Using Constraints in Teaching Software Modeling

Dan Chiorean[1], Vladiela Petraşcu[1], and Ileana Ober[2]

[1] Babeş-Bolyai University, Cluj-Napoca, Romania
{chiorean,vladi}@cs.ubbcluj.ro
[2] Université Paul Sabatier, Toulouse, France
ober@irit.fr

Abstract. The paper[1] presents an approach to teaching software modeling that has been put into practice at the Babeş-Bolyai University of Cluj-Napoca and Paul Sabatier University in Toulouse. This aims at persuading students of the advantages deriving from the usage of rigorous models. The development of such models, which relies on the Design by Contract technique, is a must in the context of the Model-Driven Engineering paradigm. Another goal of our approach is for students to acquire core software modeling principles and techniques, allowing them to identify and avoid various types of pitfalls enclosed by the modeling examples posted on web. Following a decade of use and teaching of OCL, we have concluded that starting with a defense of the necessity and advantages offered by the use of constraints (an "inverted curriculum" approach) is a far more efficient teaching method compared to a pure technical introduction into the language itself.

Keywords: rigorous modeling, OCL specifications, meaningful specifications, efficient specifications, model understanding.

1 Introduction

In MDE (Model-Driven Engineering), "models are not only the primary artifacts of development, they are also the primary means by which developers and other systems understand, interact with, configure and modify the runtime behavior of software" [9].

The final goal of MDE technologies is to generate easy maintainable applications in various programming languages, such as Java, C#, C++, and so on. This requires starting from complete and unambiguous models, specified by means of rigorous modeling languages. The current technique used to specify languages is metamodeling. Irrespective of the abstraction level involved (meta-meta, meta or model level), in order to achieve rigorous model descriptions, the use of constraints (assertions) is a must.

The use of assertions in software development is promoted by the Design by Contract technique. In [11], the author identifies four applications of their use,

[1] This paper is an extended version of [7].

J. Kienzle (Ed.): MODELS 2011 Workshops, LNCS 7167, pp. 25–39, 2012.

namely: help in writing correct software, documentation aid, support for testing, debugging and quality assurance and support for software fault tolerance.

Working with assertions is therefore a core technique that model designers must manage. Despite this, practice shows that Design by Contract is not yet employed at its full potential. This may be due to both the lack of relevant examples showing the advantages of using concrete constraint languages (such as OCL) and the availability of a large number of examples which, at best, cause confusion among readers. An experience of over ten years in teaching OCL to computer science students (at both bachelor and master levels) has allowed us to conclude that, apart from providing positive recommendations (books, papers, tools), warning potential OCL users on the pitfalls enclosed by negative examples is mandatory. As web users, students are exposed to both clear, well-written documents and to documents containing various drawbacks, on whose potential occurrence teachers have the duty of raising warnings. However, merely showing that particular models or specifications are inadequate or even incorrect with respect to the purpose they were created for is not enough. Presenting at least one correct solution and arguing on its advantages is a must.

Complementing models with OCL is meant at eliminating specifications ambiguities, thus increasing rigor, reaching a full and clear definition of query operations, as well as promoting Design by Contract through the specification of pre and post-conditions.

The development of models and applications takes place as an iterative incremental process, allowing return to earlier stages whenever the case. Enhancing models with OCL specifications facilitates their deeper understanding, through both rigor and extra detail. Whenever the results of evaluating OCL specifications suggest a model change, this change should only be done if the new version is more convenient compared to the previous ones, as a whole. The use of OCL specifications should contribute to the requirements validation. An application is considered as finished only when there is full compliance among its requirements, its model, and the application itself.

The remaining of this paper is organized as follows. Section 2 explains the reasons why teaching OCL through examples integrated in models is more advantageous compared to the classical way of teaching OCL. In Section 3, we argue on the necessity of understanding the model's semantics, which is the first prerequisite for achieving a good specification. Section 4 emphasizes the fact that we need to consider several modeling solutions to a problem and choose the most convenient one with respect to the aspects under consideration. Section 5 shows the role of OCL in specifying the various model uses, while Section 6 justifies through an example the need of using snapshots for validating specifications. The paper ends with conclusions in Section 7.

2 Teaching OCL through Examples Integrated in Models

There are various ways of teaching OCL. The classical approach emphasizes the main language features: its declarative nature and first order logic roots, the type system, the management of undefined values, the collection types together with their operations and syntax specificities, and so on [6], [4]. Many of the examples used to introduce collections employ expressions with literals, which are context-independent and easy to understand.

OCL is a textual language which complements MOF (Meta Object Facility)-based modeling languages. The students' interest in understanding and using the language increases if there are persuaded of the advantages earned from enriching models with OCL specifications. To convince students of the usefulness of OCL, the chosen examples should be suggestive in terms of models and enlightening in terms of earned benefits. That is the reason why we have considered more appropriate taking an "inverted curriculum"-type of approach, by introducing OCL through examples in which the OCL specifications are naturally integrated in models. Unfortunately, along with positive OCL specification examples, the existing literature offers also plenty of negative ones, starting with the WFRs (well-formedness rules) that define the static semantics of modeling languages. The negative examples may wrongly influence students' perception. Therefore, we argue that a major issue in teaching OCL to students is explaining them the basic principles that should be obeyed when designing OCL specifications, principles that should help them avoid potential pitfalls.

Two modeling examples that have been probably meant to argue for the use and usefulness of OCL (taking into account the title of the paper in question) are those presented in [15]. The examples and solutions proposed by this article provide an excellent framework for highlighting important aspects that should be taken into account within the modeling process. In the second semester of the 2010-2011 academic year, we have used these examples in order to warn students on the pitfalls that should be avoided when enriching models with OCL specifications.

3 Understanding the Model's Semantics

A model is an abstract description of a problem from a particular viewpoint, given by its intended usage. The design model represents one of the possible solutions to the requirements of the problem to solve. It is therefore essential for the students to realize the necessity of choosing a suitable solution with respect to the aspects under consideration. The first prerequisite for designing such a model is a full understanding of the problem at hand, reflected in a thorough informal requirements specification. Nygaard's statement "Programming is Understanding" [16] is to be read as "Modeling is Understanding", since "Object-oriented development promotes the view that programming is modeling" [12]. Understanding is generally acquired through an iterative and incremental process, in which OCL specifications play a major role. This is due to the fact that "if you don't understand something, you can't code it, and you gain understanding trying to code it." [16]. To be rigorous, "understanding" is only the first mandatory step to be accomplished in both programming and modeling. Finding the problem solution and describing it intelligibly must follow and take advantage of problem understanding. The informal specification of constraints is part of model understanding. Whenever constraints are missing from the initial problem requirements, they should be added in the process of validation and refinement of the informal problem specification.

To illustrate these statements, we will use one of the modeling examples from [15] (shown in Fig. 1), which describes parents-children relationships in a community of persons. The model requirements description is incomplete with respect to both its intended functionalities and its contained information. In such cases, the model specification, both the graphical and the complementary textual one (through Additional Operations - AOs, invariants, pre and post-conditions), should contribute to enriching the requirements description. The process is iterative and incremental, marked by repeated discussions among clients and developers, until the convergence of views from both parties.

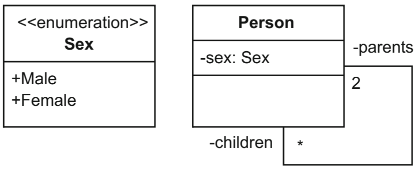

Fig. 1. Genealogical tree model [15]

The proposed solution should allow a correct management of information related to persons, even when this information is incomplete. Unknown ancestors of a particular person is such a case (sometimes not even the natural parents are known). For such cases, the model provided in [15] and reproduced in Fig. 1 is inadequate, due to the infinite recursion induced by the self-association requiring each person to have valid references towards both parents. Snapshots containing persons with at least one parent reference missing will be thus qualified as invalid.

Both this problem and its solution, consisting in relaxing the `parents` multiplicity to 0..2, are now "classical" [5]. Partial or total lack of references (1 or 0 multiplicity) indicates that either one or both parents are unknown at the time.

The only constraint imposed in [15] on the above-mentioned model requires the parents of a person to be of different sexes. Following, there is its OCL specification, as given in [15].

```
self.parents->asSequence()->at(1).sex<>self.parents->asSequence()->at(2).sex
```

The technical quality of formal constraints (stated in either OCL or a different constraint language) follows from the fulfillment of several quality factors. Among them, there is the conformance of their behavior to their informal counterparts, the intelligibility of their formal specification, the debugging support offered in case of constraint violation, the similarity of results following both their evaluation in different constraint-supporting tools and the evaluation of their programming language equivalents (code snippets resulted from translating constraints into a programming language, using the constraint-supporting tools in question)[2]. Based on these considerations, the above specification, although apparently correct, encloses a few pitfalls:

1. In case at least one parent reference is missing and the multiplicity is 2, the evaluation of WFRs should signal the lack of conformance among the multiplicity of links between instances and the multiplicity of their corresponding association. To be meaningful, the evaluation of model-level constraints should be performed only in case the model satisfies all WFRs. Unfortunately, such model compilability checks are not current practice. In case the `parents` multiplicity is 0..2 and one of the parents is left unspecified, the model will comply with the WFRs, but the constraint evaluation will end up in an exception when trying to access the missing parent (due to the `at(2)` call).
2. In case there are valid references to both parents, but the sex of one of them is not specified, the value of the corresponding subexpression is `undefined` and the whole expression reduces to either `Sex::Male <> undefined` or `Sex::Female <> undefined`. This later expressions provide tool-dependent evaluation results (`true` in case of USE [1] or Dresden OCL [14] and `undefined` in case of OCLE [10]). The results produced by OCLE comply with the latest OCL 2.3 specification [13]. However, as the topic of evaluating undefined values has not yet reached a common agreement, students should be warned on this.
3. Moreover, in case one of the parents' sex is undefined, the code generated for the above constraint will provide evaluation results which depend on the position of the `undefined` value with respect to the comparison operator. According to the tests we have performed[3], the Java code generated by OCLE and Dresden OCL throws a `NullPointerException` when the `undefined` value is located at the left of the `<>` operator, while evaluating to `true` in case the `undefined` value is at the right of the operator and the reference at the left is a valid one. As we are in the context of the MDE paradigm, which relies extensively on automatic code generation, the results provided by the execution of the code corresponding to constraints is an aspect that has to be taken into account when judging the quality of the formal constraints in question.
4. The OCL expression would have been simpler (not needing an `asSequence()` call), in case an ordering relation on `parents` had been imposed at the model level.

Similar to most of the activities involved in software production, the specification of assertions is an iterative and incremental process. That is why, in the following, we will illustrate such a process for the considered family case study. For the purpose of this section, we will work with the model from Fig. 1, assuming

[2] Apart from the technical issues involved in evaluating the quality of assertions, there is also an efficiency issue, concerned with aspects such as the efficiency of the assertion specifications themselves, the amount of undesirable system states that can be monitored by using assertions, as well as their level of detail.

[3] The corresponding Java/AspectJ projects can be downloaded from [2].

though that the multiplicity of the `parents` reference has been set to `0..2`, so as to avoid infinite recursion.

Ordering the `parents` collection with respect to sex (such that the first element points to the mother and the second to the father) allows writing an invariant that is more detailed compared to the one proposed in [15] for the constraint regarding the parents' sex. Following, there is the OCL specification we propose in this respect, when both parents are known. In case of invariant violation, the debugging information is precise, allowing to easily eliminate the error's cause.

```
context Person
 inv parentsSexP1:
  self.parents->size() = 2 implies
   Sex::Female = self.parents->first().sex and
   Sex::Male = self.parents->last().sex
```

When any of the parents' sex is `undefined`, the invariant above evaluates to `false` in Dresden OCL and to `undefined` in OCLE. In similar circumstances, both Java code snippets generated for this invariant by the two tools return `false` when executed. Therefore, this invariant shape overcomes the drawbacks of the one from [15] previously pointed at items 1, 3 and 4. The triggering of a `NullPointerException` by the generated code in case of absence of one of the parents' sex has been avoided by placing the defined values (the `Sex::Female` and `Sex::Male` literals) on the left-hand side of equalities.

The solution to the problem mentioned at item 2 above comes from obeying to the separation of concerns principle. In order to avoid comparisons involving `undefined` values, whose results may vary with the OCL-supporting tool used, the equality tests of the parents' sex with the corresponding enumeration literals should be conditioned by both of them being specified. Such a solution is illustrated by means of the invariant proposal below.

```
context Person
 inv parentsSexP2:
  self.parents->size() = 2 implies
  ( let mother = self.parents->first() in
    let father = self.parents->last() in
     if (not mother.sex.oclIsUndefined() and not father.sex.oclIsUndefined())
     then mother.sex = Sex::Female and father.sex = Sex::Male
     else false
     endif
  )
```

The invariant above evaluates to `false` when any of the parents' sex is `undefined`, as well as when they are both defined but set inappropriately (the first parent's sex is not `Sex::Female` or the second is not `Sex::Male`, as previously established by the ordering rule). The evaluation results are the same for both the OCL constraint and its Java equivalent, irrespective of the tool used, OCLE or Dresden OCL. Therefore, this last invariant shape provides solutions to all pitfalls previously detected for its analogue from [15].

Yet, a correct understanding of the model in question leads to the conclusion that the mere constraint regarding the parents' sex is insufficient, despite its explicit specification for each parent. As rightly noticed in [5], a person cannot be its own child. A corresponding OCL constraint should be therefore explicitly specified.

```
context Person
  inv notSelfParent:
    self.parents->select(p | p = self)->isEmpty()
```

However, restricting the age difference among parents and children to be at least the minimum age starting from which human reproduction is possible (we have considered the age of sixteen) leads to a stronger and finer constraint than the previous, that may be stated as follows.

```
context Person
  inv parentsAge:
    self.parents->reject(p | p.age - self.age >= 16)->isEmpty()
```

In the above expression, each `Person` is assumed to own an `age` attribute. In case both the contextual instance and its parents have valid values for the `age` slot, the `reject(...)` subexpression evaluates to the collection of parents breaking the constraint in question.

The fulfillment of this constraint could be also required at any point in the construction of the genealogical tree. Assuming any parent to be created prior to any of its children, this restriction could be stated by means of the precondition included in the contract below.

```
context Person::addChildren(p:Person)
  pre childrenAge:
    self.children->excludes(p) and self.age - p.age >= 16
  post chidrenAge:
    self.children->includes(p)
```

The conclusion that emerges so far is that the lack of OCL specifications prohibiting undesired model instances (such as parents having the same sex, self-parentship or the lack of a minimum age difference among parents and children) seriously compromises model's integrity. The first prerequisite for models to reach their purpose is to have a complete and correct specification of requirements, and to deeply understand them. An incomplete specification reveals its limits when trying to answer questions on various situations that may arise. Specifying and evaluating OCL constraints should enable us to identify and eliminate bugs, by correcting the requirements and the OCL specifications themselves. Moreover, in the context of MDE, care should be taken to the shape of constraint specifications, ensuring that their evaluation using OCL-supporting tools provides identical results to those obtained by executing their equivalent code generated by those tools. Another conclusion, as important, is that the model proposed in the analyzed paper does not fully meet the needs of the addressed problem[4] and we are therefore invited to seek for a better solution.

[4] In case there is a single parent specified, we have no means to check whether the sex has been set appropriately, according to its role (mother or father) and we may need extra attributes (e.g. `age`) for specifying finer constraints.

4 Modeling Alternatives

A model equivalent to that of Fig. 1, but which is more adequate to the specification of the required constraints, is the one illustrated in Fig. 2. The model in question contains two recursive associations: one named `MotherChildren`, with roles `mother[0..1]` and `mChildren[0..*]` and the other `FatherChildren`, with roles `father[0..1]` and `fChildren[0..*]`.

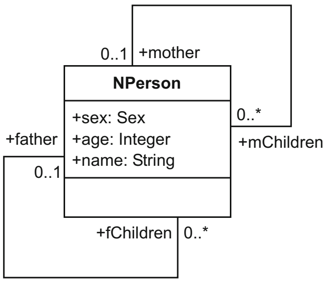

Fig. 2. An alternative model for expressing parents-children relationships

Within this model, the constraint regarding the parent' sex can be stated as proposed below.

```
context NPerson
 inv parentsSexP1:
  (self.mother->size() = 1 implies Sex::Female = self.mother.sex) and
  (self.father->size() = 1 implies Sex::Male = self.father.sex)
```

Compared to its equivalent constraint stated for the model in Fig. 1, the above one is wider, since it also covers the case with a single parent and checks the sex constraint corresponding to the parent in question. As previously pointed out, the problem with the initial model (the one in Fig. 1) is that we cannot count on an ordering when there is a single parent reference available. The parent in question would always be on the first position, irrespective of its sex. As opposed to this, in Fig. 2, the parents' roles are explicitely specified, with no extra memory required. When at least one parent's sex is `undefined`, the evaluation of this invariant returns `undefined` in OCLE and `false` in Dresden OCL, while the execution of the corresponding Java code outputs `false` in both cases. Given the invariant shape, the identification of the person breaking it is quite straightforward, therefore the problem can be rapidly fixed.

An alternative invariant shape, providing the same evaluation result in both OCLE and Dresden OCL, for both OCL and Java code is the one below.

```
context NPerson
 inv parentsSexP2:
  (self.mother->size() = 1 implies
   (let ms:Sex = self.mother.sex in
     if not ms.oclIsUndefined() then ms = Sex::Female
     else false endif )
  ) and
  (self.father->size() = 1 implies
   (let fs:Sex = self.father.sex in
     if not fs.oclIsUndefined() then fs = Sex::Male
     else false endif)
  )
```

With respect to the second constraint, we propose the following specification in context of the model from Fig. 2.

```
context NPerson
 inv parentsAge:
  self.mChildren->reject(p | self.age - p.age >= 16)->isEmpty() and
  self.fChildren->reject(p | self.age - p.age >= 16)->isEmpty()
```

The `parentsAge` invariant above uses one of the specification patterns that we have proposed in [8] for the *For All* constraint pattern. If we were to follow the classical specification patterns available in the literature, the invariant would have looked as follows.

```
context NPerson
 inv parentsAgeL:
  self.mChildren->forAll(p | self.age - p.age >= 16) and
  self.fChildren->forAll(p | self.age - p.age >= 16)
```

A simple analysis of these two proposals reveals that the `parentsAgeL` invariant shape is closer to first order logic. However, in case of constraint violation, this does not provide any useful information concerning those persons breaking the invariant, as the first one does. The specification style used for the `parentsAge` invariant offers model-debugging support [8], a major concern when writing assertions.

The corresponding pre and post-conditions are similar to their equivalents from the previous section, therefore their specification could be left to students, as homework.

5 Explaining the Intended Model Uses

Any requirements specification should include a detailed description of the intended model uses. In case of the model under consideration, it is important to know what kind of information may be required from it. Is it merely the list of parents and that of all ancestors? Do we want the list of ancestors ordered, with each element containing parents-related information, in case such information is available? Do we only need information regarding the male descendents of a person?

In case of the initial model in which the recursive association is ordered, the list of all ancestors of a person can be easily computed as follows.

```
context Person
  def allAncestors():Sequence(Person) =
    self.parents->union(self.parents.allAncestors())
```

The evaluation result for the constraint above is correct only if we assume the genealogical tree as loop-free. This latter constraint is implied by the one restricting the minimum age difference between parents and children. In the absence of this assumption, the OCL expression's complexity increases.

A simpler alternative for this case employs the semantic closure operation on collections. This operation, now included in OCL 2.3, has been implemented in OCLE ever since its first release and returns a set.

```
context Person
  def allAncestors():Sequence(Person) =
   (Sequence{self}->closure(p | p.parents))->asSequence()
```

The advantages offered by the modeling solution proposed in Fig. 2 are clear in case we are interested to compute all ancestors of a person, specifying explicitly which of them are unknown (not stored in the database). Following, there is a possible OCL query to be used in this purpose, that employs the tuple type.

```
context NPerson
 def tParents: TupleType(ch:NPerson, mo:NPerson, fa:NPerson) =
 Tuple{ch = self, mo = self.mother, fa = self.father}

 def allTParents: Sequence(TupleType(ch:NPerson, mo:NPerson, fa:NPerson)) =
 Sequence{self.tParents}->closure(i | Sequence{i.mo.tParents, i.fa.tParents})
                                      ->asSequence()->prepend(self.tParents)
```

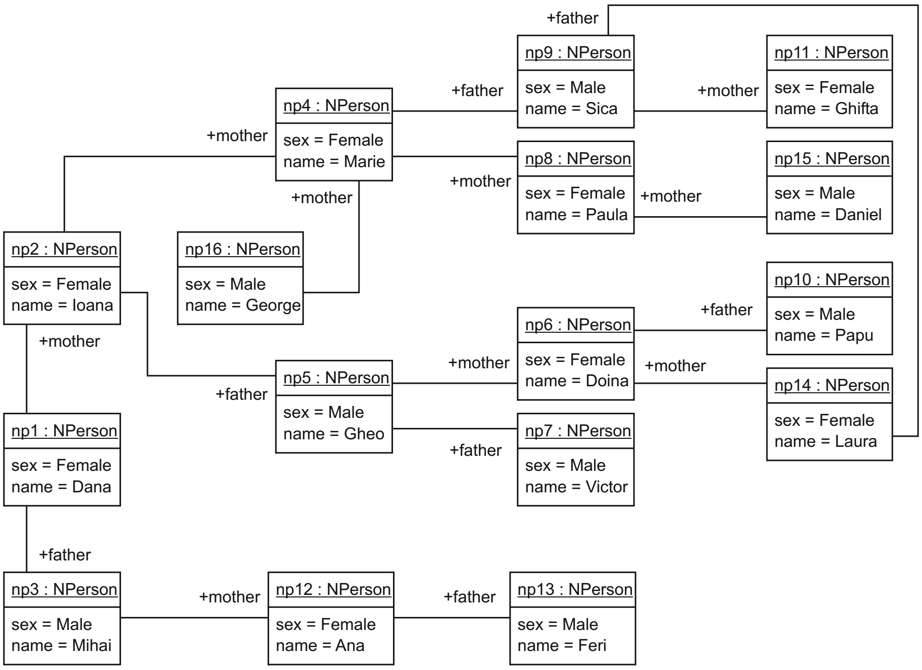

Fig. 3. Sample snapshot of the model from Fig. 2

Evaluating in OCLE the family tree presented in Fig. 3, we obtain:

```
Sequence{Tuple{np1,np2,np3,}, Tuple{np2,np4,np5}, Tuple{np3,np12,Undefined},
  Tuple{np4,np8,np9}, Tuple{np5,np6,np7}, Tuple{np12,Undefined,np13},
  Undefined, Tuple{np8,Undefined,Undefined}, Tuple{np9,np11,Undefined},
  Tuple{np6,Undefined,np10}, Tuple{np7,Undefined,Undefined},
  Tuple{np13,Undefined,Undefined}, Tuple{np11,Undefined,Undefined},
  Tuple{np10,Undefined,Undefined}
}
```

Since the members of each tuple are (child, mother, father), in this particular order, the analysis of the above evaluation result allows an easy representation of the corresponding genealogical tree.

With respect to a potential query meant to compute all descendants of a person, the only difference between the two proposed models concerns the computation of a person's children. In this respect, the model in Fig. 1 already contains a `children` reference, while in case of the one from Fig. 2, a corresponding query needs to be defined, as shown below.

```
context NPerson
 def children: Set(NPerson) =
  if self.sex = Sex::Female
  then self.m_children
  else self.f_children
  endif
```

6 Using Snapshots to Better Understand and Improve the Requirements and the Model

One of the primary roles of constraints is to avoid different interpretations of the same model. Therefore, the specification process must be seen as an invitation for a complete and rigorous description of the problem, including the constraints that are part of the model. The model must conform to the informally described requirements, even before attaching constraints. In case this condition is not fulfilled, the constraints specification process must ask for additional information, meant to support an improved description of requirements, a deeper understanding of the problem, and by consequence, a clear model specification.

Despite its importance, as far as we know, this issue has not been approached in the literature. That is why, in the following, we will try to analyze the second example presented in [15], concerning a library model. This example aims to model the contractual relationships between a library, its users and companies associated with the library. The only informal specification provided is the following: "In this example, we'll assume that the library offers a subscription to each person employed in an associated company. In this case, the employee does not have a contract with the library but with the society he works for, instead. So we add the following constraint (also shown in Figure 10): ...".

First of all, we would like to remind the definition of a contract, as taken from [3]: "A binding agreement between two or more parties for performing, or refraining from performing, some specified act(s) in exchange for lawful consideration." According to this definition and to the informal description of requirements, we conclude that, in our case, the parts in the contract are: the user on the one hand, and the library or the company, on the other hand. As one of the involved parts is always the user, the other part is either the library (in case the user is not employed in any of the library's associated companies), or the company (in case the user is an employee of the company in question).

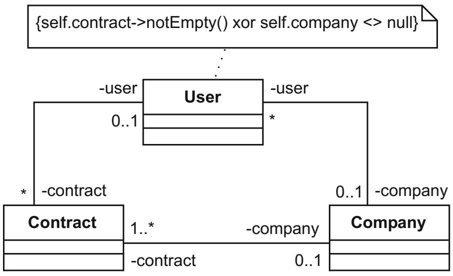

Fig. 4. The library model from [15], Figure 10

Regarding the conformance among requirements, on the one side, and model, on the other side (the class diagram, the invariant presented in Figure 10 and the snapshots given in Figures 12 and 13 of [15]), several questions arise. Since a thorough analysis is not allowed by the space constraints of this paper, in the following, we will only approach the major aspects related to the probable usage of the model. In our opinion, this concerns the information system of a library, that stores information about library users, associated companies, books, book copies and loans. The library may have several users and different associated companies.

Since the `Library` concept is missing from the model proposed in [15], we have no guaranty that, in case the user is unemployed, the second participant to the contract is the library. Moreover, in case the user is employed, the invariant proposed in [15] does not ensure that both the user and the corresponding company are the participants to the contract. As a solution to this, we propose an improved model for the Library case study (see Fig. 5), as well as two corresponding invariants, in the context of `Contract` and `User`, respectively.

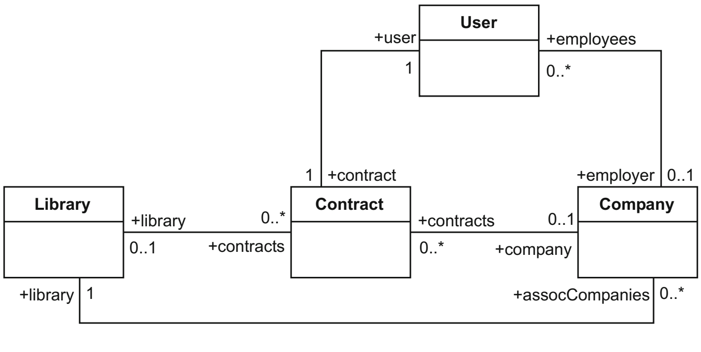

Fig. 5. A revised version of an excerpt of the library model from [15]

```
context Contract
 inv onlyOneSecondParticipant:
  self.library->isEmpty() xor self.company->isEmpty()

context User
 inv theContractIsWithTheEmployer:
  if self.employer->isEmpty()
   then self.contract.library->notEmpty()
   else self.employer = self.contract.company
  endif
```

The above constraints forbid situations like those from Fig. 6 (in which the user `u1` has a contract `c1` both with the library `l1` and the company `comp1`) and Fig. 7 (in which the user is employed by `comp3`, but its contract `c2` is with `comp2`). These undesirable model instantiations are not ruled out by the invariant proposed in [15] in the `User` context.

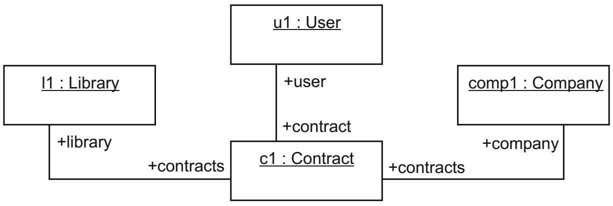

Fig. 6. The user has a contract with both the library and the company

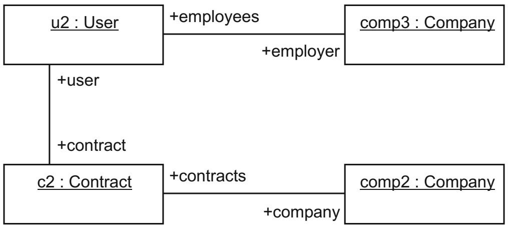

Fig. 7. The user is employed by `comp3`, but its contract `c2` is with `comp2`

Even more, in Figure 12 from [15], `contractB65` and `contractR43` have only one participant, `company80Y`, a stange situation in our oppinion. Also, in the same figure, if `userT6D` is unemployed by `company80Y`, and, by consequence, `contractQVR` is between `userT6D` and the library, we cannot understand why `company80Y` (which does not include among its employees `userT6D`) has a reference towards `contractQVR` between `userT6D` and the library.

Unfortunately, as stated before, our questions do not stop here. In Figure 10 from [15], a user may have many contracts, but in the requirements a different situation is mentioned. In the class diagram of Figure 10, all role names are implicit, fact that burdens the inteligibility of the model.

In this example, the snapshots meant to be used for testing have supported us in understanding that the requirements are incomplete and, by consequence, so are the model and the proposed invariant. In such cases, improving the requirements is mandatory.

7 Conclusions

The building of rigorous models, which are consistent with the problem requirements and have predictable behavior, relies on the use of constraints. Such constraints are not stand alone, they refer to the model in question. Consequently, the model's accuracy (in terms of the concepts used, their inter-relationships, as well as conformance to the problem requirements) is a mandatory precondition for the specification of correct and effective constraints. In turn, a full understanding of the model's semantics and usage requires a complete and unambiguous requirements specification. Requirements' validation is therefore mandatory for the specification of useful constraints.

The examples presented in this paper illustrate a number of bugs caused by failure to fulfill the above-mentioned requirements. Unfortunately, the literature contains many erroneous OCL specifications, including those concerning the UML static semantics, in all its available releases. Having free access to public resources offered via the web, students should know how to identify and correct errors such as those presented in this article. Our conclusion is that the common denominator for all the analyzed errors is *hastiness*: hastiness in specifying requirements, hastiness in designing the model (OCL specifications included), hastiness in building and interpreting snapshots (test data).

There are, undoubtedly, several ways of teaching OCL. The most popular (which we have referred as the "classic" one, due to its early use in teaching programming languages), focuses on introducing the language features. OCL being a complementary language, we deemed important to emphasize from the start the gain that can be achieved in terms of model accuracy by an inverted curriculum approach. In this context, we have insisted on the need of a complete and accurate requirements specification, on various possible design approaches for the same problem, on the necessity of testing all specifications by means of snapshots, as well as on the need to consider the effects of a particular OCL constraint shape on the execution of the code generated for the constraint in question.

However, the teaching and using of OCL involves a number of other very important issues that have been either not addressed or merely mentioned in this article, such as the specifications' intelligibility, their support for model testing and debugging, test data generation, language features, etc. This paper only focuses on the OCL introduction on new projects, with a closer look on training issues, this is why we have not detailed on the above mentioned topics.

Acknowledgements. This work was supported by CNCSIS - UEFISCSU, project number PNII – IDEI 2049/2008.

References

1. A UML-based Specification Environment, `http://www.db.informatik.uni-bremen.de/projects/USE`
2. Frame Based on the Extensive Use of Metamodeling for the Specification, Implementation and Validation of Languages and Applications (EMF_SIVLA) - Project Deliverables, `http://www.cs.ubbcluj.ro/ vladi/CUEM_SIVLA/deliverables/EduSymp2011/workspaces.zip`
3. InvestorWords, `http://www.investorwords.com/1079/contract.html`
4. The OCL portal, `http://st.inf.tu-dresden.de/ocl/index.php?option=com_content&view=category&id=5&Itemid=30`
5. Cabot, J.: Common UML errors (I): Infinite recursive associations (2011), `http://modeling-languages.com/common-uml-errors-i-infinite-recursive-associations/`
6. Chimiak-Opoka, J., Demuth, B.: Teaching OCL Standard Library: First Part of an OCL 2.x Course. ECEASST, vol. 34 (2010)
7. Chiorean, D., Ober, I., Petraşcu, V.: Avoiding OCL Specification Pitfalls. In: Proceedings of the 7th Educators Symposium: Software Modeling in Education at MODELS 2011 (EduSymp 2011). Electronic Communications of the EASST, vol. xx, 10 pages. EASST (2011)
8. Chiorean, D., Petraşcu, V., Ober, I.: Testing-Oriented Improvements of OCL Specification Patterns. In: Proceedings of the 2010 IEEE International Conference on Automation, Quality and Testing, Robotics - AQTR, vol. II, pp. 143–148. IEEE Computer Society (2010)
9. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: 2007 Future of Software Engineering, FOSE 2007, pp. 37–54. IEEE Computer Society, Washington, DC (2007), `http://dx.doi.org/10.1109/FOSE.2007.14`
10. LCI (Laboratorul de Cercetare în Informatică): Object Constraint Language Environment (OCLE), `http://lci.cs.ubbcluj.ro/ocle/`
11. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice Hall (1997)
12. Nierstrasz, O.: Synchronizing Models and Code (2011), Invited Talk at TOOLS 2011 Federated Conference, `http://toolseurope2011.lcc.uma.es/#speakers`
13. OMG (Object Management Group): Object Constraint Language (OCL), Version 2.3 Beta 2 (2011), `http://www.omg.org/spec/OCL/2.3/Beta2/PDF`
14. Software Technology Group at Technische Universität Dresden: Dresden OCL, `http://www.dresden-ocl.org/index.php/DresdenOCL`
15. Todorova, A.: Produce more accurate domain models by using OCL constraints (2011), `https://www.ibm.com/developerworks/rational/library/accurate-domain-models-using-ocl-constraints-rational-software-architect/`
16. Venners, B.: Abstraction and Efficiency. A Conversation with Bjarne Stroustrup - Part III (2004), `http://www.artima.com/intv/abstreffi2.html`