A UML profile and Tool Support for Evolutionary Requirements Engineering

Isabelle Côté
University Duisburg-Essen
Oststr. 99, Duisburg, Germany
Email: isabelle.cote@uni-due.de

Maritta Heisel
University Duisburg-Essen
Oststr. 99, Duisburg, Germany
Email: maritta.heisel@uni-due.de

Abstract—**In this paper, we present a method to perform the first steps of software evolution, namely evolutionary requirements engineering, where new requirements have to be analyzed in the context of a set of already given requirements. The basic idea is to adjust an existing requirements engineering process so that evolution is supported. In the requirements engineering process we consider, the original software development problem is decomposed into a number of subproblems that are analyzed according to the problem frame approach [1]. Evolution is performed by defining rules for each process step and each document that is generated in the respective step to incorporate the new evolution requirements into the existing requirements documents or to create, when necessary, additional documents. We show that the evolution task benefits from the chosen problem decomposition. The described software evolution method is tool-supported. Our tool UML4PF, which is based on the Eclipse Modeling Framework, supports the problem frame approach to software engineering by a specifically defined UML profile. We extend that profile so that it also covers software evolution.**

I. Introduction

In the late 1960's Lehman investigated the evolution of a large software system. Based on the observations gained he derived the laws of software evolution [2] such as "Continuing Change" and "Declining Quality". Both of the mentioned laws point out, that if a software system is not continually adapted and maintained, it will degrade. Parnas [3] states that software ages for two reasons: first, because it is changed. These changes affect the structure of the software, and, at a certain point, further changes become infeasible. The second reason for software aging is *not* to change the software. Such software becomes outdated soon, because it does not reflect new developments and technologies. We can conclude that software evolution is indispensable for obtaining long-lived software. Taking into account that today only a small fraction of software is written from scratch, it is important to provide support for coping with situations where an existing software needs to be changed in order to remain operational. For this reason, it is necessary to provide systematic support for the evolution task. Ideally, this support can be embedded into an existing development process. In conclusion, to avoid the legacy problems of tomorrow [4], we first need appropriate development processes that provide a good basis for future evolutions. Second, we need systematic evolution approaches that can make use of that basis. In this paper, we base software evolution on a development process where a complex software development problem is decomposed into a number of subproblems that are analyzed according to the problem frame approach [1]. We have defined a number of *evolution operators* that support engineers in performing the evolution task. These operators guide the engineering process in evolving given artifacts. They also assist in the reuse of related development artifacts in successive development phases. The development process we present is tool-supported. To this end, we have defined a UML profile [5] that allows us to represent problem frames in UML as well as to model the requirements analysis phase by introducing appropriate stereotypes. We show how this UML profile can be augmented with stereotypes that support software evolution. The stereotypes are complemented by constraints and queries expressed in OCL [6] that can be checked by existing UML tools. These constraints express important integrity conditions, whereas the queries help us in finding relevant documents by following the linkages between the different artifacts. This enables us to decrease the effort for performing the evolution by determining only a subset of relevant artifacts that have to be considered.

The paper is organized as follows: Section II introduces a description of the vacation rentals system serving as a case study in this paper. In Sect. III we briefly describe the software development process we start out with, illustrated by the vacation rentals system. Section IV presents the currently existing tool support for the software development process. Section V illustrates the requirements engineering steps of our evolution method and its application to the case study. Section VII discusses related work, and Sect. VIII concludes the paper with a summary and directions for future work.

II. Case Study: Vacation Rentals

In this section, we introduce the running example for our paper, namely the development of a simple online vacation rentals system. The vacation rentals system shall allow a potential guest to browse and book available holiday offers. After booking an offer the guest receives an invoice and the holiday offer is reserved. The guest has now 14 days to pay the invoice via bank transfer. Should this not be the case then the offer is automatically set available again. Staff members are responsible for recording the incoming payments and to rate the status of a vacation home after the guests have left. A negative rating means that the guest receives an additional invoice. A further responsibility of the staff member is to make new holiday offers available.

(R03)	A guest can book available holiday offers, which then are reserved.
(R04)	After a guest books a holiday offer, she is provided a corresponding invoice.
(R05)	If a reserved holiday offer is not paid within 14 days, it is automatically set available again.
(R07)	A staff member can rate the status of a vacation home, after a guest left it.
(R08)	If the status of a vacation home is rated negatively, the guest receives an additional invoice.

TABLE I
SUBSET OF INITIAL SET OF REQUIREMENTS FOR VACATION RENTALS

In the subsequent section, we illustrate the development of the vacation rentals system using a development process called ADIT (Analysis, Design, Implementation, Testing; see [7] for more details).

III. DEVELOPMENT PROCESS

ADIT is a model-driven, pattern-based development process also making use of components. The different phases of software development are divided into several steps. In this paper, we focus on the requirements analysis steps of the analysis phase which forms the basis for our evolutionary requirements engineering method, called EADIT. We describe these steps in the following subsections.

A. Problem elicitation and description

The analysis phase starts by stating the requirements that define properties of the desired system. Table I shows a subset of the initial requirements for the vacation rentals system. Furthermore, we need to state the characteristics of the part of the "real world" (environment) that is relevant for our problem, i.e., the relevant 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. "A guest pays the fee completely within 14 days or not at all." is an example for an assumption and "A vacation home can be only used by one guest (+ accompanying persons) at the same time" is a fact for the vacation rentals system.

After having collected the requirements and domain knowledge of the problem, we analyze its structure by setting up a *context diagram* [1]. The context diagram represents the overall problem situation. An example of a context diagram is shown in Fig. 1. A context diagram consists of boxes, called domains. There are different types of domains. 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. We introduced an additional domain type, namely *display domains* that are used to provide feedback to users [8]. We represent context diagrams as UML class diagrams, using specific stereotypes, see Sect. IV. The domains are connected with each other via *shared phenomena*. A shared phenomenon is an event, a message or an operation observable by at least two domains, but controlled by only one domain. An example for such shared phenomena is given in Fig. 2. The notation *G*!{*browseAvailableHolidayOffers, bookHolidayOffer*}

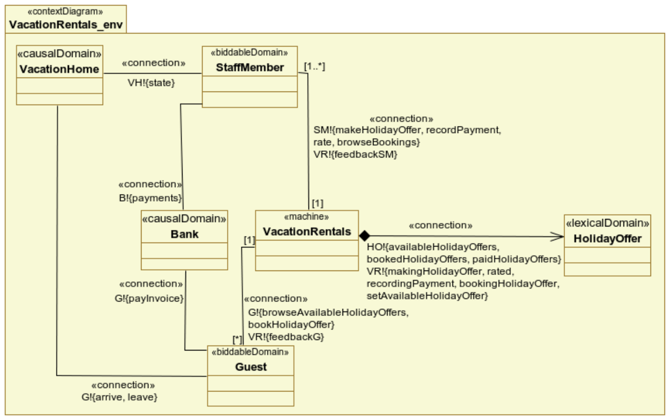

Fig. 1. Vacation Rentals: Original Context Diagram

means that the phenomena *browseAvailableHolidayOffers* and *bookHolidayOffer* are controlled (indicated by "!") by the domain *Guest* (G).

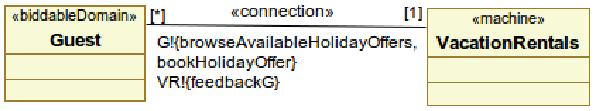

Fig. 2. Vacation Rentals: Interface between domains Guest and VacationRentals

B. Problem decomposition

In the second analysis step, we decompose the overall problem into subproblems. One way of performing this decomposition is by setting up *problem diagrams* [1]. In a problem diagram, the knowledge for a sub-problem described by a set of requirements is represented. A requirement in a problem diagram refers to some domains and constrains at least one domain. A problem diagram can be systematically derived from the context diagram by means of *decomposition operators*. Examples for such operators are:

- *introduce connection domain*: such a domain serves to mediate the communication between other domains. An example is a display domain that outputs feedback to e.g. a biddable domain.
- *reduce interface between domains*: phenomena between domains which are not relevant for the current subproblem are left out.

Figure 3 shows the problem diagram for the (composed) requirement (R03,R04).

We see that a connection domain *Email* with corresponding phenomena has been introduced to handle the distribution of invoices (application of decomposition operator *introduce connection domain*).

Whenever possible, the decomposition is done in such a way that the subproblems fit to given problem frames. Problem frames are a means to describe software development problems. They were proposed by Jackson [1], 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."*

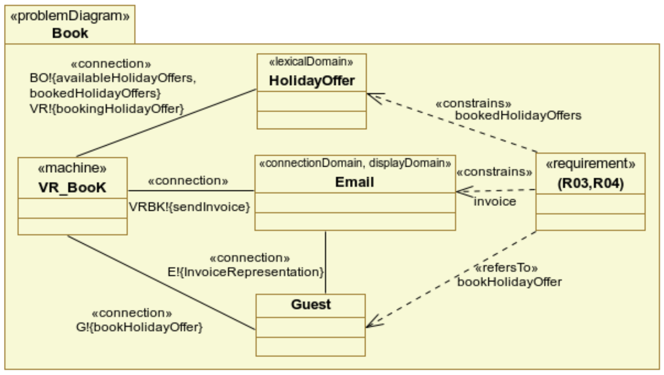
«problemDiagram» Book
«connection» BO!{availableHolidayOffers, bookedHolidayOffers} VR!{bookingHolidayOffer}
«lexicalDomain» HolidayOffer
«constrains» bookedHolidayOffers
«machine» VR_BooK
«connection» VRBK!{sendInvoice}
«connectionDomain, displayDomain» Email
«constrains» invoice
«requirement» (R03,R04)
«connection» E!{InvoiceRepresentation}
«refersTo» bookHolidayOffer
«connection» G!{bookHolidayOffer}
Guest

Fig. 3. Vacation Rentals: Problem Diagram for *book* (R03, R04)

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 instantiated frame diagram is a problem diagram. For example, the requirements R03 and R04 constitute an instance of the update frame [9] shown in Fig. 4. All requirements must be covered by some problem diagram. Similar to the context diagram, the problem diagrams as well as the problem frames are represented as class diagrams. In addition to context diagrams, problem diagrams and problem frames take a requirements reference into account (see Sect. IV).

The subsequent steps (not covered in this paper) would then be to

1. *derive abstract specifications*. Requirements refer to the environment in which the machine must operate. In contrast, the specification describes the machine and is the starting point for its development. It has to be derived using domain knowledge.
2. create a *technical context diagram*. This technical context diagram describes the technical infrastructure in which the machine will be embedded.
3. *specify operations and data structures*. The operations and internal data structures identified while deriving the abstract specifications are specified in detail.
4. define a *software life-cycle*. The overall behavior of the machine is specified.

IV. ADIT Tool Support

We have developed a tool called *UML4PF* [10] to support the requirements engineering method sketched in Section III. Its basis is the Eclipse platform [11] together with its plug-ins Eclipse Modeling Framework (EMF) [12] and Object Constraint Language (OCL) [6]. EMF allows engineers to create structured data models compliant to UML. The data models are equipped with meta-data, which can be queried and updated via a Java interface. EMF stores model information using the XML Meta-data Interchange (XMI) [13] format. With OCL it is possible to formally specify constraints over a given model. The graphical representation of the different diagram types can be manipulated by using any EMF-based editor. We selected Papyrus [14] as it is available as an Eclipse

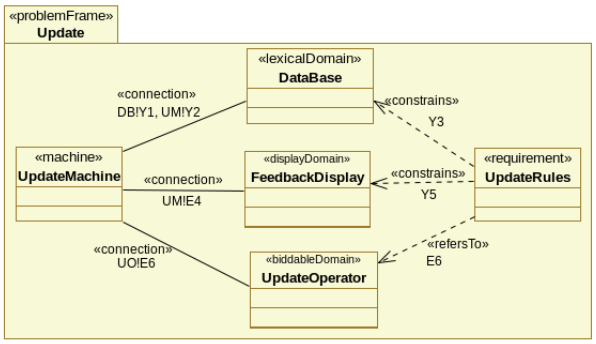

Fig. 4. *Update* problem frame

plug-in, open-source, and EMF-based. UML4PF provides the necessary UML profile to use UML diagrams for the problem frame approach. Our UML profile is conceived as an Eclipse plug-in, extending the EMF meta-model.

With the profile, it is possible to model the domains in the context diagram as well as in the problem diagrams by classes and assign to them corresponding stereotypes, e.g., *<<biddableDomain>>* or *<<lexicalDomain>>* for the respective domain types (see Figs. 1 and 3). The class with the stereotype *<<machine>>* represents the software to be developed. The shared phenomena are represented as associations between classes. Such an association is marked with the stereotype *<<connection>>* (or any of its subtypes, e.g., *<<ui>>* for user interface), and the name of the association contains the phenomena and the domain controlling the phenomena (see Fig. 2). Our tool automatically transforms a *<<connection>>* (or any of its subtypes) into an interface controlled by a domain and observed by other domains. To express this, the stereotypes *<<observes>>* and *<<controls>>* are used (see Fig. 5). The interface contains all phenomena as operations. We use the name of the association (or relevant parts of it) as name for the interface. Requirements are represented

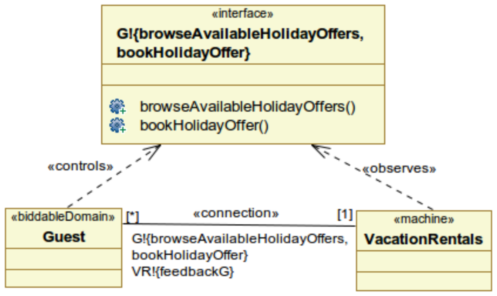
«interface» G!{browseAvailableHolidayOffers, bookHolidayOffer}
browseAvailableHolidayOffers()
bookHolidayOffer()
«controls»
«observes»
«biddableDomain» Guest
[*]
«connection»
[1]
G!{browseAvailableHolidayOffers, bookHolidayOffer} VR!{feedbackG}
«machine» VacationRentals

Fig. 5. Vacation Rentals: Generated interface

as classes, as well. They are denoted by the stereotype *<<requirement>>*. As a requirement in a problem diagram refers to some domains and constrains at least one domain. We express this using the stereotypes *<<refersTo>>* and *<<constrains>>* (see dependencies on right-hand side of Fig. 3).

To check the validity and consistency of the models set up when performing requirements analysis according to ADIT, we use OCL constraints. An example for such an OCL-constraint is given in Listing 1. We mentioned earlier that

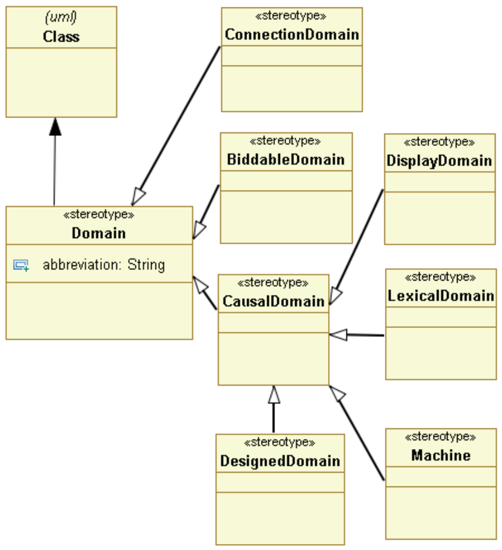

Fig. 6. Inheritance hierarchy of domain types

we have several domains types. Some of the domain types are not disjoint, so more than one stereotype can be applied on one class. However, not all combinations of stereotypes are permitted. For example, the stereotypes <<*CausalDomain*>> (or subtypes) and <<*BiddableDomain*>> are not allowed to be applied together on one class. Hence, we provide an OCL expression that checks whether this condition is fulfilled. Listing 1 depicts the corresponding expression. It is read as follows: in line 1, all the classes of the model are selected that satisfy the condition stated in 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 subtype of <<*BiddableDomain*>> **and** the stereotype <<*CausalDomain*>> or a subtype 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*). An overview of the hierarchy is shown in Fig. 6. As we currently have three hierarchy levels in our meta-model, we limit our constraints to this number (lines 3-8). If new domain types are 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 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. Domain cannot be Causal and Biddable

All in all we currently have 40 OCL constraints that check the analysis diagrams of ADIT.

V. Evolutionary Requirements Engineering

The lifespan of software often covers several years, in some cases even decades. During this time it is unavoidable that the environment as well as the requirements change or additional requirements occur. These changes or additions in the requirements or the environment form the basis on which the evolution takes place. In this section, we describe a method to systematically perform the first steps of such an evolution, i.e., evolutionary requirements engineering. The method is based on the development process described in Sect. III. This means that we rely on documents that are generated during the development. In our case, we rely on documents of the analysis phase e.g. requirements, context diagram etc. Furthermore, we need an evolution task. Usually, this task is described by a shortcoming of the current software system. The evolution task is then to overcome this shortcoming. A shortcoming for the vacation rentals system may be stated as follows: "So far, a guest can only browse and book a holiday offer. (S)he has no possibility to cancel a previously booked offer other than to wait 14 days for the offer to expire." In order to eliminate this shortcoming, we can state that "It shall be possible for guests to cancel their previously booked holiday offer(s). The cancellation is confirmed via an Email."

As a preliminary step, we derive requirements, similarly to ADIT. In order to distinguish the requirements of the original development from those requirements derived now, we refer to the latter as *evolution requirements*. For our example, we can state the following evolution requirements:

> A guest can cancel a booking as long as it has not been paid. (R10)

and

> If a guest cancels a booking, a confirming email will be sent. (R11)

In this case, no additional domain knowledge is required.

With the evolution requirements, possibly the additional domain knowledge as well as the results of the steps described in Sect. III, we meet all prerequisites to perform the evolution.

A. Relate requirements and revise problem decomposition

1) Relate requirements: The idea behind this first step is to narrow down the documents that need to be considered during the evolution. To identify those documents it is necessary to relate the evolution requirements to the (original) requirements. Since relations between requirements are important for the evolution task, we also identify relations between evolution and between original requirements. The following relations are of importance:

- **complements**: one requirement complements another requirement. In this case, both requirements are treated in the same problem diagram. This relation is similar to the <<*includes*>> relation used in use cases.
- **extends**: one requirement extends another requirement. Both requirements should be treated in the same problem diagram. This relation is basically the same as the <<*extends*>> relation used in use cases.

Taking the requirements shown in Tab. I, we can find the following relations between them:

- R04 complements R03
- R08 extends R07

Considering the evolution requirements we find that:

- R11 complements R10

Now we relate the evolution requirements to the original requirements. In addition to the above-mentioned relations, we also have the following ones at our disposition:

- **modifies**: the evolution requirement changes an original requirement, e.g. by restricting it.
- **replaces**: the evolution requirement replaces the original requirement.
- **similar**: the same domain types are referred to or constrained. It may occur that more than one similar-relation exists for one requirement. In that case, we should check for the best match. This is achieved by comparing the number of constrained and referenced domains that coincide. The best match is if all constrained and referenced domains coincide. The second best match is achieved if all constrained domains coincide, etc.
- **new**: none of the above-mentioned relations applies.

Applying the relations to our case study yields the following:

- R10 *modifies* R05: resetting a reserved holiday offer (booking) is only necessary if it has not been canceled or 14 days have passed without payment. Thus, requirement R05 must be adapted. The resulting requirement is:

  If a reserved holiday offer is not paid *nor has been canceled* within 14 days, it is automatically set available again.
- (R10, R11) is *similar* to (R03, R04) because the same domain types are being referred to or constrained.
- (R10, R11) is *similar* to (R07, R08) because the same domains are constrained. However, a different domain is referenced.

Therefore, the best match is with requirement (R03, R04).

2) Revise problem decomposition: In this second step, we must revise the existing problem decomposition. In order to find the problem diagrams which might undergo some change we can use the capability of OCL provided by EMF to carry out queries on the model information. Basically, we need to write a query in OCL that retrieves all problem diagram names that are related to a given evolution requirement. Such a query is discussed in detail in Listing 3 in Sect. VI.

Incorporating an evolution requirement into the problem diagram it is assigned to is a non-trivial task. To support this task, we have identified a number of operators that help to perform the required changes:

addPhenomena
: The requirements and/or domain knowledge introduce phenomena that can be added to an already existing interface. Listing 4 in Sect. VI shows the specification for this operator.

addDomain
: The requirements and/or domain knowledge introduce a new relevant domain. Relevant means that the domain is necessary to understand and solve the problem. This domain must be added to the corresponding problem diagram. Listing 5 in Sect. VI contains the specification for this operator. The new phenomena that occur have to be treated with operator *newPhenomena*.

newPhenomena
: In contrast to operator *addPhenomena*, it is not possible to add the phenomena to an already existing interface. Therefore, a new interface between the participating domains must be created with the provided phenomena.

modifyDomain
: The requirements and/or domain knowledge trigger a modification of existing domains. Possible modifications are for example splitting or merging of domains.
: In contrast to *addDomain*, modifying a domain does not necessarily result in introducing a new domain.

modifyPhenomena
: The shared phenomena have to be changed in order to handle the modified behavior derived from evolution requirement and/or additional domain knowledge, .e.g., by renaming. It may also be the result of operator *modifyDomain*.

The operators *replacePhenomena*, *replaceDomain*, *removePhenomena*, and *removeDomain* are defined analogously.

To analyze the evolution requirements, we must distinguish two cases. The decision, which case applies is lead by the relations we identified in Step V-A1. The above-mentioned operators help us in performing the necessary changes.

Case 1: *The evolution requirement can be integrated into an existing problem diagram*

- *An evolution requirement complements an original requirement*:
  - Add the evolution requirement to the existing problem diagram.
  - Check if all required domains are present.
    1) If they are: add any necessary phenomena to the interface between the mentioned domains (operator *addPhenomena*).
    2) If they are not: apply operator *addPhenomena* to those phenomena referring to the existing domains. Afterwards, apply operator *introduce connection domain* or add the new domain accordingly (operator *addDomain*). The phenomena that occur in this case have to be introduced with operator *newPhenomena*.
- *An evolution requirement modifies an original requirement*:
  - Check the type of modification:
    * A phenomenon is affected: modify (e.g., by renaming) the corresponding phenomena (operator *modifyPhenomena*).
    * A domain is affected: modify the corresponding domain (operator *modifyDomain*).

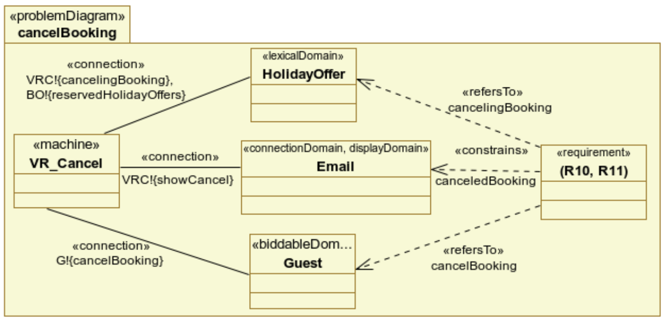

Fig. 7. Vacation Rentals: Problem Diagram for *cancel* (R10, R11)

Case 2: *The evolution requirement cannot be added to an existing problem diagram*

- *An evolution requirement is similar to an original requirement*: The domains of the original problem diagram are used as a skeleton for the new problem diagram. The necessary phenomena are introduced with the operator *newPhenomena*.
- *An evolution requirement is new.* Proceed according to ADIT to create a new problem diagram.

This procedure is carried out for every evolution requirement.

Note that in some cases, it may occur that neither domains nor shared phenomena are newly introduced or that the modification cannot be captured in the problem diagram (e.g. a parameter is modified). Then, no changes to the problem diagrams are necessary. However, the new requirements/domain knowledge may require changes in later steps. This is due to the fact that only the static aspects of the software and not the dynamic aspects are taken into account at this stage.

In the case study, we have to treat two relations. The first relation describes that evolution requirement R10 modifies R05. We see that the modification refers to a change in the dynamic behavior. Therefore, no adaptation of the corresponding problem diagram is necessary. The second relation describes that evolution requirement (R10, R11) is similar to two original requirements (R03, R04) and (R07, R08), respectively. As the problem diagram for (R03, R04) constitutes the best match, we use it as the skeleton for (R10, R11). As all required domains are present, we can focus on the phenomena:

- phenomenon *cancelingBooking*: apply operator *newPhenomena* to create the interface between the domains VR_Cancel and Booking
- phenomenon *cancelBooking*: apply operator *newPhenomena* to create the interface between the domains VR_Cancel and Guest
- phenomenon *sendEmail*: apply operator *newPhenomena* to create the interface between the domains VR_Cancel and Email

The resulting problem diagram is shown in Fig. 7.

B. Revise context diagram

We revise the context diagram according to the changes made in Step *Relate requirements and revise problem decomposition*. To this end, we can re-apply the operators of

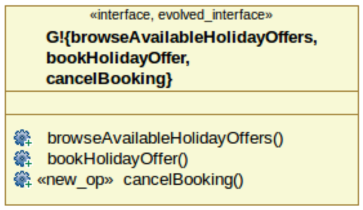
«interface, evolved_interface»
G!{browseAvailableHolidayOffers,
bookHolidayOffer,
cancelBooking}
browseAvailableHolidayOffers()
bookHolidayOffer()
«new_op» cancelBooking()

Fig. 8. Vacation Rentals: Evolved Interface between Guest and VacationRentals

Step V-A2, with one exception, however: connection domains as well as their corresponding phenomena are not added to the context diagram.

In Sect. V-A2, we saw that the modification of R05 has no impact on the static aspects. Thus, no changes to the context diagram are necessary. For the evolution requirement (R10, R11) we applied operator *newPhenomena* three times. One operator application was to add the phenomenon “sendInvoice” between the sub-machine and the connection domain “Email”. We can omit this phenomenon as connection domains are not part of the context diagram. Hence, only two phenomena remain to be added.

Listing 2 illustrates the operator call to add the phenomenon *cancelBooking* to the interface *G!*$\{$*browseAvailableHolidayOffers, bookHolidayOffer*$\}$ (details are given in Listing 4 in Sect. VI). The resulting evolved interface is shown in Fig. 8.

```
addPhenomena({'cancelBooking'},
              'G!{browseAvilableHolidayOffers,
                  bookHolidayOffer}' ,
              'G!{browseAvilableHolidayOffers,
                  bookHolidayOffer,
                  cancelBooking}')
```

Listing 2. Call of operator addPhenomena with corresponding parameters for phenomenon cancelBooking

In summary, we can state that we started with nine original requirements and seven problem diagrams. The evolution task introduced two evolution requirements, two new shared phenomena in the context diagram, one new problem diagram and modified one original requirement. The evolution requirements shared a relation with three original requirements. However, only two out of the seven original problem diagrams had to be considered. The modification of the original requirement was related to a change in the dynamic behavior. Therefore, it did not become apparent in the requirements analysis steps of EADIT as they only treat static aspects. The subsequent steps consist of:

- *revise abstract specification*: specifications for newly introduced subproblems must be derived. The specifications for modified or replaced subproblems must be adapted according to the changes in the subproblems. All other specifications remain untouched.
- *revise technical context diagram*: in some cases the technical infrastructure suffices. If not, adapt the technical context diagram, accordingly.
- *revise specify operations and data structures*: operations included in newly derived specifications need to be specified. Operations for modified or replaced specifications

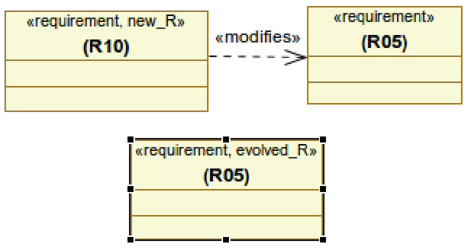

Fig. 9. Vacation Rentals: Relation of R10 to R05 and evolved R05

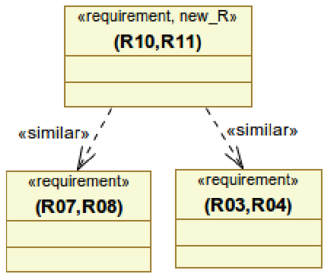

Fig. 10. Vacation Rentals: Relation of R10 and R11

must be adapted according to the changes in specifications. All other operations remain untouched.

- *adjust software life-cycle*: revise the life-cycle and incorporate new behavior appropriately.

The further steps of the evolution concern the software architecture and the code and are not treated in this paper.

VI. EADIT Tool Support

In this section we describe how our tool UML4PF can be enhanced to integrate evolution support. In Sect. V we saw that existing documents must be adapted or new ones must be created. This makes it necessary to mark the documents in order to keep track of them during the subsequent steps of the evolution. A convenient way to mark those documents is to assign special keywords to them. We use the stereotypes <<*evolved_X*>> and <<*new_Y*>>, where X and Y serve as placeholders for the evolved elements, e.g., requirement, interface, etc. (see bottom of Fig. 9 and top of Fig. 10). These stereotypes serve as special keywords throughout the complete evolution process to indicate additions, modifications, and replacements. These markings also help us to keep track of those elements that need to be validated at the end of the corresponding evolution step. After the final validation at the end of the evolution the stereotypes <<*evolved_X*>> and <<*new_Y*>> are removed. This prevents confusion, as we know that all elements that are marked with those stereotypes are relevant for the current evolution task and are not remainders of previous evolutions. To capture the relations between requirements we introduce the stereotypes <<*modifies*>>, <<*replaces*>>, <<*complements*>>, and <<*similar*>> extending the UML meta-class dependency. Examples are given in Figures 9 (top) and 10.

So far, we only used OCL constraints to check the validity of our diagram types. For the evolution we now provide OCL query expressions to retrieve specific information. An example for such a query expression is given in Listing 3.

The precondition given in Listing 3 states that we can only guarantee the result if the provided requirement (in parameter Req) exists (line 6).

The postcondition states that the function returns the names of the problem diagrams sharing a relationship with the provided evolution requirement. It works as follows: in lines 8 and 9 we select all packages that satisfy the select-statement, i.e., packages where the stereotype <<*ProblemDiagram*>> has been applied. To obtain all classes within these packages, we select all dependencies (clientDependency) where the target end includes the stereotype <<*isPart*>> (lines 10 and 11). As only classes possessing the stereotype <<*Requirement*>> are relevant, we select those classes (line 12). Within the set of these classes we select the first occurrence of the class where the name equals the provided evolution requirement indicated by Req. From this requirement we follow the dependencies and select those classes where the target class has the stereotype <<*Requirement*>> (line 16). We intersect the resulting sets (line 13) and select those (line 15) where the intersection is not empty (line 20). This results in a set of problem diagram names.

```
getRelatedProblemDiagrams(Req: String): String[]              1
                                                              2
PRE: Class .allInstances()                                    3
       ->select(getAppliedStereotypes() .name                 4
       ->includes('Requirement'))                             5
        ->exists(name=Req)                                    6
                                                              7
POST: result = Package .allInstances()                        8
   ->select(getAppliedStereotypes() .name                     9
       ->includes('ProblemDiagram'))
    -> select(pd | pd .clientDependency                       10
        ->select(getAppliedStereotypes() .name
                  ->includes('isPart')) .target               11
        ->select(oclIsTypeOf(Class))                          12
      ->intersection(                                         13
      Class .allInstances()                                   14
       ->select(getAppliedStereotypes() .name                 15
       ->includes('Requirement'))                             16
        ->select(name=Req)->asSequence()->first()             17
         .clientDependency .target                            18
        ->select(getAppliedStereotypes() .name                19
        ->includes('Requirement')))->size()<>0)               20
             .name
```

Listing 3. Query with pre-and postcondition for getRelatedProblemDiagrams

In Listing 3 we use the stereotype <<*isPart*>>. According to the UML superstructure specification [5], it is not possible that one UML element is part of several packages. Nevertheless, several UML tools (e.g.Papyrus, MagicDraw, Topcased, etc.) 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 is achieved with a dependency stereotype <<*isPart*>> from the package to all included elements (e.g., classes, interfaces, comments, dependencies, associations).

We also specify the operators presented in Sect. V by stating pre- and postconditions for the respective operator. An example of such an operator is given in Listing 4. The preconditions for this operator are that

- an interface exists that is named like the interface we

provide (parameter intfName; line 6) and
- the provided phenomena are not already owned operations of the interface (lines 8-10).

```
1  addPhenomena(phenNames: String[],
2                intfName: String,
3                newIntfName: String)
4
5  PRE:
6   Interface .allInstances()->exists(name=intfName)
7  and
8   phenNames->forAll(pn |
9     not Interface
          .allInstances()->select(name=intfName)
10    .oclAsType(Interface) .ownedOperation
          .name->includes(pn))
11
12 POST:
13  Interface .allInstances()->exists(name=newintfName)
14 and
15  Interface
       .allInstances()->select(name=newintfName)
       .getAppliedStereotypes()
       .name->includes('evolved_interface')
16 and
17  not Interface
       .allInstances()->exists(name=intfName)
18 and
19  phenNames->forAll(pn |
20    Interface
         .allInstances()->select(name=newintfName)
21      .oclAsType(Interface) .ownedOperation
           .name->includes(pn)
22    and
23     Interface
          .allInstances()->select(name=newintfName)
24      .oclAsType(Interface)
           .ownedOperation->select(name=pn)
           .getAppliedStereotypes()
           .name->includes('new_op')
25 )
26 and
27   Interface
         .allInstances()->select(name=newintfName)
28   .oclAsType(Interface) .ownedOperation->size() =
29   Interface
         .allInstances()@pre->select(name=intfName)
30   .oclAsType(Interface)
         .ownedOperation->size()+phenNames->size()
```

Listing 4. Operator *addPhenomena* with pre-and postcondition

The postcondition is fulfilled, if

- an interface with the new name (parameter newIntfName) exists (line 13),
- the stereotype <<*evolved_interface*>> has been assigned to the new interface (line 15),
- the old interface (parameter intfName) does not exist anymore (line 17),
- the new phenomena are now owned operations of the new interface (lines 19-21),
- the stereotype <<*new_op*>> has been assigned to all new phenomena (lines 23 and 24), and
- the number of operations owned by the new interface (lines 27 and 28) is increased by the number of added phenomena compared to the operations owned by the old interface (lines 27-30).

Another example for an operator is given in Listing 5. In the precondition we state that no domain should be existing that as the same name as the domain to add (parameter domName) (line 3).

The postcondition verifies that afterwards there exists a domain with the provided name (line 5) and that the provided stereotype (parameter domType) as well as the stereotype <<*new_domain*>> has been assigned to the domain (lines 6-9).

```
addDomain(domName :String, domType: String)            1
                                                       2
PRE: not                                               3
    Class.allInstances()->exists(name='domName')
                                                       4
POST: Class.allInstances() ->exists(name='domName')    5
      and                                              6
      Class.allInstances() ->select(name='domName')    7
          .getAppliedStereotypes() .name
          ->includes('domTyppe')
      and                                              8
      Class.allInstances() ->select(name='domName')    9
          .getAppliedStereotypes() .name
          ->includes('new_domain')
```

Listing 5. Operator *addDomain* with pre-and postcondition

The general procedure to use the tool for an evolution task is as follows:

- Load the existing development project into eclipse.
- Create new classes for the evolution requirements and mark them with the stereotype <<*new_R*>>.
- Relate the requirements to each other according to Sect. V-A1.
- Choose one evolution requirement and query the related problem diagrams, if any. Proceed as described in Sect. V-A2.
- Revise the context diagram according to Sect. V-B.
  Our paper concludes with this step. However, to accomplish the evolution task the remaining steps described at the end of Sect. V (as well as adjusting the implementation and testing) must be performed. The final steps of the evolution method are then to:
- remove all stereotypes starting with <<*new_*>> and <<*evolved_*>>.
- automatically check integrity and consistency constraints provided by UML4PF.

In summary the tool support described in Sect. IV is enhanced by :

- an extended UML profile providing stereotypes specific to the evolution, e.g., <<*evolved_R*>>, <<*modifies*>>.
- OCL expressions to query model information.
- a set of operators specified by pre- and postconditions expressed in OCL.

VII. Related Work

This work takes up ideas from modern software engineering approaches and processes, such as the Rational Unified Process (RUP) [15], Model-Driven Architecture (MDA) [16], and Service-Oriented Architecture (SOA) [17]. All these approaches are model-driven, which means that in principle, an evolution process for them can be defined in a similar way as for the development process ADIT presented here.

The work of O'Cinnéide and Nixon [18] aims at applying design patterns to existing legacy code in a highly automated way. They target code refactorings. Their approach is based on a semi-formal description of the transformations themselves, needed in order to make the changes in the code happen. They describe precisely the transformation itself and under which pre- and postconditions it can successfully be applied. Our approach not just performs code changes, but updates all development documents.

Researchers have used versions and histories to analyze different aspects considering software evolution. Ducasse et al. [19] propose a history meta-model named HISMO. A history is defined as a sequence of versions. The approach is based on transformations aimed at extracting history properties out of structural relationships. Our approach does not consider histories and how to analyze them. In contrast, we introduce a method for manipulating an existing software and its corresponding documentation in a systematic way.

Detecting logical coupling to identify dependencies among modules has been the research topic of Gall et al. [20]. Those dependencies can be used to estimate the effort needed to carry out maintenance tasks, to name an example. Descriptions in change reports are used to verify the detected couplings. The technique is not designed to change the functionality of a given software.

Pizka [21] investigates software evolution at run-time. Evolving a software system at run-time puts even more demands on the method used than ordinary software systems. Our approach does not take run-time evolution into account.

Sillito et al. [22] conducted a survey to capture the main questions programmers ask when confronted with an evolution task. These fit well to our method as they can be used to refine the understanding of the software at hand, especially in later phases.

We agree with Mens and D'Hondt [23] that it is necessary to treat and support evolution throughout all development phases. They extend the UML meta-model by their so-called evolution contracts for that reason. The aim is to automatically detect conflicts that may arise when evolving the same UML model in parallel. This mechanism can very well be integrated into our method to enhance the detection of inconsistencies and conflicts. Our approach, however, goes beyond this detection process. It strives towards an integral method for software evolution guiding the software engineer in actually performing an evolution task throughout all development phases.

The field of software evolution cannot be examined in isolation as it has contact points with other disciplines of software engineering. An example is the work of Demeyer et al. [24]. They provide a pattern system for object-oriented reengineering tasks. Since software evolution usually involves some reengineering and also refactoring [25] efforts, it is only natural that known and successful techniques are applied in software evolution, as well. Software evolution, however, goes beyond restructuring source code. Its main goal is to change the functionality of a given software.

Furthermore, software evolution can profit from the research performed in the fields of feature location e.g. [26], [27], re-documentation, e.g. [28], [29], and agile development processes such as extreme programming [30].

Lencastre et al. [31] define a meta-model for problem frames using UML. Their meta-model considers Jackson's whole requirements analysis 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 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 [31]).

We agree with Haley [32] 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.

Charfi et al. [33] 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. [34] 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 other tools supporting the work with problem frames on the semantic level, as does UML4PF.

VIII. Conclusion

In this paper, we have presented an evolution method and an extension to our UML profile [10] for problem frames to handle evolutionary requirements evolution as first steps of a complete software evolution process. In this profile extension, we defined a set of stereotypes supporting software evolution. We introduced OCL queries in addition to OCL constraints. One query has been presented in this paper. Furthermore, we introduced operators that help the engineer in performing the necessary changes in the different steps.

In summary, our concept has the following advantages:

- The evolution method is embedded in a development process.
- We have defined a number of rules and operators that guide the engineer when performing software evolution.
- Statements expressed using our profile help to structure and classify the evolution task. For example, it is possible to trace allartifacts that refer to one evolution requirement.

- Artifacts from the analysis phase that are part of a model created with our profile can be re-used in later phases in the software development process.
- The notation used is based on UML. UML is commonly used in software engineering and many developers are able to read our models.
- The evolution method is tool-supported.

In the future, we plan to extend our tool to support identifying priorities within the set of evolution requirement. Currently, we are working on expanding the evolution method to include later phases of the development cycle based on the method presented in this paper. Furthermore, we plan on conducting further evolution tasks on open source projects to validate the method as well as the tool.

References

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

[2] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution - the nineties view," in *METRICS '97: Proceedings of the 4th International Symposium on Software Metrics*. Washington, DC, USA: IEEE Computer Society, 1997, p. 20.

[3] D. L. Parnas, "Software aging," in *ICSE '94: Proc. of the 16th Int. Conf. on Software Engineering*. Los Alamitos, CA, USA: IEEE Comp. Soc. Press, 1994, pp. 279–287.

[4] G. Engels, M. Goedicke, U. Goltz, A. Rausch, and R. Reussner, "Design for Future – Legacy-Probleme von morgen vermeidbar?" *Informatik-Spektrum*, 2009.

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

[6] U. R. T. Force, *OMG Object Constraint Language: Reference*, May 2006, http://www.omg.org/docs/formal/06-05-01.pdf.

[7] D. Hatebur and M. Heisel, "Deriving software architectures from problem descriptions," in *Software Engineering 2009 - Workshopband*. GI, 2009, pp. 383–302.

[8] I. Côté, D. Hatebur, M. Heisel, H. Schmidt, and I. Wentzlaff, "A systematic account of problem frames," in *Proceedings of the European Conference on Pattern Languages of Programs (EuroPLoP 2007)*. Universitätsverlag Konstanz, 2008.

[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*, 2004, pp. 61–76.

[10] D. Hatebur and M. Heisel, "Making pattern- and model-based software development more rigorous," in *Proceedings of 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China*, ser. LNCS 6447, J. S. Dong and H. Zhu, Eds. Springer, 2010.

[11] "Eclipse - An Open Development Platform," May 2008, http://www.eclipse.org/.

[12] "Eclipse Modeling Framework Project (EMF)," May 2008, http://www.eclipse.org/modeling/emf/.

[13] "XMI - XML Metadata Interchange," May 2008, http://www.omg.org/docs/formal/05-09-01.pdf.

[14] "Papyrus UML Modelling Tool," Jan 2010, http://www.papyusuml.org/.

[15] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. Addison-Wesley, 1999.

[16] S. J. Mellor, K. Scott, A. Uhl, and D. Weise, *MDA Distilled*. Addison-Wesley Professional, 2004.

[17] T. Erl, *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall PTR, 2005.

[18] M. O'Cinnéide and P. Nixon, "A methodology for the automated introduction of design patterns," in *ICSM '99: Proc. of the IEEE Int. Conf. on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1999, p. 463.

[19] S. Ducasse, T. Gîrba, and J.-M. Favre, "Modeling software evolution by treating history as a first class entity," in *Proc. on Software Evolution Through Transformation (SETra 2004)*. Amsterdam: Elsevier, 2004, pp. 75–86. [Online]. Available: http://www.iam.unibe.ch/ scg/Archive/Papers/Duca04fHismo.pdf

[20] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *ICSM '98: Proc. of the Int. Conf. on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1998, p. 190.

[21] M. Pizka, "STA – a conceptual model for system evolution," in *Int. Conf. on Software Maintenance*. Montreal, Canada: IEEE CS Press, Oct. 2002, pp. 462 – 469.

[22] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *SIGSOFT '06/FSE-14: Proc. of the 14th ACM SIGSOFT Int. Symposium. on Foundation of Software Engineering*. New York, NY, USA: ACM, 2006, pp. 23–34.

[23] T. Mens and T. D'Hondt, "Automating support for software evolution in UML," *Automated Software Engineering Journal*, vol. 7, no. 1, pp. 39–59, February 2000.

[24] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[25] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[26] D. Edwards, S. Simmons, and N. Wilde, "An approach to feature location in distributed systems," in *Journal of Systems and Software*, 2006.

[27] R. Koschke and J. Quante, "On dynamic feature location," in *ASE '05: Proc. of the 20th IEEE/ACM Int. Conf. on Automated Software Engineering*. New York, NY, USA: ACM, 2005, pp. 86–95.

[28] F. Chen and H. Yang, "Model oriented evolutionary redocumentation," in *COMPSAC '07: Proc. of the 31st Annual Int. Computer Software and Applications Conference - Vol. 1- (COMPSAC 2007)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 543–548.

[29] K. Wong, S. R. Tilley, H. A. Muller, M. D. Storey, and T. A. Corbi, "Structural redocumentation: A case study," *IEEE Software*, vol. 12, pp. 46–54, 1995.

[30] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[31] 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.

[32] C. B. Haley, "Using problem frames with distributed architectures: A case for cardinality on interfaces," *The Second International Software Requirements to Architectures Workshop (STRAW'03)*, May 2003.

[33] A. Charfi, A. Gamatié, A. Honoré, J.-L. Dekeyser, and M. Abid, "Validation de modèles dans un cadre d'IDM dédié à la conception de systèmes sur puce," in *4èmes Jounées sur l'Ingénierie Dirigée par les Modèles (IDM 08)*, 2008.

[34] P. Colombo, V. del Bianco, and L. Lavazza, "Towards the integration of SysML and problem frames," in *IWAAPF '08: Proceedings of the 3rd international workshop on Applications and advances of problem frames*. New York, NY, USA: ACM, 2008, pp. 1–8.