Transformation Contracts in Practice

Christiano Braga, Roberto Menezes, Thiago Comicio, Cassio Santos, and Edson Landim

Instituto de Computação, Universidade Federal Fluminense, Niterói, Brazil

{cbraga,rmenezes,tcomicio,cfernando,elandim}@ic.uf.br

Abstract

Model-driven development (MDD) is a software engineering discipline which suggests that software development should be done at the modeling level and that applications should be generated from models. The MDD approach is becoming more mature in time with new techniques and tools being defined to support it, also as a result of intensive research in this field. A key concept of MDD is a model transformation that generates applications from models. Since models are "first-class" citzens in MDD their verification and validation is an important task. Of course, also are model transformations. In this paper we apply the transformation contract approach to model transformations. A transformation contract is a specification of *what* a particular model transformation must implement and essentially specifies a relation between metamodels, the *transformation metamodel*, and properties that such a relation must satisfy, such as invariants in the Object Constraint Language (OCL). A transformation contract is said to be *correct* when a source model in conformance with its metamodel implies in a target model in conformance with the target metamodel and the joined model of source and target is in conformance with the transformation metamodel, where model conformance means that all the properties of a metamodel hold in a model. We have defined a *design pattern* that enforces transformation contract correction over model transformations, that is, a model transformation implemented following our transformation contract will be verified and validated following transformation contracts correctness. We have also developed model transformations using our transformation contract design pattern. The UMLToEJB model transformation generates application code in Enterprise Java Beans, from class diagrams described in the Unified Modeling Language. This paper reports on our proposed design pattern, the design of UMLToEJB, and illustrates how our approach may help different actors in MDD with transformation contracts, including model transformation designers, to identify erroneous situations.

1 Introduction

Model-driven development (MDD, e.g. [1]) is a software engineering discipline that considers models as *live artifacts* in the development process. By live artifacts we mean that models are not used for documentation purposes only but actually as input to software tools that may operate on them and produce other artifacts. Such artifacts

maybe compilable source-code or other models, in the same or different abstraction levels than the source model.

A key concept in MDD is a *metamodel* which defines the syntax for a modeling language. A model is said to be *syntactically well-formed* (or simply well-formed) with respect to a metamodel when it follows the structure defined by the given metamodel. In algebraic terms, a model is well-formed with respect to a given metamodel M when it is an instance of the signature induced by M. A stronger relationship between a model and a metamodel is the so-called *conformance* relation. A model m is said to be in conformance with a given metamodel M iff m is well-formed w.r.t. M and the *properties* of the metamodel M hold in m. An example property of the metamodel of class diagrams in the Unified Modeling Language (UML) [2] is that any given inheritance chain in a model may not have cycles. Hence, a class model m is only considered in conformance with the metamodel of UML class diagrams if m does not have cycles in its inheritance hierarchies. One way to understand such properties is to specify them as *invariants* over a metamodel.

Another key concept in MDD is a *model transformation*. Model transformations may be defined with different purposes such as code-generation or reverse engineering, understood as model generation from code. A model transformation is a function among many metamodels. In this paper, we consider a subclass of model transformations called unidirectional exogenous model-to-model transformations [3], that is, model transformations between two different metamodels, labeled source and target, that has to be applied from the source to the target. An unidirectional exogenous model-to-model transformation is essentially a function that relates elements from the two given metamodels.

Models are "first-class citizens" in MDD, therefore, their verification and validation is an important task. By verification we mean to make sure that a model has *general* properties such as consistency [4], which guarantees that properties of a model are satisfiable and therefore it may be instantiated, or temporal properties [5] related to behavioral aspects of a model. By validation we mean to check that a set of invariants [6] hold in a particular scenario or model instance, such as checking for cycles in a class hierarchy in a particular class diagram when it is understood as an instance of the UML metamodel. These verification and validation techniques may also be applied to model transformations when the specification of a model transformation is also understood as a *model*, called transformation model in [7] and joined metamodel in [8]. We continue the work in [8] by specifing a model transformation as a the *join* modeloperation on the source and target metamodels related by a model transformation. Verification and validation of a model transformation means to make sure that the join of the model instances of the source and target metamodels, related by the model transformation, is in conformance with the given metamodel. We call the *transformation contract* (e.g. [8] and [9]) of the model transformation the transformation metamodel and the properties that constrain it. Therefore, correctness of a model transformation is stated by $m \models I_M \Rightarrow ((m' \models I'_M) \wedge ((m \bowtie m') \models I_{M \bowtie M'}))$ where $q \in Q$ denotes that the model q is in conformance with the metamodel Q, $q \models I_Q$ denotes that the properties of Q hold in q, $Q \bowtie Q'$ denotes the join of metamodels Q and Q' and $q \bowtie q'$ denotes the join of the models q and q'.

The main contribution of this paper is a *design pattern* that enforces transfor-

mation contracts correctness over model transformations implemented following our approach.

Problem statement and paper contribution *Our research is applied on the problem of software verification. We apply a technique for the specification, verification and implementation of model transformations called transformation contracts. A transformation contract is a relation between the domains that a transformation relates together with properties that such a relation must fulfill, where a domain is defined as the metamodel of the modeling language associated with the domain together with properties that model instances of the given metamodel must fulfill. As a matter of fact, a model transformation is understood as a domain in the transformation contracts approach. The contribution of this paper is a realization of the transformation contracts approach, in the form of a design pattern, that enforces all the verification steps that transformation contracts-based model transformation must implement.*

This paper also reports on the model transformation UMLToEJB that generates CRUD (create, read, update and delete) code in Sun's Enterprise Java Beans from UML class diagrams. We describe its architecture, as an application of the transformation contract design pattern, and explain how transformation contracts may help to identify errors in a model transformation implementation. The tool[1] is available for download at `http://lse.ic.uff.br/`.

The remainder of this paper is organized as follows. In Section 2 we detail the MDD process used in this paper and the concept of transformation contracts. Section 3 discusses how *executable* invariants may be useful for different types of users. Section 4 reports on the transformation contract design pattern for model transformation implementations and our tool. Section 5 discusses related work. We conclude this paper in Section 6 with our final remarks.

2 MDD with transformation contracts

Model-driven development is an approach for software development where models of a system are *live* artifacts in the process. A model is not only a passive document that, for instance, represents a system's design, but it may actually be used as input to a software tool that *transforms* it into compilable source-code, in a so-called model-to-code transformation [3]. One such example is the model transformation from class diagrams in the UML into a database description written in the Structured Query Language (SQL).

Model transformations relate *languages*, such as UML and SQL. If one decides to work with the standards of the Object Management Group (OMG) and take advantage of the interoperability gained from using OMG's standards, the syntax of

[1] We have also implemented another model transformation following the ideas reported in this paper. SecureUMLtoAAC is a model transformation from access control policies, specified in SecureUML modeling language, to AspectJ code. Due to space constrains, this experiment is not described in this paper. The interested reader may download the tool and its documentation from `http://lse.ic.uff.br/`. A technical report is also available from `http://www.ic.uff.br/~cbraga/mda/tr.pdf`.

such languages may be described in the form of an UML class diagram.[2] Such a model is called a *metamodel* since it describes the syntax that models should follow. A model that is an instance of a metamodel is said well-formed with respect to the metamodel. The notion of well-formedness may be understood as the pertinence of a program with respect to the programming language it is written in, that is, a model must be well-formed w.r.t. its metamodel as a program written in a language L must be well-formed with respect to L's syntax. For example, UML has a metamodel and any UML class diagram may be seen as an instance of the UML metamodel that should be well-formed with respect to it. Figure 1 shows a simplified version of the UML metamodel, improved from [1]. The metamodel essentially represents the notions of datatypes, classes, attributes, operations, interfaces, association ends, their inter-relations and their typing relations.

UML is an extensible modeling language. Any given UML model element may be *tagged* or stereotyped, using UML terminology, in order to denote a new entity named after the tag's name. This is "UML's way" of defining *domain specific modeling languages*. We may now, once again, draw a relationship between model-driven concepts and programming languages concepts. One may understand a UML profile, which is essentially a UML extension consisting of a set of stereotypes and other extension elements, as the *concrete syntax* of a modeling language M. The metamodel of M may be understood as its *abstract syntax*. With that understanding in mind, the first step that a model transformation should do when transforming an UML model m written using an UML profile that represents a modeling language M is to map m into an instance of the metamodel of M. This step is similar to language *parsing* in a compiler when the abstract syntax tree of a program p is built according to the grammar of the programming language is written. Here we compare a program to a model and a metamodel to the grammar of a programming language. Even though stereotypes are not used the in the experiment we describe in this paper it is an important part of our model-driven development process and is used in the SecureUMLtoAAC tool we mentioned in Footnote 1 in Section 1. In our experiment described in Section 4.2 we do not use any profiles. The concrete syntax there is XMI [10], the OMG standard for model interoperability, and the parsing function creates an instance of the UML metamodel from a given class diagram.

Up to this point, the model-driven development process we are describing in this paper can be drawn as follows

$$\begin{array}{ccc} m \in Ms_{concrete\ \ syntax} & \xrightarrow{parse} & m' \in Ms_{Metamodel} \\ & \searrow^{\tau} & \\ n \in Mt_{Metamodel} & \xrightarrow{pretty\ \ print} & n' \in Mt_{concrete\ \ syntax} \end{array}$$

where m, m', n, n' are models; Ms and Mt represent the source and target metamodels of a model transformation τ; we write $m \in M$ to denote that the model m is well-formed with respect to M; $parse$ is the mapping that given a model m

[2] Of course, the Backus-Naur Form, or BNF for short, could also be used for such descriptions but the main point here is to interoperate and use all the efforts based on such standards.

written in the concrete syntax of a modeling language M generates an abstract syntax version m' of m, where m' is well-formed w.r.t. M's metamodel; $pretty\ print$ is the inverse mapping of $parse$; $M_{Metamodel}$ represents the abstract syntax for a modeling language M, represented as an UML class model; and $M_{concrete\ syntax}$ represents the concrete syntax, described in textual form (could be the concrete syntax of a programming language for instance), for a modeling language M.

It is not always true, however, that any well-formed model with respect to a given metamodel is *in conformance with it*. For instance, a UML class model m with an inheritance chain that has a cycle may be syntactically well-formed with respect to UML's metamodel but it is not in conformance with it. The reason is that there is an *invariant* in the UML metamodel that specifies that there should be no cycles in any inheritance chain. Since the invariant does not hold in m, the model m is not in conformance with UML's metamodel. The conformance relation between a model m and a metamodel M is given by well-formedness of m w.r.t. M and validity of the invariants of M in m, assuming M consistent, that is, assuming that M has instances.[3] The conformance relation between a metamodel and an instance of it is similar to the concept of *type checking* in programming languages. A syntactically correct program p w.r.t. a language L is ill-typed if the typing rules of L do not apply to p. It should be noted that invariants specified in OCL is one possibility for the specification of such properties. Other properties such as model consistency, which is essentially boolean satisfiability of a model with its constraints, or temporal formulae representing dynamic properties of a model, could also be specified and checked within our approach. However, they are out of the scope of this paper.

One way to specify such invariants is using the Object Constraint Language (OCL) [11]. Essentially, OCL has several constructs for manipulating collections of typed model elements in a model m, navigating through m's relationships, defining operations and invariants in M, where M is the metamodel of m. For example, the invariant *noCyclesinClassHierarchy* below checks for the presence of cycles in class hierarchies by verifying, for each class c, if c is not included in the transitive closure of the *inheritsFrom* relationship, in the UML metamodel, that represents the inheritance hierarchy. (This invariant is one of the invariants implemented in our tool in Section 4.2.) The invariant uses two operations, namely *superPlus* and *superPlusOnSet*, to calculate the transitive closure. The operation *superPlusOnSet* does the actual calculation by a recursive call on each element of the collection yielded by the *inheritsFrom* relation for each class c. Regarding OCL syntax, the keyword *context* defines the type of objects that the invariant should be applied to. The keyword *inv* defines an invariant. The informal meaning of the remaining OCL constructors in the example are as follows: *forAll* iterates over the elements of a given collection checking for a given predicate; *excludes* checks if a given collection does not contain a given element; *collects* creates a collection of objects such that a given predicate holds; *flatten* receives a set which may have other sets as elements and produces a flatten set of objects from its set elements; *asSet* casts a collection into a set; and *including* includes a given element in a given collection. The user-defined function

[3] In this paper, we assume every model consistent. We refer the interested reader to [4] for a discussion on this subject.

emptySet constructs an empty set of objects of type *Class*.

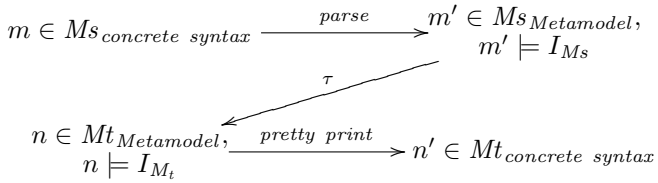
```
context Class inv noCyclesinClassHierarchy: self.inheritsFrom−>forAll(r|r.superPlus()
    −>excludes(self))

context Class::superPlus():Set(Class) body: self.superPlusOnSet(self.emptySet())

context Class::superPlusOnSet(rs:Set(Class)):Set(Class) body:
if self.inheritsFrom−>notEmpty() and rs−>excludes(self)
then self.inheritsFrom−>collect(c : Class | c.superPlusOnSet(rs−>including(self)))−>
    flatten()−>asSet()
else rs−>including(self) endif
```

OCL can be used to *automatically* validate UML models. Considering an implementation of an OCL interpreter, such as [12], one may actually apply the invariants of a metamodel M to a syntactically well-formed model m w.r.t. M to guarantee m's conformance with respect to M. Therefore, before applying a model transformation to a given model m, one must make sure that m is in conformance with its metamodel M (such as in Figure 1), that is, m is syntactically well-formed w.r.t. M and all invariants in M (such as *noCyclesinClassHierarchy*) hold in m.

The MDD process adopted in this paper when invariants are considered may be drawn as follows



where I_M are the invariants of the metamodel of the modeling language M and $m \models I_M$ means that all the invariants in I_M hold in the model $m \in M$.

We are now ready to explain the concept of transformation contracts and how it fits into the MDD approach considered in this paper.

A transformation contract is a specification of *what* a model transformation should do. It is written in the form of invariants that must hold in the *joined* metamodel of the source and target languages related by the given model transformation. An example invariant in a transformation contract for a model transformation $\tau : UML \rightarrow SQL$ is that for each class in $m \in UML$ there must exist a table t in $\tau(m) \in SQL$.[4] By joined metamodel we mean a metamodel M resulting from a model operation $M_1 \bowtie_A M_2$ on two given metamodels M_1 and M_2, that *extends* the metamodels M_1 and M_2 by: (i) exclusively uniting all the metamodel elements of M_1 and M_2; (ii) preserving the invariants of M_1 and M_2, that is, for all invariants $\iota \in I_{M_j}$, $j \in \{1, 2\}$, if ι holds in M_j implies that ι holds in M and no other invariant $\kappa \neq \iota$ holds in M_j as a result of $M_1 \bowtie M_2$[5]; and (iii) declaring associations $a \in A$, such that A is disjoint

[4] A more precise model transformation would consider a stereotype marking a given class in m as persistent. However, for pedagogical purposes, let us consider a simpler model transformation without such markings, hence assuming all classes persistent.

[5] In algebraic terms [13], M *extends* the metamodels M_1 and M_2 and adds "no confusion", that is, preserves the included model.

with the associations of M_1 and M_2, and each a may relate classes in M_1 and M_2. A transformation contract is the metamodel $M = M_1 \bowtie M_2$ called transformation metamodel and the set of invariants I_M.

The MDD process adopted in this paper when transformation contracts are considered may be drawn as follows where $M = Ms \bowtie_A Mt$, $k \in M$, and $k = (m \bowtie_L n)$, where L is a set of links l instances of the associations in A.

We continue the paper in Section 3 showing the usefulness of the invariants mentioned in this section for validation purposes and for different types of users of our approach when they are executed.

3 Invariants as live assertions

A transformation contract may be executed for each application of a model transformation that implements it in the same way it is done for the invariants of a metalanguage (such as noCyclesinClassHierarchy). Hence, model transformations may be validated using transformation contracts as *live assertions* when an OCL interpreter is used in the implementation of a model transformation. This approach is similar to Meyer's Design by Contract [14] (DbC). In DbC, programs are annotated with assertions and raise exceptions when they fail. Here, a model transformation terminates abruptly when an OCL assertion fails. Therefore, transformation contracts have the same benefits for the transformation developer as DbC provides for a programmer that adopts it.

In this work we consider a model-driven development process as depicted in Figure 3 and identify three types of actors that may be benefited by a model transformation implemented using our transformation contract-based approach: (i) designers of modeling languages, (ii) developers of a model transformation and (iii) users of a model transformation. During the development process, an actor may identify problems in an artifact, thanks to the execution of invariants defined by him or her or other actors of the process.

The validation process can be divided in two steps: (i) an actor identifies a problem on his or hers own artifact, such as a modeling language designer checking the invariants that he or she designed or (ii) an actor identifies a problem on an artifact designed by a previous actor in the development process, such as a model transformation designer reporting a problem in a modeling language while testing the model transformation he or she is developing. Moreover, each actor may encounter the following situations: (i) a model is ill-formed with respect to its metamodel, (ii) a model is well-formed with respect o its metamodel and is in conformance with the properties of its metamodel, and (iii) a model is well-formed and is *not* in conformance with the properties of its metamodel.

Traceability is the ability to relate model elements in different models that are potentially in different abstraction levels. This ability plays an important role in the implementation of model transformations since it helps the identification of the different errors listed in the previous paragraph. Our approach provides a form of traceability, which we call *invariant-based traceability*, that identifies model elements that make a property *fail* during the verification or validation of a given model. For

the case when the properties of a metamodel are described in OCL, which is the focus of this paper, when an invariant fails in a model instance of the given metamodel, we select the objects of the type of the context of the failing invariant that do *not* have the property specified by the failing invariant. This allows the actor playing with the failing model to either submit a report to another actor with the failing scenario or to query the model about the failing objects, in a form of model debugging. It should be noted that other forms of traceability are required for different formalizations of the properties of a metamodel, such as model consistency in Description Logic or behavior properties as temporal formulae mentioned in the introduction of this paper. It is, however, out of the scope of this paper to discuss them.

In Section 4.2.6 we illustrate, in the context of model-driven development of distributed systems using Enteprise Java Beans, how our approach may help identify the different situations described in this section using traceability.

We continue this paper in Section 4, discussing how we have used the concepts in this section to implement a general model transformation design pattern that implements the structure that implements the structure and behavior of transformation contracts and a model transformer from a subset of UML class models to Oracle/Sun's Enterprise Java Beans.

4 Transformation contracts in practice

The objective of this section is twofold. In Section 4.1 we introduce a design pattern for the rigorous development of model transformations following the transformation contracts approach. In Section 4.2 we apply in the development of a model transformation from UML class diagrams to Java code with Enterprise Java Beans support, following the model transformation specified in [1].

4.1 A design pattern for model transformations

A transformation contract is a specification of *what* a model transformation should do. It is written in the form of invariants that must hold in the *joined* metamodel of the source and target languages related by the given model transformation.

From our presentation of model transformations with transformation contracts in Section 2 one identifies the following basic concepts: metamodel, metamodel invariants, transformation metamodel, transformation metamodel invariants, model parsing and model pretty printing.

We propose a design pattern to enforce the transformation process described by Figure 2 in Section 2. Its main components and their relationships are depicted in Figure 4.

The meaning of each class in the abstract architecture is as follows. The class *ModelManager* provides an interface for declaring class diagrams, object diagrams, invariants, and querying object diagrams. It declares several insertion methods for class diagrams model elements (such as *insertClass*) and object diagram model elements (such as *insertObject*) along with insertion methods for invariants and operations on class diagrams.

The class *Domain* represents a modeling language and allows for the instantiation and manipulation of models which are instances of the domain's metamodel. It declares methods for: (i) the creation of the modeling language's metamodel on the *ModelManager* instance associated with a domain (method *createMetamodel*), (ii) the creation of a model in the modeling language's concrete syntax as an instance of the modeling language's metamodel on the *ModelManager* instance associated with the domain (method *parse*), (iii) generation of the concrete syntax of a model represented as an instance of the domain's metamodel, stored in the the *ModelManager* instance associated with the domain (method *pretty-print*), and (iv) validation of a model as an instance of the domain's metamodel, stored in the *ModelManager* instance associated with the domain (method *validate*).

The class *JoinedDomain* represents the joined metamodel of a model transformation, that is, the disjoint union of source and target metamodels together with model elements (classes or associations) that relate them. It inherits from class *Domain* all the behavior that a domain should have and is associated with an instance of class *TransformationContract* that manages the transformation process between the domains related by *JoinedDomain*.

The class *TransformationContract* essentially defines a method that implements the general behavior of a transformation contract which, given a well-formed model, instance of the source metamodel of the *JoinedDomain* associated with the transformation contract, (i) validates the given model according to the source metamodel's validators, (ii) transforms it into a model instance of the target metamodel, (iii) validates the generated model using the target metamodel's validators, and, finally, (iv) generates the target model in the target metamodel's concrete syntax. It also defines a constructor responsible for instantiating the source domain and requesting it to parse a given model, and a method that requests for an instance of the source domain to parse a given model (method *parse*).

The interface *IValidator* provides a method for requesting a reasoner to validate a model instance of the validator's domain (method *validate*). Finally, the interface *IXMIParser* defines a method that produces a model instance of a domain's metamodel given a file in OMG's XML Metadata Interchange (XMI) standard (method *parse*).

This design pattern not only enforces the transformation process of transformation contracts but also allows for an easy addition of new validators for domains, such as consistency checking, as described in [4]. We will return to this point in Section 5.

Section 4.2 describes UMLtoEJB, a model transformation from UML class diagrams to Java code with Enterprise Java Beans support. It is also a model-to-model transformation, from a simplified UML metamodel to a simplified EJB metamodel, that pritty-prints Java code when proper models are generated according to UMLtoEJB transformation contract.

4.2 A transformation contract approach to distributed systems development

The purpose of the paper is to describe a design pattern that implements the transformation contracts approach. To illustrate an application of the design pattern, we have

chosen to implement the model transformation from UML to Enterprise Java Beans described in [1] as it is one of the main references in the MDA literature. Even though they specify, in a language very close to OMG's Query View Transformations (QVT), and very clearly, a transformation from UML class diagrams to (coarse-grained) EJB components, the given specification is not executable. With teaching purposes in mind, but also aiming at having a full-fledged model transformer available for us to experiment with the transformation contract approach, we have implemented a model transformation from UML class diagrams to Java code with EJB support based on [1] using the concept of transformation contracts described in Section 2. We call the resulting model transformation UMLtoEJB. In UMLtoEJB we extend and improve the metamodels and model transformation described in [1] by allowing class inheritance in the source model, specifying and implementing a transformation contract for the model transformation. This is the reason why we have chosen to use the UML metamodel described in Figure 1 (which is a subset of version 1.4 of the UML standard) and not another metamodel such as Eclipse's Ecore. To our purposes, the metamodel in Figure 1 is as good as Ecore or any other. Moreover, as mentioned above, we have teaching purposes in mind and we present in the class room both approaches. The results of this discussion, from a pedagogical point of view is out of scope of this paper.

This section is organized as follows. In Section 4.2.1 we describe the metamodels related by the model transformation UMLtoEJB. Section 4.2.2 describes the invariants of each metamodel related by UMLtoEJB. Section 4.2.3 describes the transformation contract. Section 4.2.4 describes the model transformation implementation as an application of the design pattern in Figure 4. Section 4.2.5 describes an example of the application of UMLtoEJB. Finally, Section 4.2.6 exemplifies how the concept of transformation contract helped improving the confidence on the correctness of UMLtoEJB.

4.2.1 Source and target metamodels

UML metamodel We use a simplified UML metamodel depicted in Figure 1. Every element in a diagram is an instance of a *ModelElement*, that must have a name. Essentially, every element in a model is a *Classifier*, *Typed* or *Association*. The abstract metaclass *Classifier* is a generalization of *Class*, *Interface* and *DataType*. *Typed*, as *Classifier*, is an abstract metaclass and a generalization of *Feature* and *Parameter*. Each *Typed* has its type defined by a *Classifier*. *Association* defines how classes are associated through instances of *AssociationEnd*. A *Feature* can be an *AssociationEnd*, *Attribute* or *Operation*. An *AssociationEnd* represents one side of an association between classes. The metaclasses *Attribute* and *Operation* represent, respectively, an attribute and a method of a given class. The metaclass *Operation* may have *Parameters*, that, of course, represent the parameters of a given method. *DataType* represents the data types that can be used in the models. *Interface*, as the name suggests, represents an interface as in the Java language. The metaclass *AssociationClass* is both a *Class* and an *Association*. The metaclass *Class* can implement *Interface* and can inherit from another *Class*.

Enterprise Java Beans EJB is a server-side component architecture intended to enable scalable business applications developed by Sun/Oracle. It supports three component types: entity bean, responsible for object persistence; session bean, responsible for the implementation of an application's business rules; and message-driven bean, responsible for the implementation of asynchronous concurrent behavior in an EJB-based application.

The EJB metamodel is depicted in Figure 5. *EJBDataAssociation* and *EJBDataClass* represent, respectively, associations and classes of the *internal structure of a component*. An *EJBDataSchema* represents a set of instances of *EJBDataAssociation* and *EJBDataClass*. An *EJBServingAttribute* represents an association end between an entity bean and a data class. An *EJBKeyClass* is an identifier to an *EJBEntityComponent* used to avoid sending an entire object through the network. *EJBEntityComponent* and *EJBSessionComponent* represent, respectively, entity beans and session beans of an application. *EJBEntityComponent* persistence is represented by its association with the metaclass *Table*.

The EJB metamodel is almost the same one presented in [1, pg. 115], except for: (i) the insertion of the metaclass *EJBModelElement* that every other metaclass inherits, to simplify the definition of characteristics common to all metaclasses and (ii) the insertion of the metaclass *EJBSet* to represent sets in EJB, as in the UML metamodel.

4.2.2 Source and target metamodel invariants

For both metamodels, a common type of invariant is to guarantee the multiplicity of each association. As an example, the OCL invariant *restrictionMinimunOneAssociationEndPerAssociation* details, for the UML metamodel, that every *Association* requires at least one *AssociationEnd*.

---

context Association **inv** restrictionMinimunOneAssociationEndPerAssociation: self.end
−>size() $\geq$ 1

An important invariant of UML metamodel is *noCyclesinClassHierarchy*, described in Section 2, guarantees the absence of cycles in class hierarchies.

The invariant *restrictionRequiredFieldNameToModelElement* specifies that all UML model elements, except for associations and association ends, must be named.

---

context ModelElement **inv** restrictionRequiredFieldNameToModelElement:
self.name <> '' or self.oclIsKindOf(Association) or self.oclIsKindOf(AssociationEnd)

The association ends of an association class may not be compositions, specified by *UMLAssociationmusthaveoneAssociationEndcompositiontrue*, and no association end may be associated with an association class, specified by *AssociationClassfromUMLAssociationClassToEJBDataClasscannothaveAssociationEndwithcomposition*.

---

context Association **inv** UMLAssociationmusthaveoneAssociationEndcompositiontrue:
self.ejbDataAssociation−>notEmpty() implies self.oclIsTypeOf(Association) and self.end
−>exists(end : AssociationEnd | end.composition = true)

```
context AssociationClass inv
    AssociationClassfromUMLAssociationClassToEJBDataClasscannothaveAssociationEndwithcomposition
    :
self.ejbDataClass->notEmpty() implies not self.feature->exists(end : AssociationEnd |
    end.otherEnd->exists(other : AssociationEnd | other.composition = true))
```

Finally, only binary associations may be composite. This is specified by invariant *limitCompositeAssociationSize*.

```
context Association inv limitCompositeAssociationSize: self.end->exists(end :
    AssociationEnd | end.composition = true) implies self.end->size() = 2
```

Entity beans contain four groups of methods: (i) create methods, responsible for the creation of a new instance, (ii) finder methods, responsible for locating an entity bean, (iii) remove methods, responsible for removing an entity bean, and finally (iv) home methods, which implement the component's business logic.

Methods in the groups (i)–(iii) must have "ejb" as a prefix, as defined in the EJB standard. A designer of an EJB model does not need to care about these methods as they are automatically handled by the EJB domain pretty-printer. Home methods, on the other hand, represented in the EJB metamodel by the metaclass *BusinessMethod*, must be specified by the EJB model designer and may *not* have such a prefix. The invariant *prefixEjbisforbidden* specifies this constraint over *BusinessMethods*.

```
context BusinessMethod inv prefixEjbisforbidden: self.name.size() >= 3 implies self.
    name.substring(1,3) <> 'ejb'
```

4.2.3 Transformation contract for UMLtoEJB

Following the transformation contract design pattern described in the introduction of this section, the transformation from an UML class diagram m to Java source code c with EJB support has four steps: (i) all UML invariants must hold in m; (ii) the model m is transformed into an instance of the EJB metamodel e; (iii) all invariants of the EJB metamodel and the transformation metamodel of UMLtoEJB must hold in e; (iv) the Java source code with EJB support is generated from e.

Figure 6 shows the UMLtoEJB transformation metamodel. On the left-hand side it represents the metaclasses of the UML metamodel and on the right-hand side the metaclasses of the EJB metamodel. In the center part one may find the associations among the source and target metaclasses that specify which EJB elements the model transformation must generate from elements in a UML class diagram.

Due to space constraints, in this section we focus only on two associations: (i) *ejbKeyClass*, between the metaclass *Class*, from the UML metamodel, and the metaclass *EJBKeyClass*, from the EJB metamodel, and (ii) *ejbId* relating a class from the UML metamodel and an *EJBAttribute* from the EJB metamodel. The associations *ejbKeyClass* and *ejbId* specify that, given a UML *Class*, the model transformation must generate an *EJBKeyClass* and an *EJBAttribute*. Moreover, the generated *EJBAttribute* is an attribute of the generated *EJBKeyClass*.

The invariants *keyClassnameisValid* and *idAttributesmustbeInteger* specify that an *EJBKeyClass* must be named after the UML class that it is originated from and that

the type of the generated *EJBAttribute* must be of type *Integer*, our chosen type to represent identifiers. The invariants *validatetransfomedKeyClasselementsfromClasses* and *validateIfEveryClasswastransformedtoEJBKeyClass* guarantee that the generated *EJBAttribute* is a feature of the generated *EJBKeyClass* (see Figure 5) and that an *EJBKeyClass* and an *EJBAttribute* must be generated from an UML *Class*.

The complete set of invariant is listed in Appendix A.

```
context Class inv keyClassnameisValid: self.ejbKeyClass->notEmpty() implies self.name
    = self.ejbKeyClass.name
context Class inv idAttributesmustbeInteger: self.ejbId->notEmpty() implies self.ejbId.
    type.name = 'Integer'
context Class inv validatetransfomedKeyClasselementsfromClasses: self.ejbKeyClass->
    notEmpty() and self.ejbId->notEmpty() implies self.ejbId.classifier = self.
    ejbKeyClass
context Class inv validateIfEveryClasswastransformedtoEJBKeyClass: self.ejbKeyClass
    ->notEmpty() and self.ejbId->notEmpty()
```

4.2.4 Architecture

UMLtoEJB model transformations is implemented as an extension of the abstract architecture presented in the introduction of this section. Figure 7 shows the extension.

Domain, *JoinedDomain* and *IValidator*, the classes and interface in grey, come from the abstract architecture. *UMLDomain* and *EJBDomain* represent, respectively, the UML and EJB metamodels, both based on the metamodels presented in [1]. *UMLEJBDomain* extends *JoinedDomain*, so it uses *UMLDomain* and *EJBDomain* as source and target domain. It implements the rules to transform an UML class diagram, loaded by *UMLDomain* in the *ModelManager*, to an EJB metamodel's instance. *EJBDomain* pretty-prints the result of transformation in Java syntax.

In order to validate our domains, we have extended *IValidator* to *InvariantValidator*, that represents invariants written in OCL to validate a given metamodel's instance. *UMLInvariantValidator*, *EJBInvariantValidator* and *UMLEJBInvariantValidator* extend *InvariantValidator* and, respectively, validates the UML, EJB and the UMLtoEB joined domain using their respective invariants.

Our implementation uses the transformation rules and metamodels specified in [1], extended in the following ways: (i) the execution of the transformation rules is split in two phases: first the metaclasses are generated and then the associations between them to avoid errors for linking elements that were not transformed yet; (ii) navigable associations are treated as plain associations as they are not explicitly handled in EOS, our OCL interpreter of choice; (iii) our tool processes models described in XMI produced by ArgoUML, which handles composition associations differently from the book (the XMI file inform the composition in the correspondent association end while [1] considers this information in the opposite association end). This implied a change in the way classes are transformed into *EJBEntityComponents*, which affects the transformation.

4.2.5 Example

Figure 8 shows a meeting manager system modeled as an UML class diagram. Essentially, it consists of a system that users can schedule meetings and notify its participants by messages. In this system an entity called *Person* is responsible for representing a real person and *User* represents the information necessary to allow a person to log in the system. *Role* represents which roles an *User* can assume in the system, such as *root*. *Meeting* represents a real meeting that will occur in a given date and will have participants, that can be notified by a *Message*.

The application of the model transformation UMLtoEJB to the meeting model generates classes that represent *EJBObject*, *EJBHome* and *EntityBean*, all the infrastructure needed to deploy an EJB application. *EJBObject* defines the business methods callable by a client. *EJBHome* defines methods that allow a client to create, remove and find EJB objects. The class *EntityBean* defines how the container EJB treats an instance's life cycle events.

Figure 9 shows part of the transformation result: everything directly related with the entity *Meeting* from Figure 8. The class *Meeting* and *MeetingHome* represent, respectively, an *EJBObject* and *EJBHome* interfaces to represent the meeting's entity bean, called *MeetingBean*. The *EJBDataClass*, that represents the persistent data, for the class *Meeting* from the Figure 8 is called *MeetingDataObject*. It has a primary key called *MeetingKey* and have associations with *PersonKey* to represent the meeting owner and participants. These objects represent the basic structure needed for an entity bean to be instantiated correctly, when using the coarse-grained components approach of [1].

The result of the transformation, this tool and its documentation is available for download at `http://lse.ic.uff.br/` including a deployed application server loaded with this project.

4.2.6 Transformation contracts as live assertions

Recall from Section 3 that we defined three different "actors" in our proposed MDD process with transformation contracts and three situations that the validation process may identify regarding well-formedness and conformance of a model with respect to its metamodel. The actors are the modeling language designer, the model transformation developer and a model transformation user. The validation situations are a ill-formed model, a well-formed but invalid model and, finally, a well-formed and valid model. We also defined *invariant-based traceability*, a form of traceability supported in our approach to aid model debugging, which lists the model elements that fail in an invariant check thus allowing the generation of an error report or query to the model about the failing objects.

Invariant-based traceability is automatically available to any model transformation that uses our implementation of the design pattern described in Section 4.1. When validation of a domain fails, method `validate` raises an exception with information identifying the failing invariant and the objects that did not pass the invariant check.

In this section we illustrate how our approach may help identify erroneous situations given the structure we described in Section 3. Besides the implementation

of invariant-based traceabilty we also developed an application, that we call UMLtoEJBPad, to help on the visualization of the erroneous scenario. The tool is an instantiation of a small object-oriented framework for model visualization and query based on transformation contracts. It was devised to be a simple tool for teaching metamodeling and model transformations. A complete description of the architecture of this tool is out of scope of this paper. For the purposes of this section, it suffices to explain that: (i) it allows visualization of a model as an instance of its metamodel as a graph, (ii) it supports OCL query over the models in a given application of the model transformation, and (iii) it provides a simple graphic-user interface for traceability.

We give two examples of erroneous scenarios: when a model is ill-formed and when it is well-formed but invalid. For each example we: (i) describe the erroneous scenario, (ii) explain the invariant that fails, (iii) explain the output of UMLtoEJBPad, and (iv) explain the traceability information.

The first example is for an *ill-formed* metamodel instance resulting from the parsing of the UML class diagram in Figure 8. This error situation is a result of the UML *language designer actor* not implementing a proper model parser for UML and resulting models do not have a cardinality between *Association* and *AssociationEnd* greater or equal to 1. The OCL invariant that fails is `restrictionMinimunOneAssociationEndPerAssociation` and the output message of UMLtoEJBPad and the log file are shown in Figure 10.

With the information provided by the traceability infrastructure, one may query the failing model using UMLtoEJBPad (see Figure 11) to identify the problematic model element and see its relations with other model elements. In this example is easy to see given the error message, the failing invariant and the visual output that there is a problem while building the relations between *Association* and *AssociationEnd* for objects representing classes *User* and *Role*, which is a responsibility of the UML domain parser.

The next example is an error situation given by a well-formed but invalid model. Assume that a *model transformation user actor* defined a cycle in the inheritance hierarchy among classes *User*, *Role* and *Person* in Figure 8. The invariant `noCyclesinClassHierarchy` would fail and UMLtoEJBPad would show the error message in Figure 12, which also shows the error log entry.

Once again, the actor may query the failing model, as shown in Figure 13, for the problematic objects. Given the error message, the failing invariant and the visual output shown by `UMLtoEJBPad` the actor may conclude that there is a problem while building the inheritance relations among the objects representing the classes *User*, *Role* and *Person*.

5 Related work

The main contribution of this paper is a design pattern for the rigorous implementation of model transformations following the transformation contracts approach. The Query-view-transformations language [15], an OMG standard, also induces a pattern: a model transformation may only be applied if the preconditions stated in the *when* clause of the given model transformation hold and also the postconditions stated in

the *where* clause of the given model transformation must hold after the application of the given transformation. When we compare QVT specifications to our approach there are a few aspects that must be taken into account:

- QVT is a *language* while transformation contracts is an *approach* that may be implemented in *any* language, such as QVT or even Java, as long as the properties of all domains hold.

- The transformation contract approach suggests an abstract way to specify a model transformation as a relation between domains. A QVT description of a model transformation is *one* possible realization of such a specification. A QVT description which is not based on a transformation contract would rely on the model transformation designer to remember to write all the properties of the source and target domain as well as the model transformation properties. The design pattern proposed in Section 4.1 enforces that not only pre and postconditions of a model transformation must hold but also the domains properties.

- QVT only allows for the specification of OCL properties in its *when* and *where* clauses while the transformation contract approach abstracts from that and enforces that all domain properties should hold, which may be described in any suitable formalism not only OCL. Different properties may be best described and verified in different formalisms.

- Domains are a key aspect in transformation contracts and not explicitly supported on QVT. For instance, the property that specifies the absence of cycles in class hierarchies does not belong to a transformation but rather to the UML domain. Without this property a EJB dataclass with cycles in its hierarchy could be generated and would not be the model transformation designer fault!

- Domains also help on the design of a model transformation as the latter should be *syntax driven*: every model element in a domain should be understood as a language construct with its own semantics. As an example, in [1] the composition association is not handled properly as no data integrity enforcement is produced in the SQL representation of a UML class diagram. If the model transformation was described in a syntax driven way perhaps the transformation designer would have defined it properly.

The concept of transformation model is central to our work. In [7] the idea of transformation model instead of model transformation is discussed. The authors describe the benefits of omitting details of the transformation process and concentrating in depicting the transformation as a model, in which models are instances of metamodels and transformations can be described in conformity to a metamodel. From this point it is possible to describe a transformation model in a formal way, thus it can be validated and verified.

Using invariants to specify transformation models is another key aspect of the work discussed in this paper. (We would like to stress that invariants, and in particular OCL-specified ones, are one class of properties that may be described.) The

correctness of a transformation between models has been discussed by many authors. In [16] the author emphasizes that model transformation specification can be considered a special kind of model and as such it can be subject to existing model analysis techniques. According to [17], the correctness of a model becomes a major issue as the model defects will directly become implementation defects in the final software system due to the application of code-generation techniques. However, popular modeling notations are not formal enough to prove the correctness of models. Therefore, a set of model-level verification techniques are needed to ensure the quality of software model specifications.

Some approaches transform models into other languages and formalisms, in which the validation occurs by the use of solvers or theorem provers. In [18], Constraint Satisfaction Problem (CSP) are used to verify UML/OCL models using a translation method. After the translation, a constraint solver is used to verify if the problem is satisfiable under a finite search space of possibilities. The advantage of this method is the assurance that the model and its invariants are well defined. Our work could be complemented by this kind of validation to ensure the consistency of metamodels and transformation contracts, where this would no longer be a responsibility of the designer, e.g., combining the presenting work with the approach described in [4].

Another key point in our approach is the concept of transformation contract. In [19], transformation contracts are presented as one of the test case generation techniques used in the validation of transformations. To check the transformation consistency it describes the creation of a new metamodel adding an element to connect elements from the source and target metamodels. Their approach consists on generating test cases. Our approach also aims at gaining confidence of model transformations but instead we prefer *to view the specification of a model transformation also as a metamodel* and apply the same techniques one would use to validate a model also to validate a model transformation.

In [9] the contracts are verified in three steps: in the source model, in the target model and in the transformation model. For the transformation, it is presented two possible methods to verify the consistency using transformation contracts. In the first one, the elements from the source model are verified in the precondition and the transformation is verified in the postcondition linking elements from the source model with the **@pre** constructor from OCL. This constructor enables the elements from the target model to be compared with the elements from the source model. In this method, the transformation operation is described in OCL and both source and target metamodels are the same or close enough. The second method consists in the creation of two packages. The first contains elements from the source model and the second contains elements from the target model. The transformation operation is also defined in OCL, having a package for the source model as the input and resulting in a package of the target model. In this case, both elements from the source and target model can be referenced in the transformation operation. Instead of the notion of using constraints to validate the model, the methods defined in [9] are too dependent of the UML metamodel.

A difference between our approach and the methods above is the use of OCL for the transformation operation, which is not fixed in our approach, even though we made extensive use of it in this paper. Our approach does not define contracts for

test cases, instead it is defined as part of transformation specification. It is positive to consider OCL as the standard for the specification of the constraints and also the transformation. Nevertheless, it limits the transformation, e.g., in the first method the source and the target metamodels needs to be the same. In the second method it is necessary to create a new OCL operator to map elements from the source and target models. In our approach, the designer must specify the joined metamodel and the association between the source and target elements in an explicit way representing the equivalence between the structure of languages represented by the metamodels. However, the transformation operation is not specified in OCL. What we propose is that the designer specifies the properties that must be fulfilled and the actual transformation may implement the transformation in any way as long as the specified properties are checked. Executing OCL invariants is one such possibility. Other properties such as consistency, as discussed in [4], or temporal formulae could also be checked in our approach.

Finally, our process is different and lies on the premise that models are in conformity to metamodels, which represent languages. The transformation of models can be seen as a transformation of languages. We also present the transformation contracts as invariants of the process and not as pre and postconditions of the transformation.

6 Final remarks

We have presented a design pattern to support a rigorous approach to model transformation development with transformation contracts. A transformation contract is a set of invariants that specify a relationship between a source and a target metamodel. A transformation contract is an executable specification when described in OCL and the model transformation implementation interacts with model validators such as an OCL interpreter. With such an architecture, a transformation contract becomes a conjunction of live assertions that may be checked every time a model transformation is executed. As an illustrative example, we discussed the UMLToEJB model transformation that generates Java code with EJB support from UML class diagrams. It implements the concept of transformation contracts by applying our design pattern. We have also exemplified the benefits of using transformation contracts to different types of users of our approach and how it may improve the confidence on the correctness of the implementation of the model transformation.

References

[1] Kleppe A, Warmer J, Bast W. MDA Explained - The model driven architecture: practice and promise. Addison Welsey; 2003.

[2] Booch G, Rumbaugh J, Jacobson I. UML - User's Guide. Addison Wesley; 2005.

[3] Czarnecki K, Helsen S. Classification of Model Transformation Approaches. In: OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture; 2003. p. 1–17.

[4] Braga C, Hæusler EH. Lightweight analysis of access control models with description logic. Innovations in Systems and Software Engineering. 2010;6:115–123.

[5] Jr EMC, Grumberg O, Peled DA. Model Checking. The MIT Press; 1999.

[6] Warmer J, Kleppe A. The Object Constraint Language. 2nd ed. Addison-Wesley Longman Publishing Co., Inc.; 2003.

[7] Bézivin J, Büttner F, Gogolla M, Jouault F, Kurtev I, Lindow A. Model Transformations? Transformation Models! In: Proceedings of the 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6. vol. 4199. Springer; 2006. p. 440–453.

[8] Braga C. A Transformation Contract to Generate Aspects from Access Control Policies. Journal of Software and Systems Modeling. Springer, 2010;DOI: 10.1007/s10270-010-0156-x.

[9] Cariou E, Marvie R, Seinturier L, Duchien L. OCL for the Specification of Model Transformation Contracts. In: Patrascoiu O, editor. OCL and Model Driven Engineering, UML 2004 Conference Workshop, October 12, 2004, Lisbon, Portugal; 2004. p. 69–83.

[10] XML Metadata Interchange. Object Management Group; 2007.

[11] OMG. Object Constraint Language Specification, v2.0 [type]; 2006 [cited 04/27/2010]. Available from: http://www.omg.org/spec/OCL/2.0/.

[12] Clavel M, Egea M, de Dios MAG. Building an Efficient Component for OCL Evaluation. ECEASST. 2008;15.

[13] Goguen JA, Malcom G. Algebraic Semantics of Imperative Programs. MIT Press; 1996.

[14] Meyer B. Object-Oriented software construction. 2nd ed. Prentice Hall; 1997.

[15] MOF QVT Final Adopted Specification, OMG Adopted Specification ptc/05-11-01. Object Management Group; 2005.

[16] Anastasakis K, Bordbar B, Küster J. Analysis of Model Transformations via Alloy. In: Baudry AB, Faivre S, Ghosh AP, editors. Proceedings of the workshop on Model-Driven Engineering, Verification and Validation (MoDeVVA 2007), Nashville, TN (USA). Springer; 2007. p. 47–56.

[17] Cabot J, Clarisó R, Riera D. Verifying UML/OCL Operation Contracts. In: Leuschel M, Wehrheim H, editors. IFM. vol. 5423 of Lecture Notes in Computer Science. Springer-Verlag; 2009. p. 40–55.

[18] Cabot J, Clarisó R, Riera D. Verification of UML/OCL Class Diagrams using Constraint Programming. In: ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop. IEEE Computer Society; 2008. p. 73–80.

[19] Baudry B, Dinh-Trong T, Mottu JM, Simmonds D, France R, Ghosh S, et al. Model transformation testing challenges. ECMDA workshop on Integration of Model Driven Development and Model Driven Testing. 2006 July;.

A Invariants for UMLtoEJB transformation contract

---

context Class **inv** keyClassnameisValid:
self.ejbKeyClass−>notEmpty() implies self.name = self.ejbKeyClass.name

context Class **inv** idAttributesmustbeInteger:
self.ejbId−>notEmpty() implies self.ejbId.type.name = 'Integer'

context Class **inv** validatetransfomedKeyClasselementsfromClasses:
self.ejbKeyClass−>notEmpty() and self.ejbId−>notEmpty() implies self.ejbId.classifier
= self.ejbKeyClass

context AssociationClass **inv**
primaryKeymustbelongstoaEJBKeyClassfromUMLAssociationClass:
self.ejbKeyClass−>notEmpty() implies self.name = self.ejbKeyClass.name

context AssociationClass **inv** primaryKeymustbeEJBIntegerfromUMLAssociationClass:
self.ejbId−>notEmpty() implies self.ejbId.type.name = 'Integer'

context AssociationClass **inv** validatetransfomedKeyClasselementsfromAssocClasses:
self.ejbKeyClass−>notEmpty() and self.ejbId−>notEmpty() implies self.ejbId.classifier
= self.ejbKeyClass

context Class **inv** UMLClassfromUMLClassToEJBDataClasscannotbeanOutermostclass
:
self.ejbKeyClass2−>notEmpty() implies not self.isOuterMostContainer()

context Association **inv** UMLAssociationmusthaveoneAssociationEndcompositiontrue:
self.ejbDataAssociation−>notEmpty() implies self.oclIsTypeOf(Association) and self.
end−>exists(end : AssociationEnd | end.composition = true)

context AssociationClass **inv**
AssociationClassfromUMLAssociationClassToEJBDataClasscannothaveAssociationEndwithcomposition
:
self.ejbDataClass−>notEmpty() implies not self.feature−>exists(end : AssociationEnd
| end.otherEnd−>exists(other : AssociationEnd | other.composition = true))

context AssociationEnd **inv**
AssociationfromUMLAssociationEndToEJBDataEndusingRule8mustbetypedasAssociation
:
self.ejbEnd_r1−>notEmpty() implies self.association.oclIsTypeOf(Association)

context AssociationEnd **inv**
AssociationrelationtypefromUMLAssociationEndToEJBDataEndusingRule8:

self.ejbEnd_r1->notEmpty() implies self.classifier.getOuterMostContainer() = self.
type.getOuterMostContainer()

context AssociationEnd **inv**
AssociationfromUMLAssociationEndToEJBDataEndusingRule9mustbetypedasAssociation
:
self.ejbEnd_r2->notEmpty() implies self.association.oclIsTypeOf(Association)

context AssociationEnd **inv**
AssociationrelationtypefromUMLAssociationEndToEJBDataEndusingRule9:
self.ejbEnd_r2->notEmpty() implies self.classifier.getOuterMostContainer() <> self.
type.getOuterMostContainer()

context AssociationEnd **inv**
AssociationEndUppermustbedifferentthan1forUMLAssociationEndEmEJBAssociationusingRule10
:
self.ejbEnd1_r3->notEmpty() or self.ejbEnd2_r3->notEmpty() or self.
ejbDataAssociation->notEmpty() implies self.upper <> '1'

context AssociationEnd **inv**
AssociationofanAssociationEndmustbeanAssociationClassfromUMLAssociationEndEmEJBAssociationusingRule10
:
self.ejbEnd1_r3->notEmpty() or self.ejbEnd2_r3->notEmpty() or self.
ejbDataAssociation->notEmpty() implies self.association.oclIsTypeOf(
AssociationClass)

context AssociationEnd **inv**
outermostclassofanAssociationfanAssociationEndmustbeequaltotheoutermostclassofthetypeofthesameAssociationEndfr
:
self.ejbEnd1_r3->notEmpty() or self.ejbEnd2_r3->notEmpty() or self.
ejbDataAssociation->notEmpty() implies self.association.
getOuterMostContainer() = self.type.getOuterMostContainer()

context AssociationEnd **inv**
lowerEJBAssociationEnd1mustbe0fromUMLAssociationEndEmEJBAssociationusingRule10
:
self.ejbEnd1_r3->notEmpty() or self.ejbEnd2_r3->notEmpty() or self.
ejbDataAssociation->notEmpty() implies self.ejbEnd1_r3.lower = '0'

context AssociationEnd **inv** upperEJBAssociationEnd1mustbe*
fromUMLAssociationEndEmEJBAssociationusingRule10:
self.ejbEnd1_r3->notEmpty() or self.ejbEnd2_r3->notEmpty() or self.
ejbDataAssociation->notEmpty() implies self.ejbEnd1_r3.upper = '*'

context AssociationEnd **inv**
lowerdEJBAssociationEnd2mustbe1fromUMLAssociationEndEmEJBAssociationusingRule10
:
self.ejbEnd1_r3->notEmpty() or self.ejbEnd2_r3->notEmpty() or self.
ejbDataAssociation->notEmpty() implies self.ejbEnd2_r3.lower = '1'

context AssociationEnd **inv**
upperEJBAssociationEnd2mustbe1fromUMLAssociationEndEmEJBAssociationusingRule10
:
self.ejbEnd1_r3->notEmpty() or self.ejbEnd2_r3->notEmpty() or self.
ejbDataAssociation->notEmpty() implies self.ejbEnd2_r3.upper = '1'

context AssociationEnd **inv**
compositionAssociationEnd1mustbefalsefromUMLAssociationEndEmEJBAssociationusingRule10
:
self.ejbEnd1_r3->notEmpty() or self.ejbEnd2_r3->notEmpty() or self.
ejbDataAssociation->notEmpty() implies self.ejbEnd1_r3.composition = false

context AssociationEnd **inv**
bothEJBAssociationEndsmusthavethesameEJBDataAssociationfromUMLAssociationEndEmEJBAssociationusingRule1
:
self.ejbEnd1_r3->notEmpty() or self.ejbEnd2_r3->notEmpty() or self.
ejbDataAssociation->notEmpty() implies self.ejbEnd1_r3.association = self.
ejbDataAssociation and self.ejbEnd2_r3.association = self.ejbDataAssociation

context AssociationEnd **inv**
upperAssociationEndmustbedifferentthan1fromUMLAssociationEndEmEJBAssociationusingRule11
:
self.ejbEnd_r4->notEmpty() implies self.upper <> '1'

context AssociationEnd **inv**
associationofAssociationEndmustbeanAssociationClassfromUMLAssociationEndEmEJBAssociationusingRule11
:
self.ejbEnd_r4->notEmpty() implies self.association.oclIsTypeOf(AssociationClass)

context AssociationEnd **inv**
outermostAssociationmustbeequaloutermostTypeAssociationEndfromUMLAssociationEndEmEJBAssociationusingRule
:
self.ejbEnd_r4->notEmpty() implies self.association.getOuterMostContainer() <> self
.type.getOuterMostContainer()

context AssociationEnd **inv**
lowerEJBAssociation2Endmustbe1fromUMLAssociationEndEmEJBAssociationusingRule11
:
self.ejbEnd_r4->notEmpty() implies self.ejbEnd_r4.lower = '1'

context AssociationEnd **inv**
upperEJBAssociationEnd2mustbe1fromUMLAssociationEndEmEJBAssociationusingRule11
:
self.ejbEnd_r4->notEmpty() implies self.ejbEnd_r4.upper = '1'

context Class **inv** everyoutermostClassbecamesEntityComponent:
self.ejbEntityComponent->notEmpty() or self.ejbDataClass->notEmpty() or self.
ejbDataSchema->notEmpty() or self.ejbServingAttribute->notEmpty() implies
self.isOuterMostContainer()

context Class **inv**
ServingAttributemustbelongtoEntityComponentfromUMLClassToEJBEntityComponent
:
self.ejbEntityComponent−>notEmpty() or self.ejbDataClass−>notEmpty() or self.
ejbDataSchema−>notEmpty() or self.ejbServingAttribute−>notEmpty() implies
self.ejbServingAttribute.classifier = self.ejbEntityComponent

context Class **inv**
ServingAttributeTypemustbeDataClassfromUMLClassToEJBEntityComponent:
self.ejbEntityComponent−>notEmpty() or self.ejbDataClass−>notEmpty() or self.
ejbDataSchema−>notEmpty() or self.ejbServingAttribute−>notEmpty() implies
self.ejbServingAttribute.type = self.ejbDataClass

context Class **inv**
DataClassPackagemustbeDataSchemafromUMLClassToEJBEntityComponent:
self.ejbEntityComponent−>notEmpty() or self.ejbDataClass−>notEmpty() or self.
ejbDataSchema−>notEmpty() or self.ejbServingAttribute−>notEmpty() implies
self.ejbDataClass.package = self.ejbDataSchema

context DataType **inv** verifyUMLDataTypethatmustbetransformedtoEJBKeyClass:
self.ejbDataType−>notEmpty()

context Class **inv** validateIfEveryClasswastransformedtoEJBKeyClass:
self.ejbKeyClass−>notEmpty() and self.ejbId−>notEmpty()

context AssociationClass **inv**
verifyUMLAssociationClassthatmustbetransformedtoEJBKeyClass:
self.ejbKeyClass−>notEmpty()

context Class **inv** verifyUMLClassthatmustbetransformedtoEJBEntityComponent:
not self.feature−>exists(end : AssociationEnd | end.composition = true) implies self.
ejbEntityComponent−>notEmpty() and self.ejbDataClass−>notEmpty() and self
.ejbDataSchema−>notEmpty() and self.ejbServingAttribute−>notEmpty()

context Class **inv** verifyUMLClassthatmustbetransformedtoEJBDataClass:
self.feature−>exists(end : AssociationEnd | end.composition = true) implies self.
ejbDataClass2−>notEmpty()

context Association **inv**
verifyUMLAssociationthatmustbetransformedtoEJBDataAssociation:
self.ends−>exists(end | end.composition = true) implies self.ejbDataAssociation−>
notEmpty()

context AssociationClass **inv**
verifyUMLAssociationClassthatmustbetransformedtoEJBDataClass:
self.ejbKeyClass−>notEmpty()

context Attribute **inv** verifyUMLAttributethatmustbetransformedtoEJBAttribute:
self.ejbAttribute−>notEmpty()

context AssociationEnd **inv**
verifyUMLAssociationEndthatmustbetransformedtoEJBDataEndusingRule8:
self.classifier.getOuterMostContainer() = self.type.getOuterMostContainer() implies
self.ejbEnd_r1->notEmpty()

context AssociationEnd **inv**
verifyUMLAssociationEndthatmustbetransformedtoEJBDataEndusingRule9:
self.classifier.getOuterMostContainer() <> self.type.getOuterMostContainer() implies
self.ejbEnd_r2->notEmpty()

context AssociationEnd **inv**
verifyUMLAssociationEndthatmustbetransformedtoEJBAssociationusingRule10:
self.upper <> '1' and self.association.oclIsTypeOf(AssociationClass) and self.
association.getOuterMostContainer() = self.type->first().
getOuterMostContainer() implies self.ejbEnd1_r3->notEmpty() and self.
ejbEnd2_r3->notEmpty() and self.ejbDataAssociation->notEmpty()

context AssociationEnd **inv**
verifyUMLAssociationEndthatmustbetransformedtoEJBAssociationusingRule11:
self.upper <> '1' and self.association.oclIsTypeOf(AssociationClass) and self.
association.getOuterMostContainer() <> ae.type->first().
getOuterMostContainer() implies self.ejbEnd_r4->notEmpty()

context Operation **inv** verifyUMLOperationthatmustbetransformedtoBusinessMethod:
self.businessMethod->notEmpty()

context Parameter **inv** verifyUMLParamenterthatmustbetransformedtoEJBParameter:
self.ejbParameter->notEmpty()

context Set **inv** verifyUMLSetthatmustbetransformedtoEJBSet:
self.ejbSet->notEmpty()

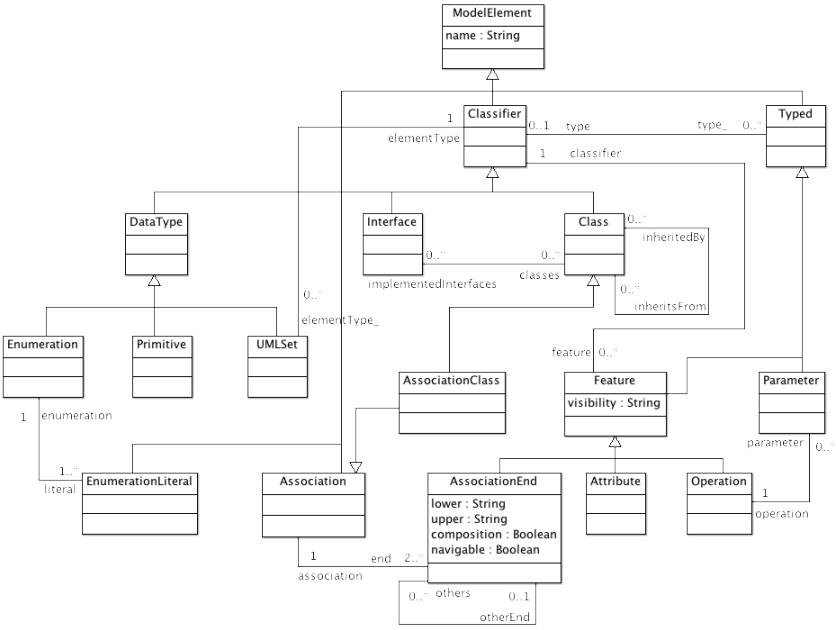


Fig. 1: Simplified UML metamodel

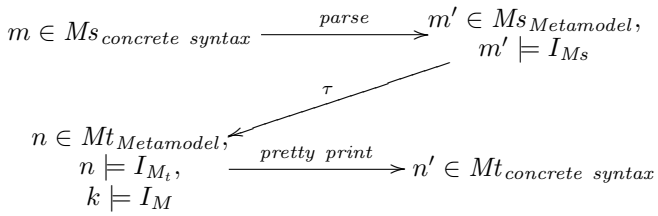


Fig. 2: MDD with transformation contracts

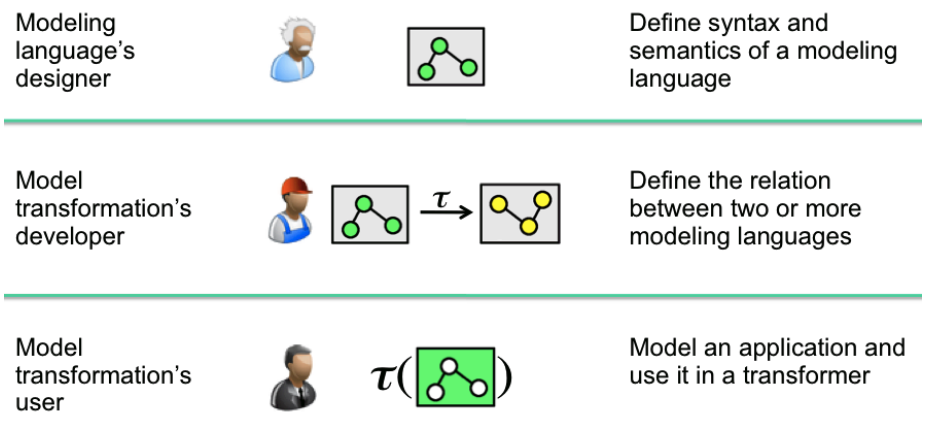


Fig. 3: Types of users that may benefit from the transformation contract approach

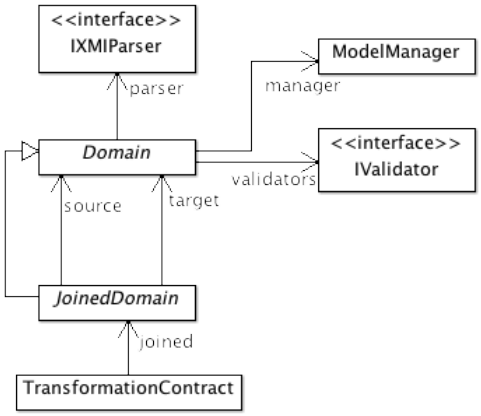


Fig. 4: UML class diagram of the design pattern

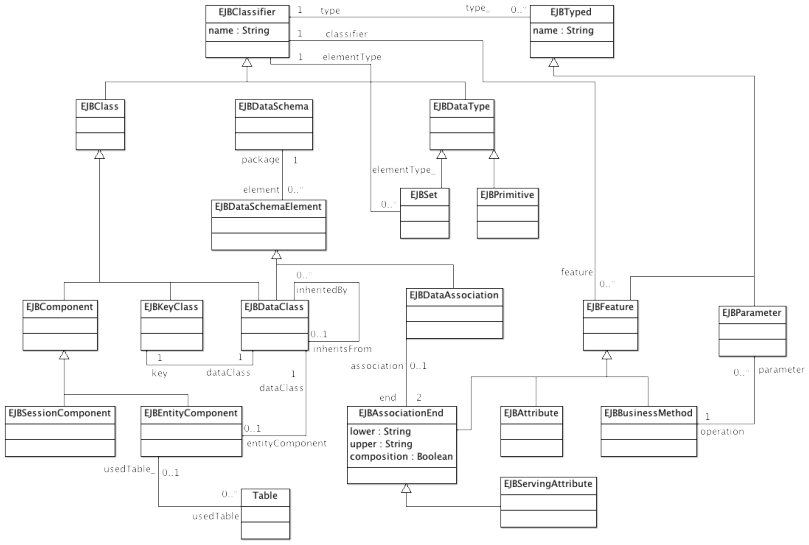


Fig. 5: EJB metamodel used on the transformation from UML to EJB

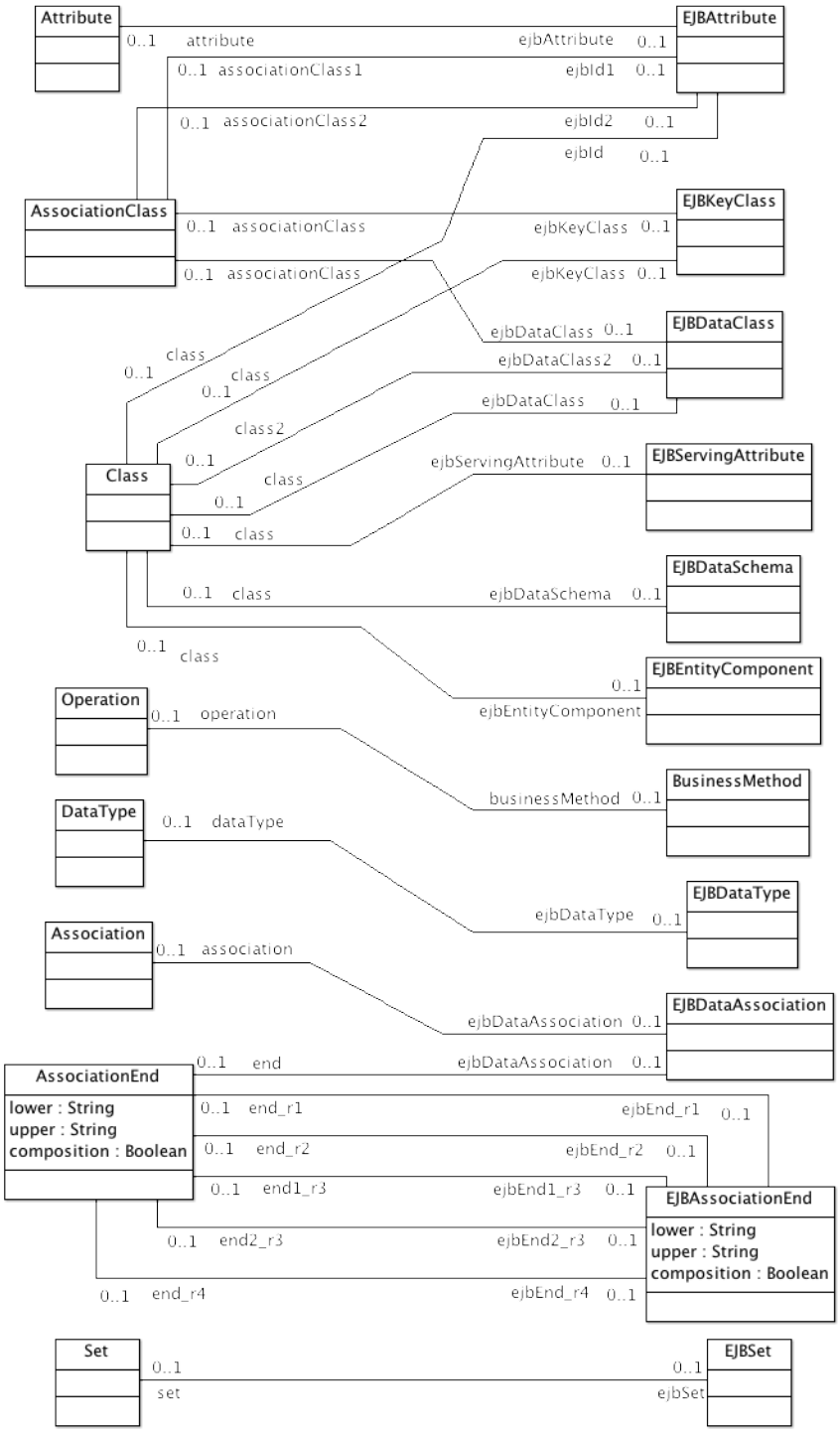


Fig. 6: UMLtoEJB metamodel

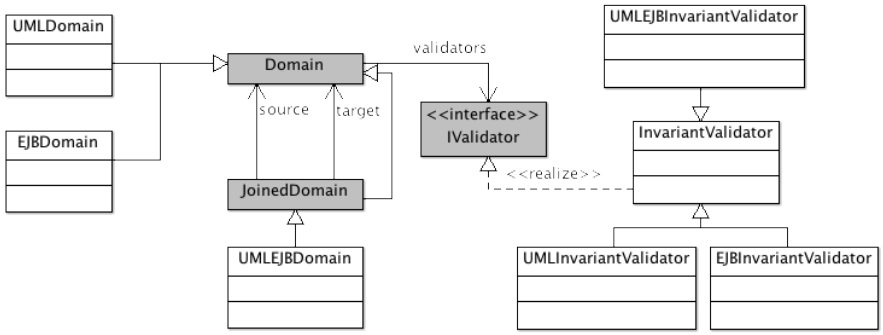


Fig. 7: UML class diagram representing the architecture of the UML2EJB transformer that applies the design pattern.

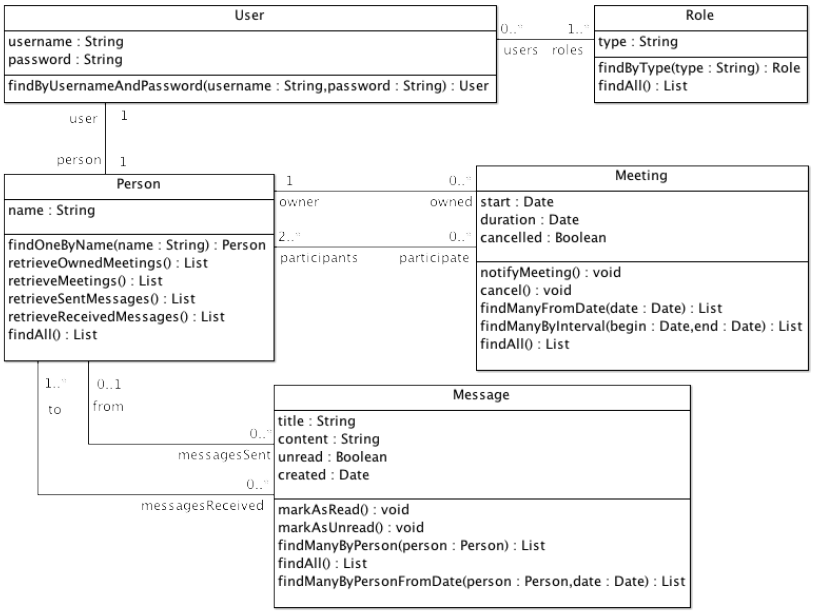


Fig. 8: UML class diagram representing a meeting manager system

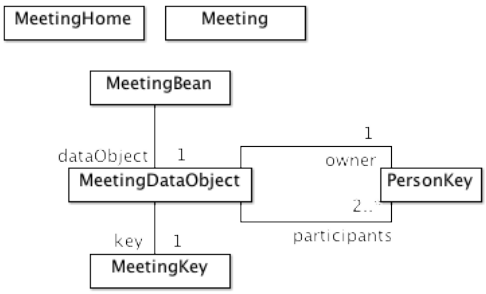
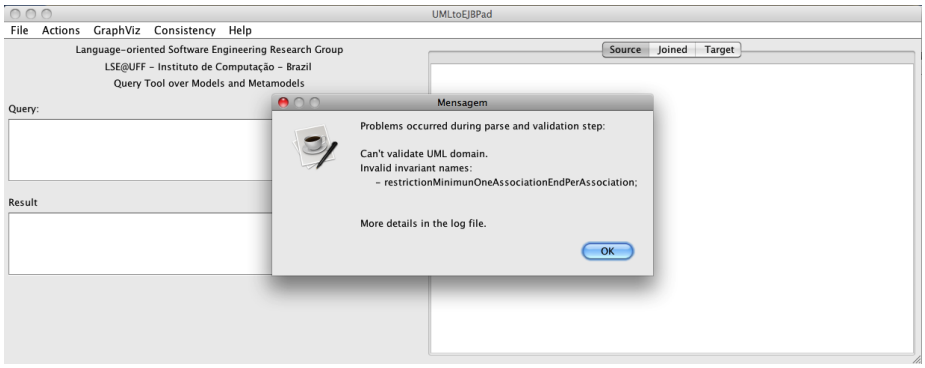


Fig. 9: Part of the transfomation result generated by UMLtoEJB for the meeting example




```
2011-05-30 15:49:48,945 ERROR InvariantError (PrincipalJFrame.java:615)
- Invariant errors:
2011-05-30 15:49:48,953 ERROR InvariantError (PrincipalJFrame.java:617)
- context Association inv restrictionMinimunOneAssociationEndPerAssociation:
Association.allInstances()->forAll(a : Association | a.end->size() >= 1)
Failed objects: ID648801234b48385212c5256123c80000000000000000D21;
```

Fig. 10: Identifying ill-formed models

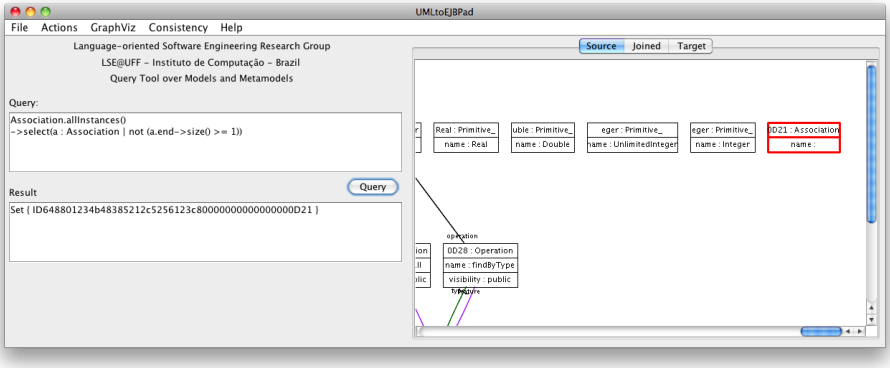
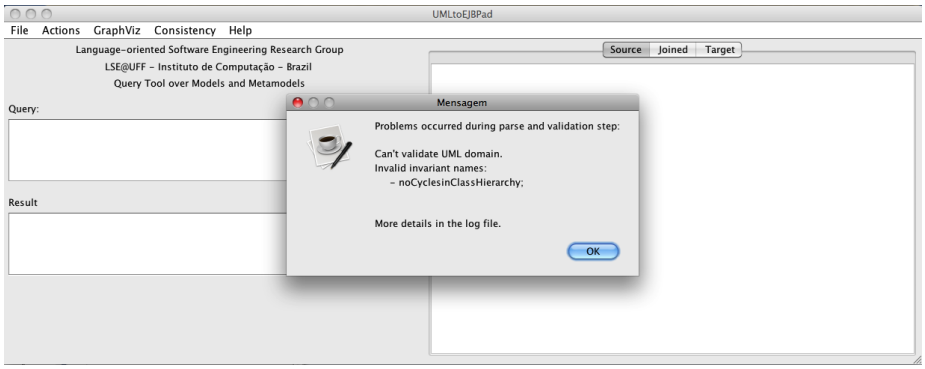


Fig. 11: Understanding the error in an ill-formed model




```
2011-05-30 15:16:06,583 ERROR InvariantError (PrincipalJFrame.java:615)
- Invariant errors:
2011-05-30 15:16:06,606 ERROR InvariantError (PrincipalJFrame.java:617)
- context Class inv noCyclesinClassHierarchy:
Class.allInstances()->forAll(c : Class | c.inheritsFrom->
          forAll(r : Class | r.superPlus()->excludes(c)))
Failed objects: ID648801234b48385212c5256123c80000000000000000D0D,
ID648801234b48385212c5256123c80000000000000000CEA,
ID6488012351a2c16212c4abba32780000000000000000CE2;
```

Fig. 12: Identifying invalid models

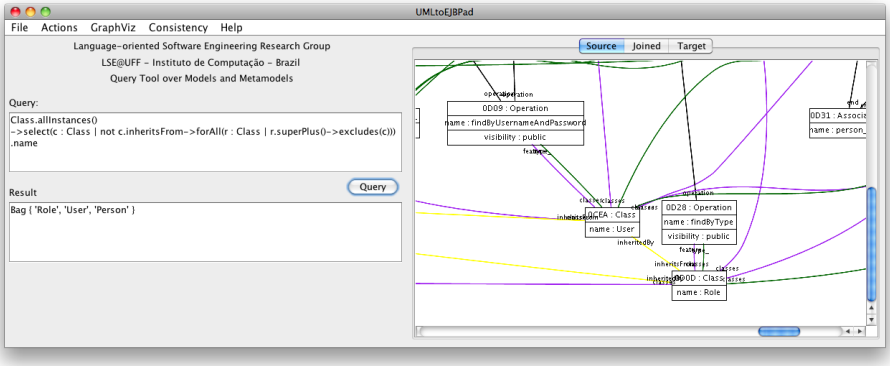


Fig. 13: Understanding the error in an invalid model