame.

```
    inv: self.domains->forAll(d: Domain |
      not d.isMachine implies self.requirementReferences
      ->includes(d.isConstrainedByUsing)
      or d.isConstrainedByUsing->size()=0)
```

A domain can be referred to by requirement references, which are part of the problem frame.

```
inv: self.domains->forAll(d: Domain |
  not d.isMachine implies self.requirementReferences
  ->includes(d.isReferredToByUsing)
  or d.isReferredToByUsing->size()=0)
```

The names of domains must be unique.

```
inv: self.domains->forAll(d1: Domain |
  self.domains->forAll(d2: Domain |
  (d1.name = d2.name) implies (d1=d2)))
```

The abbreviations of domains must be unique.

```
inv: self.domains->forAll(d1: Domain |
  self.domains->forAll(d2: Domain |
  (d1.abbreviation = d2.abbreviation)
  implies (d1=d2)))
```

The names of problem domains and the phenomena names must be disjoint.

```
inv: self.domains->forAll(d: Domain |
  self.interfaces->forAll(
  i: Interface | i.contains->forAll(
  p: InterfacePhenomenon | d.name <> p.name)))
```

Each interface is connected to a domain contained in the problem frame.

```
inv: self.interfaces->forAll(i: Interface |
  self.domains->includesAll(i.connects))
```

Phenomena contained in an interface are controlled by exactly one domain that is contained in the problem frame and connected by the interface at hand.

```
inv: self.interfaces->forAll(i: Interface |
  i.contains->forAll(p: InterfacePhenomenon |
  self.domains->one(d: Domain |
  d.controls->includes(p) and
  i.connects->includes(d))))
```

The requirement's constraining requirement references and requirement references are those contained in the problem frame.

```
inv: self.requirementReferences->includesAll(
  self.requirement.constrainsUsing)
```

The requirement references are connected to the requirement of the problem frame (both directions of association).

```
inv: self.requirement.refersToUsing =
  self.requirementReferences

inv: self.requirementReferences->forAll(
  rr: RequirementReference |
  rr.isConnectedToReq = self.requirement)
```

The constraining requirement references and requirement references are connected to the domains of the problem frame.

```
inv: self.requirementReferences->forAll(
  rr: RequirementReference |
  self.domains->includes(rr.isConnectedToDomain))
```