On the Specification, Verification and Implementation of Model Transformations with Transformation Contracts

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

Instituto de Computação, Universidade Federal Fluminense, Brazil
{cbraga,rmenzes,tcomicio,cfernando,elandim}@ic.uff.br

Abstract. Model transformations are first-class artifacts in a model-driven development process. As such, their verification and validation is an important task. We have been developing a technique to specify, verify, validate and implement model transformations. Our technique is based on the concept of *transformation contracts*, a specification that relates two modeling languages and declares properties that must be fulfilled in such a relation. A transformation contract is essentially a transformation model that allows for the verification and validation of a model transformation using the same techniques one uses to verify and validate any given model. This paper describes our technique, discusses *consistency* of model transformations and reports on its application to a model transformation from access control models to Java security.

1 Introduction

Model-driven development (MDD, e.g. [15]) 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 may be compilable source-code or other models, in the same or different abstraction levels than the source model. MDD aims at allowing for a *generative* software development process in which applications are produced out of models possibly described at the application domain level.

Model transformations are first-class artifacts in a model-driven development process. As such, their specification, verification and validation are important tasks in an MDD process. A transformation contract [10, 14, 7, 8, 12] is a specification of a model transformation. Essentially, a transformation contract is comprised by *relations* between the model elements of the modeling languages it relates and *properties* that such relations must fulfill. Therefore, a model transformation specification may be understood as a metamodel. In this paper, we follow the terminology of [6] and call the metamodel representing a model transformation a *transformation metamodel*. A particular application of a model transformation is represented as an object model instance of the transformation metamodel. A transformation contract is thus a transformation metamodel and a set of properties over it. Under this perspective, *model reasoning* techniques may be applied to *reason about model transformations* as well.

A. Simao and C. Morgan (Eds.): SBMF 2011, LNCS 7021, pp. 108–123, 2011.

More formally, we say that a model m is *well-formed* with respect to a metamodel M, denoted by $m \in M$, if m is syntactically a proper instance of M, that is, essentially, the objects in m are instances of metaclasses in M and links in m are instances of associations in M. A model m is *in conformance* with M, denoted by $m \models P_M$, if the properties of the metamodel M hold in m. A transformation contract (see Section 4) between two modeling languages M and M' related by a set of associations A, denoted by $M \bowtie_A M'$, is a *pair* comprised by the *transformation metamodel* resulting from the disjoint union of M, M' and A, and *a set of properties* over the transformation metamodel. A model transformation is said *correct* with respect to a transformation contract $M \bowtie_A M'$ iff $m \models P_M \Rightarrow ((m' \models P'_M) \wedge ((m \bowtie_l m') \models P_{M \bowtie_A M'}))$ *if* $m \in M, m' \in M', l \in A, m \bowtie_l m' \in M \bowtie_A M'$, where P_M are the properties of the metamodel M and $m \bowtie_l m'$ is an instance of the transformation metamodel of $M \bowtie_A M'$ with l a set of links well-formed with respect to A.

In [10, 14, 7, 8], the authors specify such properties as invariants in the Object Constraint Language representing *typing rules* of the different metamodels involved in a model transformation. In this paper we move forward by generalizing previous work by allowing the automatic verification of *model transformation consistency* understood as satisfiability of an associated theory in Description Logic [2]. We also discuss the implementation of transformation contracts as the application of a *design pattern* that captures our way of designing transformation contracts in the context of a model transformation from access control models to Java security. Space constraints prevent us from discussing other applications of our approach. We refer the interested reader to `http://lse.ic.uff.br` for a model transformation from UML class diagrams to Enterprise Java Beans code and other tools.

This paper is organized as follows. Section 2 discusses related work. Section 3 describes our proposed model transformation process. Section 4 gives an algebraic definition of transformation contracts and exemplifies it in the context of our example model transformation from access control models to Java security. Section 5 describes how consistency verification may be added to our model transformation process and exemplifies its application. Section 6 reports on a design pattern for the implementation of model transformations according to our proposed model transformation process and exemplifies its application. We conclude this paper in Section 7 with our final remarks.

2 Related Work

Our previous work in [7, 8, 12] agrees in spirit with [10, 14]. However, there are differences at the specification, verification and implementation levels. At the specification level, we adopt a *relational* approach towards the specification of a model transformation, similar in essence to [1, 6] but different from [10, 14] where transformation contracts are specified as OCL invariants from source to target model elements. In [1, 6, 14] their approach is discussed informally while in Section 4 transformation contracts are formalized as algebras. The specification of a relation between the model elements of the metamodels related by a model transformation is essential to generalize from OCL invariants and understand that different *kinds* of properties may be specified *over such relation*. It is important to make *explicit* the relationship among the metamodels.

At the verification level, different kinds of properties may be reasoned upon despite OCL invariants. One such property is model consistency understood as satisfiability of the description logic theory (or knowledge base or TBox, in description logic terminology) associated with a given model. Note that given the perspective that we apply here, model transformations may also be checked for consistency, as it is also a metamodel! At the implementation level, given our generalization, we are not confined to OCL based languages such as OMG's Query View Transformation [16] to specify our transformation contracts. QVT is one possibility, that allows for the specification of the relation among the metamodels related by the model transformation. However, it should be clear that it is one particular implementation of the the transformation metamodel, not the only one. Moreover, we also generalize the understanding of concrete and abstract syntaxes discussed in [1, 14]. There, the authors understand that the concrete syntax of a modeling language must be in bijection with the abstract syntax described by the metamodel. This has the benefit for the model transformation designer to have a parser for any given modeling language. However, this choice is cumbersome for the user since one must create a quite detailed model so the machine may understand what one wants. The idea of a *domain-specific* modeling language is precisely to allow descriptions at the domain level and if possible concise ones. This is what UML profiles are for. In the design pattern described in Section 6, we allow the model transformation designer (actually the modeling language designer) to define how a model must be represented as an instance of a metamodel and its inverse, in the form of *parsing* and *pretty-printing* functions.

QVT and Triple Graph Grammars [17] (TGG) are other possible specification frameworks for a model transformation that requires specific theory and machinery to reason and implement model transformations. The transformation contracts approach proposed in this paper is not biased to any such specification languages and may be used with them as well. Note, however, that QVT is bound to OCL as the specification language for the properties of metamodels and there are properties best specified in other semantic frameworks, such as consistency in DL as discussed in this paper. TGG may not have QVT's restriction on the specification language for metamodel properties but another aspect of our approach is that we apply to model transformation specification design the same specification languages and techniques one would to design a modeling language. No additional framework, such as QVT or TGG, is necessary.

In [6] the authors discuss the idea of a transformation models to specify a model transformation. We share the idea of transformation models but in this work is make precise what we mean by transformation model and how it may be used to reason on model transformations.

3 Model-Driven Development with Transformation Contracts

Model transformations relate *languages*. If one decides to work with the standards of the Object Management Group (OMG) to take advantage of the interoperability gained from using such standards, the abstract syntax of the languages related by a model transformation may be described in the form of a UML class diagram. Such a model is called a *metamodel* since it describes the syntax that models should follow.

A model that is a proper instance of a metamodel is called *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 with respect to 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, slightly enhanced from [15] by considering inheritance between classes through the association *inherited-inheritsFrom*. The metamodel essentially represents the notions of datatypes, classes, attributes, operations, interfaces, association ends, their inter-relations and their typing relations.

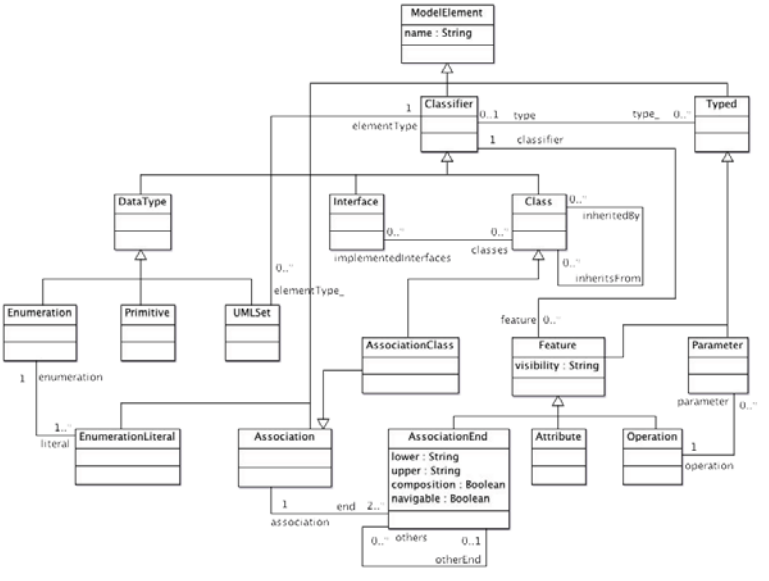


Fig. 1. Simplified UML metamodel

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 M's metamodel. This step is similar to language *parsing* in a compiler.

Up to this point, the model-driven development process we are describing in this paper can be drawn as follows, where m, m', n, n' are models; $\mathcal{S}$ and $\mathcal{T}$ represent the source and target metamodels related by a model transformation τ; we write $m \in M$ to denote that the model m is well-formed with respect to M where M represents the concrete syntax for a modeling language M and $\mathcal{M}$ represents the abstract syntax of the modeling language M; $parse$ is the mapping that given a model m written in the concrete syntax of a modeling language M (S and T in the diagram) generates an abstract syntax version m' of m where m' is well-formed with respect to $\mathcal{M}$; finally, $pretty\ print$ is the inverse mapping of $parse$, that is, it generates the concrete syntax of the modeling language M given a model instance of $\mathcal{M}$,

$$m \in S \xrightarrow{parse} m' \in \mathcal{S} \xrightarrow{\tau} n \in \mathcal{T} \xrightarrow{pretty\ print} n' \in T.$$

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 with respect to M and validity of the invariants of M in m, assuming M consistent, that is, assuming that M has instances. 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 with respect to a language L is ill-typed if the typing rules of L do not apply to p.

One way to specify such invariants is using the Object Constraint Language (OCL). 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 in a model instance of the UML metamodel by verifying for each class c if c is not included in the transitive closure of the *inheritsFrom* relationship that represents class inheritance hierarchy. 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 *emptySet* constructs an empty set of objects of type *Class*.

```
1 context Class inv noCyclesinClassHierarchy: self.inheritsFrom→forAll(r|r.superPlus()→excludes(self))
2 context Class::superPlus():Set(Class) body: self.superPlusOnSet(self.emptySet())
3 context Class::superPlusOnSet(rs:Set(Class)):Set(Class) body:
4 if self.inheritsFrom→notEmpty() and rs→excludes(self)
5 then self.inheritsFrom→collect(c : Class | c.superPlusOnSet(rs→including(self)))→flatten()→asSet()
6 else rs→including(self) endif
```

OCL can be used to *automatically* validate UML models. Considering an implementation of an OCL interpreter, such as [11], one may actually apply the invariants of a metamodel M to a syntactically well-formed model m with respect to 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 syntactically well-formed with respect to M and all invariants in M (such as *noCyclesinClassHierarchy*) hold in m. For example, a UML class diagram must be well-formed with respect to the metamodel in Figure 1 and the invariant *noCyclesinClassHierarchy* should hold on it.

The MDD process adopted in this paper when invariants are considered may be drawn as follows where $I_{\mathcal{M}}$ are the invariants of the metamodel of the modeling language M and $m \models I_{\mathcal{M}}$ means that all the invariants in $I_{\mathcal{M}}$ hold in the model $m \in \mathcal{M}$,

$$m \in S \xrightarrow{parse} m' \in \mathcal{S}, m' \models I_{\mathcal{S}} \xrightarrow{\tau} n \in \mathcal{T}, n \models I_{\mathcal{T}} \xrightarrow{pretty\ print} n' \in T.$$

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 transformation metamodel of the source and target languages related by a set of associations. By transformation metamodel we mean a metamodel $\mathcal{K}$ resulting from a model operation $\mathcal{S} \bowtie_{A_{\mathcal{K}}} \mathcal{T}$ on two given metamodels $\mathcal{S}$ and $\mathcal{T}$ that *extends* the metamodels $\mathcal{S}$ and $\mathcal{T}$ by: (i) disjointly uniting all the model elements of $\mathcal{S}$ and $\mathcal{T}$; (ii) declaring associations $a \in A_{\mathcal{K}}$ that relate classes in $\mathcal{S}$ with $\mathcal{T}$ and disjointly uniting $A_{\mathcal{K}}$ with $\mathcal{S}$ and $\mathcal{T}$; and (iii) declaring invariants $I_{\mathcal{K}}$ over $A_{\mathcal{K}}$. The MDD process adopted in this paper when transformation contracts are considered may be drawn as follows where $\mathcal{K} = \mathcal{S} \bowtie_{A_{\mathcal{K}}} \mathcal{T}$, $k \in \mathcal{K}$, $l \in A_{\mathcal{K}}$ and $k = (m \bowtie_l n)$,

$$m \in S \xrightarrow{parse} m' \in \mathcal{S}, m' \models I_{\mathcal{S}} \xrightarrow{\tau} n \in \mathcal{T}, n \models I_{\mathcal{T}}, k \models I_{\mathcal{K}} \xrightarrow{pretty\ print} n' \in T.$$

4 Specifying Transformation Contracts

In this section we formalize algebraically the concept of transformation contracts and exemplify its specification. We begin with the formal definitions and then, for our example, we represent metamodels as UML class diagrams constrained by expressions in OCL. The equational interpretation of a class diagram means essentially to understand it as an algebraic signature where a class declaration is formalized as a sort declaration with an appropriate constructor operation and an association declaration is formalized as an operation over the appropriate sorts representing the classes that the given association relates. Cardinality constrains and OCL invariants are formalized as equations over the signature defined from class and association declarations. For OCL in particular,

there is a general theory for basic OCL operations which is extended (in a precise algebraic sense) for each OCL constrained class diagram. (We refer the interested reader to [13] for an algebraic formalization of OCL.) The formal definitions below are used in the example, described later in this section, to make it precise.

Definition 1 (Equational theory). *An equational theory $\mathcal{M}$ is a structure $\langle \Sigma, E \rangle$ where Σ is the signature of $\mathcal{M}$ and E is a set of terminating and confluent equations over Σ.*

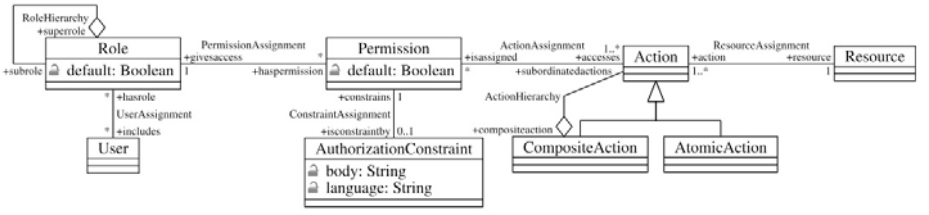
Definition 2 (Metamodel). *A metamodel $\mathcal{M}$ of a modeling language M is an equational theory $\mathcal{M} = \langle C \cup A, I \rangle$ where C is the signature defining the metaclasses of M, A is the signature defining the associations of M, and I is a set of equations over $C \cup A$ representing the invariants of M.*

Definition 3 (Transformation contract). *A transformation contract $S \bowtie_A T$ between modeling languages S and T related by the associations in A is an equational theory $\mathcal{K} = \langle (C_\mathcal{S} \cup A_\mathcal{S}) \uplus (C_\mathcal{T} \cup A_\mathcal{T}) \uplus A_\mathcal{K}, I_\mathcal{S} \cup I_\mathcal{T} \cup I_\mathcal{K} \rangle$, where $\uplus$ is the disjoint union operation over sets, $\mathcal{S} = \langle (C_\mathcal{S} \cup A_\mathcal{S}), I_\mathcal{S} \rangle$ is the metamodel of the modeling language S, $\mathcal{T} = \langle (C_\mathcal{T} \cup A_\mathcal{T}), I_\mathcal{T} \rangle$ is the metamodel of the modeling language T, $A_\mathcal{K}$ is a signature representing associations in A, and $I_\mathcal{K}$ is a set of equations over $A_\mathcal{K}$ representing invariants over the associations between S and T.*

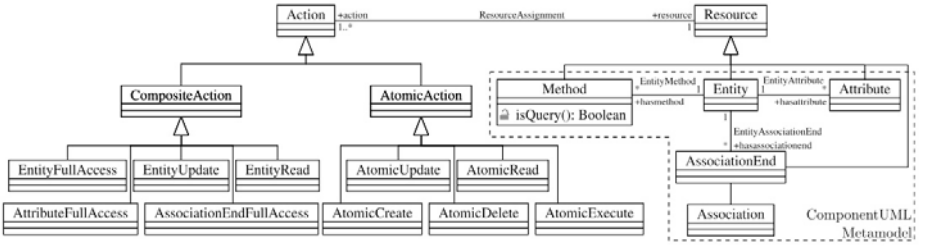
We exemplify the specification of a transformation contract with an excerpt, due to space constraints, of the model transformation from the platform independent modeling language SecureUML+ComponentUML [4], for access control modeling, to a platform specific modeling language we call JAAS that represents the Java Authentication and Authorization Service. This excerpt is part of the SecureUMLtoJAAS+AAC model transformer that generates AspectJ code, an extension of the Java programming language with aspect-oriented concepts, with JAAS support from access control models for Java-based applications. The tool is available for download from `http://lse.ic.uff.br`. The complete description of the model transformation is given in [12].

The modeling languages. SecureUML+ComponetUML is a language to model access control. A SecureUML+ComponentUML model describes permissions that user roles have in order to perform actions over entities. Examples of such actions are: (i) the *execution* of a method, (ii) *updating* an entity's state, or (iii) *full access* to an entity. The first two actions are *atomic actions* and the last one a *composite action*. As opposed to atomic actions, composite actions form a collection of actions which may be atomic or composite. The *EntityFullAccess* composite action, for instance, allows for both read and update access to all elements of an entity, that is, its attributes, methods and association ends. It includes *EntityRead* and *EntityUpdate* which in turn include *AtomicRead* and *AtomicUpdate*, respectively. SecureUML also allows for the modeling of user roles' *hierarchies*. Role inheritance means that if role r_1 inherits from role r_2 than all permissions of r_2 also apply to r_1. The metamodel of SecureUML+ComponentUML is given in Figure 2.

We have defined a modeling language called JAAS, which is the acronym for the Java Authorization and Authentication Service, to capture the access control subset of the Java security framework. Its metamodel is depicted in the diagram in Figure 3.



(a) SecureUML metamodel



(b) ComponentUML metamodel

Fig. 2. SecureUML+ComponentUML metamodel

In JAAS there are different authentication mechanisms such as LDAP or NIS. These mechanisms are captured as instances of a protection domain. The metaclasses *Principal*, *JAASPermission* and *JAASAction* are the counter parts of *Role*, *Permission* and *Action* in SecureUML. We will focus on the transformation from *Role*, *Permission* and *Action* to *Principal*, *JAASPermission* and *JAASAction* in this paper.

The transformation contract. The "raison d'être" of a transformation contract is to guarantee that essential properties of the source model are preserved in the target model. In our example, we want to guarantee that a user in a given SecureUML role is properly represented as a principal, that is, a principal may enact the same actions, with the same constraints, of its associated role, no more no less.

As opposed to SecureUML, JAAS does not have role hierarchies or composite action hierarchies. The transformation contract from SecureUML+ComponentUML model to JAAS models is the result of a *composition* of two contracts: (i) the *flattening* contract $\mathcal{F}$, in which the role and action hierarchies are flattened in SecureUML and (ii) the *mapping* contract $\mathcal{M}$, in which flattened SecureUML and JAAS are related. With the composed contract, a principal will be able to enact the actions associated with the permissions of the role that the given principal is related with, since: (i) flattening the role hierarchy associates with a given role all the permissions of the transitive closure of its inheritance hierarchy and (ii) flattening the action hierarchy associates with a given role all the atomic actions, and their constraints, for each composite action in a given permission.

Recall from Definition 3 that a contract is a structure $\mathcal{K} = \langle (C_{\mathcal{S}} \cup A_{\mathcal{S}}) \uplus (C_{\mathcal{T}} \cup A_{\mathcal{T}}) \uplus A_{\mathcal{K}}, I_{\mathcal{S}} \cup I_{\mathcal{T}} \cup I_{\mathcal{K}} \rangle$. For the flattening contract $\mathcal{F}$, $C_{\mathcal{S}}$ and $A_{\mathcal{S}}$ are the metaclasses and associations from the SecureUML+ComponentUML metamodel. The invariants

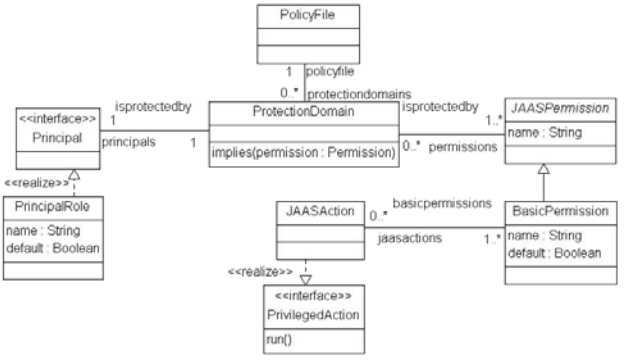


Fig. 3. JAAS metamodel

$I_{\mathcal{S}}$ will *not* be discussed here as they are not relevant to our example. (For the sake of exemplification, one such invariant is the need of a *default role* that every role must inherit from and that has a *default permission* over resources, in any given SecureUML model.) The set $C_{\mathcal{T}}$ includes metaclasses *FRole* and *FAction*, for flattened role and flattened action, respectively. The set $A_{\mathcal{T}}$ includes[1]: (i) a one-to-one association *role-frole* between metaclasses *Role* in SecureUML and *FRole* in flattened SecureUML, (ii) a one-to-one association *atomicaction-faction* between *AtomicAction* in SecureUML and *FAction* in flattened SecureUML, (iii) a one-to-many association *frole-permission* between *FRole* in flattened SecureUML and *Permission* in SecureUML, and finally, (iv) a one-to-many association *faction-permission* between *FAction* in flattened SecureUML and *Permission* in SecureUML. The set $I_{\mathcal{T}}$ is empty since $\mathcal{F}$ is an endogenous transformation in SecureUML that substitutes the role and action hierarchies for equivalent ones without inheritance. Therefore, there are no invariants in the target of $\mathcal{F}$ since the contract is only about flattening.

The set $I_{\mathcal{K}}$ is the interesting one as it specifies the flattening process. The first invariant in $I_{\mathcal{K}}$, roleFlattening, specifies that an *FRole* "mirror" instance of a *Role* in SecureUML model has the same permissions of the reflexive-transitive closure of the superrole relation of the given *Role*. The operation allPermissions is defined in [3] and calculates all the *Permissions* of the transitive closure of the *superrole* relation between instances of *Role* in SecureUML.

```
1 context FRole inv roleFlattening:
2 self.frole-permission→includesAll(self.role-frole→allPermissions())
3 context Role::allPermissions():Set(Permission) body: self.superrolePlus().haspermission→asSet()
4 context Role::superrolePlus():Set(Role) body: self.superrolePlusOnSet(self.superrole)
5 context Role::superrolePlusOnSet(rs:Set(Role)):Set(Role) body:
6 if rs.superrole→exists(r|rs→excludes(r))
7 then self.superrolePlusOnSet(rs→union(rs.superrole)→asSet())
8 else rs→including(self) endif
```

[1] There could be associations between a class in the source metamodel and different classes in the target metamodel. This example is functional but it should be clear that $A_{\mathcal{T}}$ denotes a relation.

The invariant actionFlattening specifies that every *FAction* instance has the same permissions as its *AtomicAction* counterpart which means gathering the permissions of all *CompositeAction* that the given *AtomicAction* is part of together with the permissions attached to the *AtomicAction* itself. The operation allPermissions for *AtomicAction* is calculated in a way similar to *Role* but using the transitive closure of the *compositeactions* relation.

```
1 context FAction inv actionFlattening:
2 self.faction-permissions→includesAll(self.atomicaction-faction→allPermissions())
```

Once the hierarchies are flattened, the mapping contract $\mathcal{M}$ from flattened SecureUML to JAAS, with respect to *FRole*, *Permission*, *FAction*, *PrincipalRole* and *JAASAction*, is trivial. The signatures in $\mathcal{M}$ are given by the metaclasses and associations of flattened SecureUML and JAAS. The set $I_{\mathcal{K}}$ of $\mathcal{M}$ essentially establishes a bijection between *FRole* and *PrincipalRole* and a bijection between *FAction* and *JAASAction*: for every *FRole* there must exist a *PrincipalRole* such that the *PrincipalRole*'s associated instances of *JAASAction* are those related with the instances of *Permission* of the given *FRole*. There are other aspects of model transformation that are handled by $\mathcal{M}$ but are out of the scope of this paper.

5 Verifying and Validating Transformation Contracts

5.1 Model Consistency Reasoning and Description Logic

In Section 3 we have outlined our model-driven development process with transformation contracts. We defined that a model m is in *conformance* with its metamodel $\mathcal{M}$ if $m \models I_{\mathcal{M}}$, that is, if $I_{\mathcal{M}}$, the invariants of $\mathcal{M}$, hold in m. This definition is sound under the assumption that both $\mathcal{M}$ and m are *consistent*, that is, that they may be *instantiated*. An example of inconsistency is as follows: assume that inheritance is a *complete* and *disjoint* relation, that is, if classes B and C inherit from A then A is completely defined by the union of B and C and that B and C are disjoint sets. Now consider that, perhaps after a refactoring operation in a model, B also inherits from C. Clearly, this is an inconsistent model as B can not be included in C and disjoint with C at the same time.

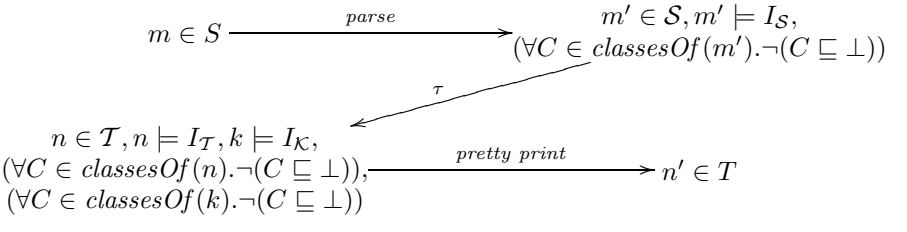
Description logic [2] is a family of logics defined to be efficiently decidable. Each fragment of the logic was carefully studied on its expressiveness and efficiency of reasoning. Consistency reasoning is a decision procedure commonly associated with DL reasoning. DL consistency reasoning may be applied to class diagrams when a proper encoding is defined between class diagrams and DL theories (or knowledge bases, in DL terminology). Such a mapping has been defined in [5], proven correct and the complexity of DL reasoning on class diagrams has been calculated. The encoding of class diagrams in DL essentially relates classes with DL concepts, which denote sets, and associations with DL roles, which are binary relations. Class diagrams are encoded in the logic $\mathcal{ALCQI}$ which is a DL that allows for the specification of: (i) cardinality constraints over roles, denoted by axioms of the general form $\geq\ n\ R.C$ where n is a natural number, R is a role and C is a concept, that constrain the number of individuals (or instances) of C to be at least n in the relation R, (ii) concept negation, denoted by

formulas of the form $\neg C$ where C is a concept, specifying the set of individuals that do not belong to the set denoted by C, (iii) concept conjunction, denoted by formulas of the form $C_1 \sqcap C_2$ where C_1 and C_2 are concepts, which specifies union of the sets denoted by C_1 and C_2 and (iv) definition of inverse of roles, denoted by formulas of the form R^- where R is a role, specifying the inverse relation of R.

For the purposes of this paper, it suffices to explain the encoding for classes, inheritance and binary associations. In [5, Section 7.1] they are described as follows: (i) A *class* C is represented by an atomic concept C; (ii) A *generalization* between a class C and its child class C_1 can be represented using the inclusion assertion $C_1 \sqsubseteq C$. A class hierarchy can be represented by the assertions $C_1 \sqsubseteq C, \ldots, C_n \sqsubseteq C$ when C_i inherits from C. A disjointness constraint among classes $C_1, \ldots, C_n$ can be modeled as $C_i \sqsubseteq \bigsqcap_{j=i+1}^{n} \neg C_j$, with $1 \leq i \leq n-1$, while a covering constraint can be expressed as $C \sqsubseteq \bigsqcup_{i=1}^{n} C_i$; (iii) Each binary association (or aggregation) A between a class C_1 and a class C_2, with multiplicities $m_l..m_u$ and $n_l..n_u$ on each end, respectively, is represented by the atomic role A, together with the inclusion assertion $\top \sqsubseteq \forall A.C_2 \sqcap \forall A^-.C_1$. The multiplicities are formalized by the assertions $C_1 \sqsubseteq (\geq n_l\ A.\top) \sqcap (\leq n_u\ A.\top)$ and $C_2 \sqsubseteq (\geq m_l A^-.\top) \sqcap (\leq m_u A^-.\top)$, where $\top$ denotes the largest concept (top) that includes all concepts and $\forall R.C$ is just syntactic sugar for $\leq 0\ R.\neg C$.

5.2 Model Consistency Verification in Model Transformations with Transformation Contracts

We have incorporated consistency reasoning into our MDD process with transformation contracts. The idea is essentially to check for model consistency before validating the appropriate invariants as it only makes sense to check for invariants of models that are consistent. Concept inconsistency is denoted by $C \sqsubseteq \bot$, where C is a concept and $\bot$ denotes the empty (bottom) concept. A model m is consistent iff $\forall C \in classesOf(m).\neg(C \sqsubseteq \bot)$ where $classesOf(m)$ denotes the set of concepts that encode classes of a model m. When model consistency is considered, our model transformation process may be drawn as follows, where $\mathcal{K} = \mathcal{S} \bowtie \mathcal{T}$, $k \in \mathcal{K}$, and $k = (m \bowtie n)$. Note that checking for consistency of models n and k *is* necessary as the new relations introduced in k may turn classes both in m' and in n inconsistent.



To verify the consistency of a model $m \in \mathcal{M}$ it is necessary, of course, to define an encoding of $\mathcal{M}$ in an description logic, such as the authors did for UML class diagrams into $\mathcal{ALCQI}$ in [5]. The encoding will depend on the reasoning procedure that will be used. There are two types of reasoning procedures in DL. The so-called ABox reasoning means to check that the axioms of a knowledge base (also called TBox) hold on a particular set of individuals (or instances) of concepts and roles. The so-called TBox

analysis means to perform a general symbolic reasoning process over a given TBox that verifies if the axioms of a TBox are generally satisfiable (and not only for a particular set of individuals).

The ABox analysis of a model m instance of a metamodel $\mathcal{M}$ requires the representation of the metamodel $\mathcal{M}$ as a TBox and m as an ABox, that is, a set of individuals. The ABox analysis process consists of checking that the axioms of the TBox representation of $\mathcal{M}$ hold in the ABox representation of m. The TBox analysis of a model m requires an *extension* of the TBox that represents the metamodel $\mathcal{M}$ of m with concepts and roles representing the classes and associations of $\mathcal{M}$, following the encoding defined in 5.1. Essentially, the axioms representing m are defined as follows, assuming m well-formed with respect to $\mathcal{M}$: (i) the concepts that represent metaclasses in $\mathcal{M}$ are subsumed by concepts representing objects, of the appropriate class, in m, (ii) roles representing associations in $\mathcal{M}$ are subsumed by roles representing links, of the appropriate associations, between objects in m, and (iii) include axioms to constrain roles representing links in m, following the encoding in Section 5.1. We have chosen TBox analysis since it is more general than ABox analysis.

As a concluding remark for this section, let us discuss a bit on the combination of consistency verification in DL and invariant validation in OCL. It is out of the scope of this paper, however, a detailed discussion on this subject as the objective of this paper is to discuss how model transformations may be developed rigorously with transformation contracts. It should be clear that consistency verification in DL is *general*. Using DL one may check for: (i) *metamodel consistency*, in other words, answer the question "Does this modeling language admit models?", and therefore reason about the consistency of a modeling language, and (ii) *model consistency*, in other words, answer the question "Does this model admit scenarios?" and therefore reason about the possibility of the instantiation of a particular, that is, the existence of scenarios for a given model. Note that to answer these questions in general we need TBox reasoning. OCL validation executes of OCL invariants on a particular scenario or metamodel instance. It does not allow any reasoning at the modeling language level. However, both techniques allow for reasoning of models. At this point one may wonder why OCL validation is necessary at all if DL reasoning is considered. The answer is that they are *complementary techniques*, as pointed out in [9], in the sense that OCL validation, that is, the execution of OCL invariants over models, identifies errors that DL reasoning may miss. DL has the so-called *open world assumption* which means that a *missing* link between objects, for instance, is not considered an error, as opposed to the so-called *closed world assumption*, where the absence of information, such as a missing link between objects for example, is an error. Therefore, if DL reasoning does not identify a problem because there is missing information in a model then the execution of OCL invariants would. This is the reason why these techniques should be applied sequentially starting with DL reasoning.

5.3 Verifying and Validating Access Control Models

As an illustrative example, let us consider the consistency analysis of access control models in SecureUML+ComponentUML. We discuss two scenarios: (i) DL reasoning

identifies a problem, and (ii) DL reasoning does not identify a problem due to the open world assumption but OCL validation does.

For the first scenario, let us consider a model m, instance of the SecureUML+ComponentUML in Figure 2, that contains an authorization constraint a associated with two permissions p_1 and p_2 through its *ConstraintAssignment* association. The knowledge base of m extends the knowledge base that represents the metamodel of SecureUML+ComponentUML with axioms representing objects and links using the metamodel and model representations required by TBox analysis, described in Section 5.2. An excerpt of the knowledge base, for model elements a, p_1 and p_2 and the concepts they extend, is as follows: (i) from the knowledge base for SecureUML+ComponentUML metamodel,

$$\top \sqsubseteq \forall ConstraintAssignment^{-}.Permission \sqcap \forall ConstraintAssignment.AuthorizationConstraint \quad (1)$$

$$AuthorizationConstraint \sqsubseteq (\equiv 1\ ConstraintAssignment.\top), \quad (2)$$

where$(\equiv 1C.R)$ is syntactic sugar for

$$\geq 1C.R \sqcap \leq 1C.R, C \text{ is a concept and } R \text{ is a role.} \quad (3)$$

$$Permission \sqsubseteq (\geq 0\ ConstraintAssignment^{-}.\top) \sqcap (\leq 1 ConstraintAssignment^{-}.\top) \quad (4)$$

(ii) for authorization constraint a in m, $a \sqsubseteq AuthorizationConstraint$, where a and $AuthorizationConstraint$ are concepts representing the classes with the same names; (iii) for permissions p_1 and p_2, $p_1 \sqsubseteq Permission$ and $p_2 \sqsubseteq Permission$; (iv) for the associations between a and permissions p_i, $i \in \{1, 2\}$:

$$ConstraintAssignment\text{-}a\text{-}p_i \sqsubseteq ConstraintAssignment,$$
$$\top \sqsubseteq \forall ConstraintAssignment\text{-}a\text{-}p_i.a \sqcap \forall ConstraintAssignment\text{-}a\text{-}{p_i}^{-}.p_i,$$
$$a \sqsubseteq (\equiv 1\ ConstraintAssignment\text{-}a\text{-}p_i.p_i),$$
$$p_i \sqsubseteq (\geq 0\ ConstraintAssignment\text{-}a\text{-}{p_i}^{-}.\top) \sqcap (\leq 1 ConstraintAssignment\text{-}a\text{-}{p_i}^{-}.\top).$$

The knowledge base described above is *inconsistent* because axioms 1 to 4 constrain role *ConstraintAssignment* to *exactly* one *Permission* for each *AuthorizationConstraint*. This may *not* be the case if there are individuals from both concepts p_1 and p_2 related to an individual of a.

For the second scenario, let us consider a model m containing an instance of *Action*. According to SecureUML+Component metamodel, there must exist a one-to-one association between a given *Action* and a *Resource*, which is *not* the case in our scenario. Due to the open world assumption, DL reasoning would *not* identify this violation. As we mentioned before, the open world assumption allows us to identify inconsistencies on *given* information. Nothing can be said if the information is not there. This is where OCL validation comes into place. The application of the invariant that constrains the cardinality on the association between *Action* and *Resource* would fail for m as the collection returned by navigating through the association between *Action* and *Resource* from a would produce an empty collection when it should be of size 1.

It should be straightforward to see that the verification and validation illustrated in this section applies to the model transformation context with transformation contracts

described in Section 5.2 as the model one wants to verify and validate is the model resulting from the join of the source and target models. Therefore, the same techniques that one uses to verify and validate a model as an instance of a metamodel can be used to verify and validate a model transformation when specified by a transformation contract since it is a transformation model given by the joined model of the source and target models of a transformation.

6 A Design Pattern for the Implementation of Model Transformations with Transformation Contracts

We have defined a *design pattern* that captures the general process of model transformations with transformation contracts described in Section 3. The design pattern enforces the verification and validation at the different points that they must occur in a model transformation, that is, the analysis of: (i) the source model before the model transformation is applied, (ii) the target model after the transformation is applied and (iii) both source and target models and the associations between them also after the application of the model transformation. By analysis we mean both verification and validation, applying DL reasoning and OCL validation as described in Section 5. As a matter of fact, the design pattern, as well as the model transformation process it implements, is general enough to incorporate new analysis techniques and not only DL reasoning and OCL validation for different modeling languages when the proper encodings are defined, of course.

Figure 4 presents a class diagram of our proposed design pattern. In the pattern, a *Domain* represents a modeling language which interacts with a *ModelManager*, responsible for model persistency, and validators responsible for model analysis. Each *Domain* has a parser from the XMI standard representation, that is, the *Domain*'s concrete syntax, to its metamodel, which is the *Domain*'s abstract syntax. The joined metamodel of a transformation contract is represented by class *JoinedDomain* which references the two instances of *Domain* it relates, named source and target. The class *TransformationContract* declares a *static* method *transform* that executes the model transformation process that we have explained in Section 5. It is static because it is always the same behavior, independently of the actual domains that a particular model transformation relates. The instances of *Domain* will perform the "real" work since parsing, pretty printing and the encodings to the different formalisms are either implemented in methods within classes that inherit from *Domain* or that a *Domain* delegates to its instances.

Figure 5 depicts the application of the design pattern for model transformations with transformation contracts to the model transformation from SecureUML+ComponentUML to JAAS. (As mentioned before, the model transformation also uses aspect-oriented model elements, which are out of the scope of this paper, but this is the reason why the acronym AAC appears in the model.) SecureUML and SecureUML+ComponentUML are coded as different domain classes. The latter extends the former with metaclasses and associations making the action and resource hierarchies more concrete, as explained in Section 4. Moreover, each domain has its own validator class that implements the encoding to the proper reasoner. For DL reasoning, we use

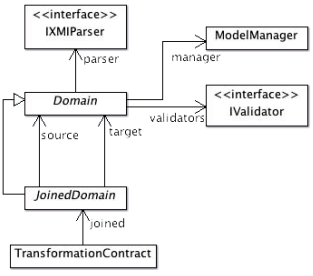


Fig. 4. A design pattern for model transformations with transformation contracts

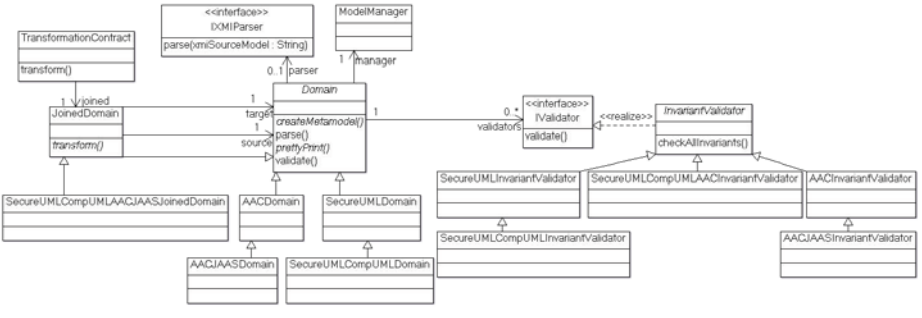


Fig. 5. Applying the design pattern to the SecureUML to JAAS model transformation

the Pellet[2] reasoner and for OCL execution we use EOS[3], which is also used to manage model persistence. Implementing a model transformation as an application of our proposed design pattern enforces the implementation of a rigorous model transformation as different verification and validation techniques can be applied.

7 Final Remarks

We are developing and applying a general technique for the rigorous specification, verification and implementation of model transformations using the concept of transformation contracts. A transformation contract is essentially a transformation metamodel that relates metamodels and a set of properties over the transformation metamodel. Implementations of model transformations are realized as an application of a design pattern that enforces our proposed model transformation process. In this paper we have used the standardized metalanguages of UML class diagrams, OCL for the specification of metamodels and invariants over them and Description Logic to verify consistency. However, our approach is not coupled with any particular choice of metalanguages and different

[2] http://clarkparsia.com/pellet/
[3] http://www.bm1software.com/eos/

metanotations and reasoners may be employed in the development of a model transformation. We plan to continue our work by integrating different automated analysis techniques to our model transformation process and to apply our approach to industrial case studies.

References

1. Akehurst, D.H., Kent, S.: A relational approach to defining transformations in a metamodel. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 243–258. Springer, Heidelberg (2002)
2. Baader, F., Diego Calvanese, D.M., Nardi, D., Patel-Schneider, P.: The Description Logic Handbook. Cambridge University Press (2003)
3. Basin, D., Clavel, M., Doser, J., Egea, M.: Automated analysis of security-design models. Inf. Softw. Technol. 51(5), 815–831 (2009)
4. Basin, D., Doser, J., Lodderstedt, T.: Model driven security: From uml models to access control infrastructures. ACM Trans. Softw. Eng. Methodol. 15(1), 39–91 (2006)
5. Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams. Artif. Intellig. 168, 70–118 (2005)
6. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model Transformations? Transformation Models! In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 440–453. Springer, Heidelberg (2006)
7. Braga, C.: From access control policies to an aspect-based infrastructure: A metamodel-based approach. In: Chaudron, M.R.V. (ed.) MODELS 2008. LNCS, vol. 5421, pp. 243–256. Springer, Heidelberg (2009)
8. Braga, C.: A transformation contract to generate aspects from access control policies. J. of Software and Systems Modeling (2010), doi:10.1007/s10270-010-0156-x
9. Braga, C., Hæusler, E.H.: Lightweight analysis of access control models with description logic. Innov. in Systems and Soft. Eng. 6, 115–123 (2010)
10. Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: OCL for the specification of model transformation contracts. In: Proc. of OCL and Model Driven Eng. Work., pp. 69–83 (2004)
11. Clavel, M., Egea, M., de Dios Miguel Angel, G.: Building an efficient component for OCL evaluation. ECEASST 15 (2008)
12. Comicio, T.: A transformation contract approach for model-driven security. Master's thesis, Universidade Federal Fluminense (2011)
13. Egea, M.: An Executable Formal Semantics for OCL with Applications to Model Analysis and Validation. PhD thesis, Universidad Complutense de Madrid (2008)
14. Gorp, P.V., Janssens, D.: Cavit: a consistency maintenance framework based on transformation contracts. In: Transformation Techniques in Soft. Eng., Dagstuhl Seminar Proc., vol. 05161 (2006)
15. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained. Addison-Wesley, Reading (2003)
16. OMG. MOF QVT final adopted specification, omg adopted specification ptc/05-11-01 (2005)
17. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)