International Journal of Software Engineering
and Knowledge Engineering
Vol. 18, No. 7 (2008) 895–918
© World Scientific Publishing Company

A TRANSFORMATION-BASED APPROACH TO APPLICATION MODEL DEVELOPMENT: CLASS DIAGRAM GENERATION

AJAREE NACO* and VILAS WUWONGSE†

Computer Science and Information Management Program,
School of Advanced Technologies, Asian Institute of Technology,
P.O. Box 4, Klongluang Pathumthani 12120, Thailand
**ajaree@cs.ait.ac.th*
†vw@cs.ait.ac.th

CHUTIPORN ANUTARIYA

School of Information and Communication Technology,
Shinawatra University, 99 moo 10, Bangtoey Samkok, Patumthani 12160, Thailand
chutiporn@shinawatra.ac.th

Received 15 October 2006
Revised 3 May 2007
Accepted 25 June 2007

Application software development is normally the process of developing slightly different application models of the same application domain for different companies. Each application model corresponds to general features of the application domain and additionally involves supplementary requirements of a particular company. An application model for each individual company is always redesigned even though the general features can be reused. This leads to the need for new approaches to the development of a generic application model that can be reused with respect to a company's requirements. This paper presents a new development approach, namely Transformation-based Model Generation (TMG), which enhances the productivity of application development. It focuses on the reuse of a generic application model to automatically generate specific application models that satisfy different requirements of the same problem domain. The concepts of model representation and model generation are taken from Model Driven Architecture (MDA). XML Declarative Description (XDD) is the underlying theory for implementing the TMG. The knowledge for generating specific application models is domain-independent, hence it can be applied to various problem domains.

Keywords: Model driven architecture; UML; model generation; XML declarative description.

1. Introduction

Application software development is normally the process of developing slightly different application models of the same application domain for different individual companies. Each application model reflects the common general features of the application domain and in addition incorporates supplementary requirements of a particular company. Consider, for example, bank account models, core classes of application models in the banking domain. A common general feature of such models is the containment of Saving-Accounts and Current-Accounts as they are general account types provided by all banks. The associations between customers and bank accounts can be considered as general features as well. A general class diagram for all banks can be modeled and depicted as in Fig. 1(a). Differences in bank account models come from new services provided by a particular bank (e.g., new account types) and some additional constraints or specific requirements (e.g., no interest for a certain account type). These differences can be considered as supplementary requirements of a particular design case (e.g., development of an account system for Citibank). Therefore, the application model in Fig. 1(a) can be customized or specialized into several bank account models using additional requirements. For instance, the Islamic account model (Fig. 1(c)) is a model that is generated from Fig. 1(a) by including specific requirements (Fig. 1(b)) required by Islamic law for use by Muslims. Consequently, the general features of an application domain can be modeled and treated as a metamodel of its corresponding specific application models. This pre-designed metamodel can be reused to generate specific application models (e.g., Citibank account model or Future Land Bank account model) by inclusion of additional requirements from a design case.

Model Driven Architecture (MDA) [22] has been proposed and accepted as a new approach to software development. It provides a level of abstraction that separates the development of domain models from the implementation technologies. MDA separately defines two types of model: the Platform Independent Model (PIM) and the Platform Specific Model (PSM). PIM represents a domain model that is independent from implementation details, while PSM contains specific platform details such as CORBA components. The MDA-based software development is the chain of refinement from PIM to PSM and PSM to source code. The literature on MDA [7, 14, 17, 26] has focused on standardizing the transformation method from PIM to PSM (e.g. QVT RFP [27], graphical approaches [16, 29]), which are mostly specific to EJB components, i.e., EJB for J2EE server [6], and Web application [12, 18]. Although PIM plays an important role in MDA, the method and process of constructing models at PIM-level do not exist [15].

This paper proposes a concrete MDA-based approach called Transformation-based Model Generation (TMG) for developing a specific application model of a domain of interest. TMG's main feature is the distinction between generic and specific application models at the PIM-level and its assumption that implementation models are obtained by mapping from their specific application models at the PIM-level. This approach extends the MDA concepts to cover the early stage of the software development process. TMG emphasizes the reuse of generic application models and automatic generation of specific application models. A generic application model defines the knowledge on the domain that the application system needs to know. It is represented in terms of a UML metamodel, which is augmented with

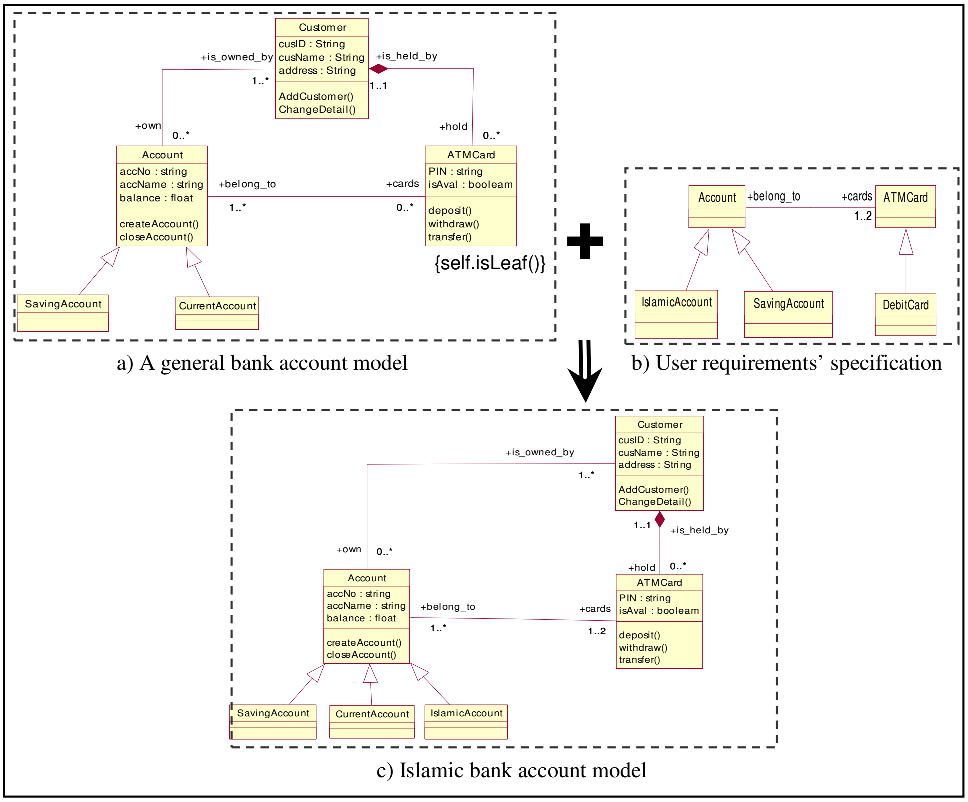


Fig. 1. Class diagrams of bank account models.

precise semantics described in terms of Object Constraint Language (OCL) [28]. A specific application model is a result of generated models and is represented by means of a UML model.

TMG employs XML Declarative Description (XDD) [30] as the underlying implementation theory. XDD provides extensive rule-oriented facilities for reasoning with UML diagrams which are encoded in the XML Metadata Interchange (XMI) format [24], a standard text-based format for UML. Model generation is carried out automatically via UML/XMI and constraints that are represented in XML format by applying General Transformation Knowledge (GTKL). GTKL is represented in terms of XDD clauses and is a domain-independent knowledge. It can be reused to generate specific application models of any application domain. The reusability of a generic application model reduces development time and ensures consistent quality that enhances the capability of TMG.

Section 2 recalls the concepts of model transformation and XDD. Section 3 presents the TMG approach. Section 4 demonstrates the generation of a specific application model using the TMG approach. Section 5 discusses related works, and Sec. 6 draws conclusions and suggests further work.

2. Basic Concepts

2.1. *Model transformation*

Model transformation is a process of producing target models from a given source model by using predefined relationships between elements. A target model can be either a source code or another model [22]. It can be considered as a specialized case of its source model with the inclusion of more specific requirements or details of platform technologies. Based on the details added to a source model, the result of transformations can be either model refinement or model generation [8]. Model refinement is an enhancement of a source model rather than creating a new model from an existing one. This kind of transformation is typically applied when there are incompleted transformations, necessitating human intervention to add information that cannot be deduced. It normally occurs in the transformation of the same kind of models (PIM-to-PIM, PSM-to-PSM) and can be considered as a horizontal transformation since the transformation is acted at the same level of abstraction. Model generation is a creation of new independent model elements in a target model that occurs frequently during the PIM-to-PSM transformation and is considered to be a vertical transformation of models at different levels of abstraction.

The purpose of this paper is to propose an approach to PIM-level model refinement, the outcome of which can serve as a source model for the PIM to PSM transformation phase.

2.2. *XML Declarative Description* (*XDD*)

XDD [30] is an XML-based knowledge representation, which extends ordinary well-formed XML elements by incorporation of variables for an enhancement of expressive power and representation of implicit information into so called *XML expressions*. Ordinary XML elements — XML expressions without variables — are called *ground XML expressions*. Every component of an XML expression can contain variables, e.g., its expression or a sequence of sub-expressions (*E-variables*), tag names or attribute names (*N-variables*), strings or literal contents (*S-variables*), pairs of attributes and values (*P-variables*) and some partial structures (*I-variables*). Every variable is prefixed by '\$*T*:', where *T* denotes its type; for example, \$*S*:value and \$*E*:expression are *S*- and *E*-variables, which can be specialized into a string or a sequence of XML expressions, respectively.

An *XDD description* is a set of *XML clauses* of the form:

$$H \leftarrow B_1, \ldots, B_m, \beta_1, \ldots, \beta_n\,,$$

where $m, n \geq 0$, H and B_i are XML expressions, and each β_i is a predefined *XML constraint* — useful for defining a restriction on XML expressions

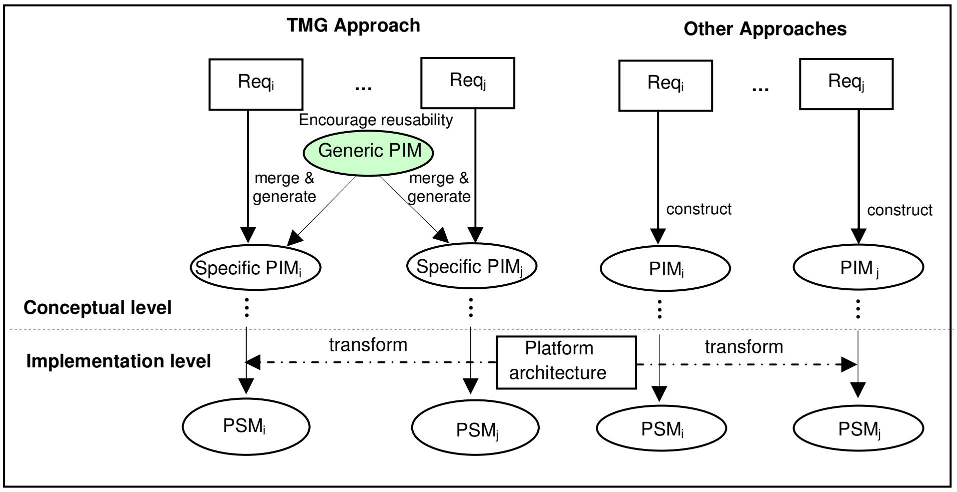


Fig. 2. Transformation-based Model Generation (TMG) approach.

or their components. The XML expression H is called the *head*, and the set $\{B_1, \ldots, B_m, \beta_1, \ldots, \beta_n\}$ is the *body* of the clause. When the body is empty, such a clause is referred to as an *XML unit clause* and the symbol '$\leftarrow$' will often be omitted; hence, an XML element or document can be mapped directly onto *a ground XML unit clause*. Given an XDD description D, its meaning is the set of all XML elements which are directly described by and derivable from the unit and non-unit clauses in D, respectively.

Since every UML-supported tool is capable of reading and writing models using XMI — a standard text-based representation for UML. XDD provides the facility to reason directly with UML diagrams in terms of XMI format by expressing transformation rules in terms of XDD clauses.

3. Transformation-based Model Generation (TMG) Approach

In the context of TMG (Fig. 2), for each application domain, models at the PIM-level can be split into two kinds with respect to the refinement by means of the addition of more details into its models: a generic PIM and one or more specific PIMs. The generic PIM holds the general features of the application domain, while the specific PIMs contain the model elements that correspond to the generic PIM and those that fulfill specific requirements of individual design cases. Several specific PIMs can be generated from a generic PIM using model transformations. With reference to the Meta Object Facility (MOF) [21] abstraction, generic PIMs are defined at the metamodel layer while specific PIMs at the model layer. Model transformation, which includes inconsistency resolving capability, will incorporate specific requirements for transforming a generic PIM into specific PIMs; hence a specific PIM is always consistent with its generic PIM. Generic PIMs and specific

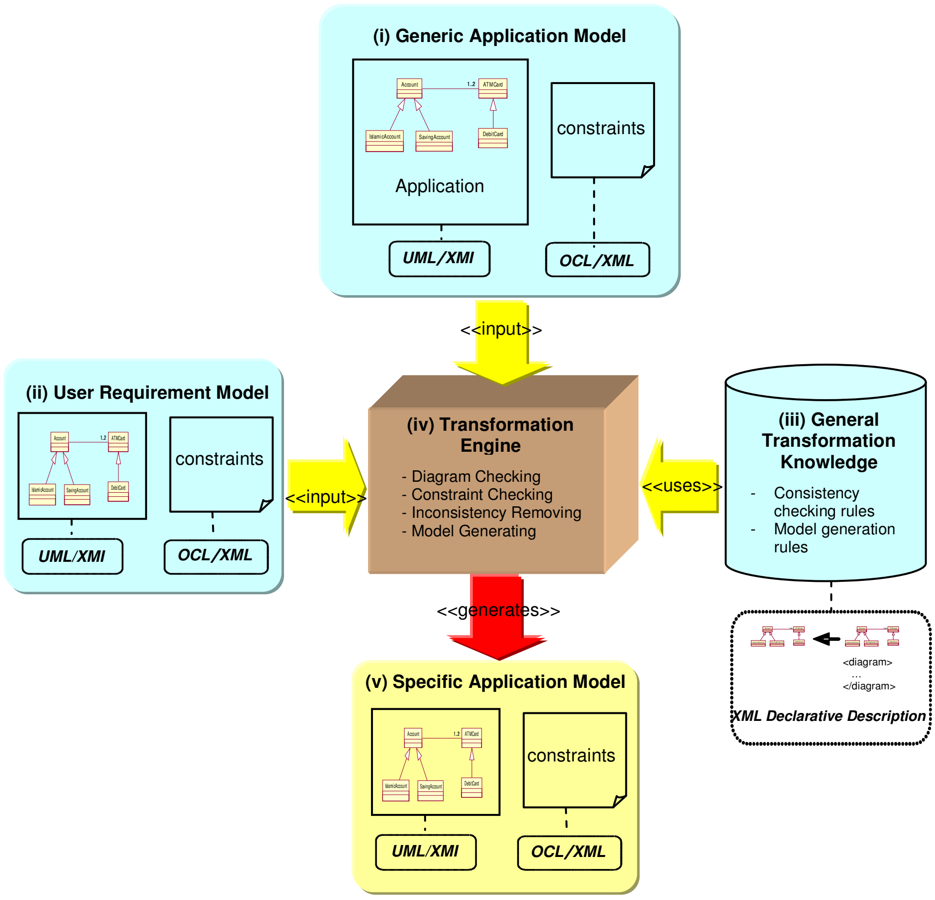


Fig. 3. The components of TMG approach.

PIMs are generally called *generic application models* and *specific application models*, respectively.

3.1. *TMG components*

The components of TMG are (see Fig. 3):

(i) *Generic Application Model*: A generic application model is a metamodel constructed by domain experts. The model assembles the general features needed for developing an application system in that domain and represents them as the domain knowledge. It consists of two components: *application metamodel* and *application constraints*. An application metamodel describes the syntax and semantics of the domain and represents them in terms of a UML metamodel. These metamodel elements can be defined as either mandatory or optional. Application constraints, expressed in terms of *Object Constraint Language*

(*OCL*), describe conditions and restrictions on a specific application model construction. Since OCL has no interchangeable standard in XML, this paper employs OCL/XML schema [19] for encoding OCL invariants in XML. Hence, both metamodels and constraints can immediately be represented by their corresponding XML syntax, i.e., UML/XMI and OCL/XML, respectively.

(ii) *User Requirement Model*: Based on a given application metamodel, a user requirement model is constructed in order to specify the requirements of a design case (such as Citibank requirements), which may include additional classes, attributes, methods, associations or constraints, or may modify or remove unnecessary ones. These requirements are represented by means of UML diagrams and OCL expressions, and hence can be simply encoded in UML/XMI and OCL/XML, respectively.

(iii) *General Transformation Knowledge* (*GTKL*): This is a kind of meta-knowledge, used to generate model elements of a specific application model from its generic one by incorporating the corresponding user requirements. The knowledge is independent of application domains; therefore it can be applied to various problem domains. It can be considered as the most important component since it has a set of knowledge for model transformation and has a capability to handle inconsistencies. Detailed explanations as well as examples of expressing GTKL as XML clauses of XDD theory are provided in the next sub-section.

In essence, GTKL consists of:

- *Consistency checking knowledge*: A specific application model should be consistent with its corresponding metamodel; therefore any inconsistencies which arise should be removed. This knowledge is used to detect and resolve the inconsistencies between the generic application model and user requirement model.
- *Model generation knowledge*: This knowledge is used to construct a new model element of a specific application model, which could be constructed in three ways: (i) copy a mandatory element from the metamodel, (ii) copy a new element from the user requirement model, and (iii) generate a conflict-resolved model element when the corresponding elements of the metamodel and user requirements are inconsistent.

(iv) *Transformation Engine*: The engine works by means of Equivalent Transformation computation model [1, 2]. It regards the components (i) generic application model and (ii) user requirement model as its input data, which are computed based on the knowledge defined by the component (iii) GTKL, and hence yields a specific application model (i.e., the component (v)) conforming to the given metamodel and satisfying the input requirements. In essence, the computation is performed by semantically and equivalently transforming the components (i) and (ii) successively, using the knowledge GTKL as the transformation rules, until the resulting specific application model is obtained.

Since each transformation step is semantics-preserving, the correctness of the obtained result is always guaranteed.

(v) *Specific Application Model*: A specific application model, automatically generated from its corresponding generic one by merging the user requirement model, is expressed using UML/XMI representation, which can be transformed back to its corresponding UML model using the reverse engineering method provided by UML CASE tools. This model can be considered as a PIM in other MDA-based software development approaches, and serves as an input for the next transformation steps.

It can be seen that the main benefit of this TMG approach is that once a generic application model is created, several specific application models can be generated by incorporating particular user requirements. Moreover, since GTKL is domain-independent, it can be applied to generate a specific application model within any particular application domain. Hence, reusability is another benefit of this approach.

3.2. *General Transformation Knowledge (GTKL)*

As illustrated by Fig. 3, GTKL plays an important role in defining consistency checking and model generation processes. This section discusses how GTKL is formulated as corresponding XML clauses in order to serve such purposes. Note that for ease of presentation and understanding, the XML clauses formulated here will be partially represented in terms of corresponding UML graphical notation instead of solely representing them in UML/XMI syntax.

GTKL comprises two types of knowledge: *consistency checking* and *model generation knowledge*. The former will be discussed first, and followed by the latter.

3.2.1. *Consistency checking knowledge*

Since a specific application model should be consistent with its corresponding metamodel, inconsistencies which may arise should be removed. Consistency checking knowledge is used to detect and resolve any inconsistencies, which can be further classified into *diagrammatic* and *constraint checking knowledge* as follows:

- Diagrammatic checking rules are used to identify whether the model elements of the application metamodel and those of the user requirements are consistent or not. In general, the consistency is checked by comparing the property value of each corresponding model element of user requirements against that of the metamodel element. After the comparison, three possible cases may occur: (i) *consistent* — the property values are similar, (ii) *tolerable* — the property values are different but acceptable, and (iii) *inconsistent* — the property values are acutely disparate.
- Constraint checking rules: Application constraints of a generic application model, formalized as OCL constraints, can be classified into two cases: (i) a restriction

on a model element's attribute value such as a restriction on an aggregation value of an association end role; (ii) a restriction on incorporating a new model element such as a restriction that a class cannot have a subclass. A constraint checking rule is used to resolve any inconsistencies which may arise from these two cases.

Definition 1. A diagrammatic checking rule is formulated as an XML clause of the form:

```
<DiagrammaticChecking>
   <Consistency  type  = t  name = n > c   </Consistency>
   <MMElement>       m       </MMElement>
   <ReqElement>      r       </ReqElement>
   <DiagramStatus>   s       </DiagramStatus>
</DiagrammaticChecking>
                                  ←      b1, b2 , ..., bn .
```

where

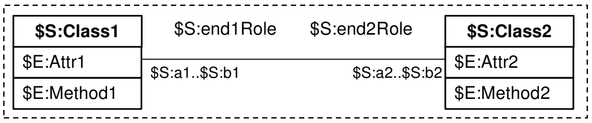
t $\in$ {"AssoicationEnd", "Class", "Attribute"}
specifies the element's type of consistency checked,
n specifies the name of the element t,
c specifies the property of the element t,
m specifies the value of property c that is extracted from the metamodel,
r specifies the value of property c that is extracted from the requirement,
s $\in$ {"consistent", "tolerable", "inconsistent"}
defines the status of the checking element,
b_i are XML expressions or XML constraints which extract a model element from the metamodel and the requirement, and then verify their consistency.

Example 1. *Diagrammatic checking rule*

Figures 4 and 5 present an example of diagrammatic checking rules represented as XML clauses. The clause C_{diagram} emphasizes the consistency between the lower multiplicity value of a particular association specified within a metamodel and that within a user requirement model. Its head describes the generated output, which includes the name ($S:end1Role) and type of elements being considered (AssociationEnd), the values extracted from the metamodel ($S:a1) and the user requirement ($S:c1) as well as the assigned inconsistency status ($S:d_status). The first two elements in the clause's body extract the corresponding lower multiplicity values from the metamodel and the user requirement diagram, respectively. The last element in the clause's body then assigns the appropriate consistency status for the extracted values.

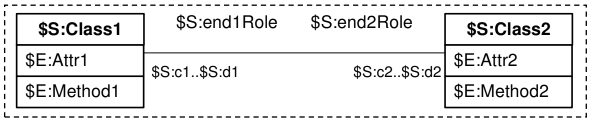
```
Cdiagram : <DiagrammaticChecking>
               <Consistency type="AssociationEnd" name=$S:end1Role>
                    MultiplicityRange.lower </Consistency>
               <MMValue> $S:a1 </MMValue>
               <ReqValue> $S:c1 </ReqValue>
               <DiagramStatus> $S:d_status </DiagramStatus>
           </DiagrammaticChecking>

                                    ←
```

Metamodel



// Extracting the lower
// multiplicity value (**$S:a1**) of a
// particular association which
// relates the class **$S:Class1** to
// the class **$S:Class2** and is
// specified within the
// metamodel.

RequirementModel



// Extracting the lower
// multiplicity value (**$S:c1**) of a
// particular association which
// relates the class **$S:Class1** to
// the class **$S:Class2** and is
// specified within the
// requirement.

```
<DiagrammaticKnowledge property="MultiplicityRange.lower">
      <MMValue> $S:a1 </MMValue>
      <ReqValue> $S:c1 </ReqValue>
      <Status> $S:d_status </Status>
</DiagramaticKnowledge> .
```

Fig. 4. An example of a diagrammatic checking rule.

```
Cknowledge1:  <DiagrammaticKnowledge property="MultiplicityRange.lower">
                   <MMValue> $S:Metamodel     </MMValue>
                   <ReqValue> $S:Requirement  </ReqValue>
                   <Status>    consistent      </Status>
              </DiagrammaticKnowledge>
                          ←              [$S:Metamodel = $S:Requirement].

Cknowledge2:  <DiagrammaticKnowledge property="MultiplicityRange.lower">
                   <MMValue> $S:Metamodel     </MMValue>
                   <ReqValue> $S:Requirement  </ReqValue>
                   <Status>    tolerable       </Status>
              </DiagrammaticKnowledge>
                          ←              [$S:Metamodel != $S:Requirement].

Cknowledge3:  <DiagrammaticKnowledge property="Aggregation Value">
                   <MMValue> $S:Metamodel     </MMValue>
                   <ReqValue> $S:Requirement  </ReqValue>
                   <Status>    consistent      </Status>
              </DiagrammaticKnowledge>
                          ←              [$S:Metamodel = $S:Requirement].

Cknowledge4:  <DiagrammaticKnowledge property=" Aggregation Value">
                   <MMValue> $S:Metamodel     </MMValue>
                   <ReqValue> $S:Requirement  </ReqValue>
                   <Status>     tolerable      </Status>
              </DiagrammaticKnowledge>
                          ←              [$S:Metamodel != $S:Requirement].
```

Fig. 5. Examples of inconsistency status definitions for checking lower multiplicity and aggregation values.

The clauses $C_{\text{knowledge1}}$ and $C_{\text{knowledge2}}$ define how this inconsistency status of lower multiplicity value is obtained. In particular, clause $C_{\text{knowledge1}}$ defines that: if such a value extracted from the user requirement (denoted by the variable $S:Requirement) is equivalent to the corresponding one extracted from the metamodel (denoted by the variable $S:Metamodel), then the inconsistency status is "consistent". On the other hand, clause $C_{\text{knowledge2}}$ states that if the extracted values are different, the inconsistency status is "tolerable". In other words, although the values are not consistent they are acceptable, meaning that the inconsistency can be tolerated. The other two clauses $C_{\text{knowledge3}}$ and $C_{\text{knowledge4}}$ define the inconsistency status of an aggregation value which is similar to the previous one. The value of the inconsistency status is obtained by using the basic knowledge of UML diagrams and other research results on UML semantics [9]. Other types of diagrammatic checking rules, such as upper multiplicity value, navigation value, etc. can be formalized in the same manner.

Definition 2. A constraint checking rule is formulated as an XML clause of the form:

```
<ConstraintChecking>
   <Consistency  type = t   name = n >   c     </Consistency>
   <MMConstratin>         m     </MMConstraint>
   <ReqElement>           r     </ReqElement>
   <ConstraintStatus>     s     </ConstraintStatus>
</ConstraintChecking>
                          ←             b1, b2 , ..., bn .
```

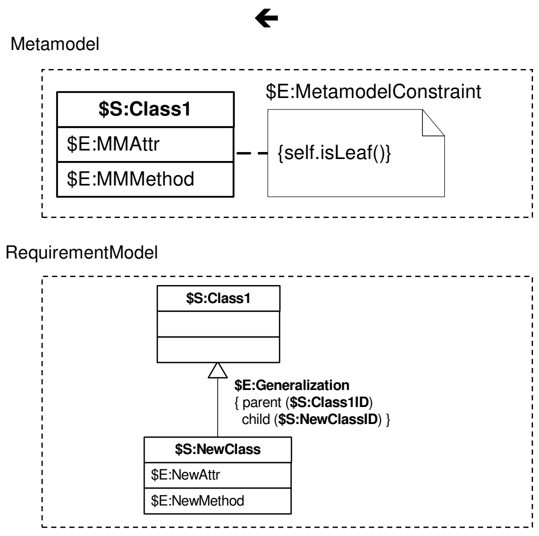
where

t	specifies the type of constraint consistency checked,
n	specifies the name of the element being considered,
c	specifies the property of the element n,
m	specifies the constraint attached to the metamodel's element n,
r	specifies the model element or the attribute value extracted from the requirement corresponding to the element n,
s	$\in$ {"consistent", "tolerable", "inconsistent"} defines the status of the checking element,
b_i	are XML expressions or XML constraints which extract a model element from the metamodel and the requirement, and then verify their consistency.

Example 2. *Constraint checking rule*

Figure 6 presents an example of constraint checking rules formulated as an XML clause. In particular, $C_{\text{constraint}}$'s body specifies that if there is a constraint defined by the metamodel and restricting class $S:Class1 from having a subclass, and such a class in the user requirement model violates this (by having $S:NewClass as its

```
C_constraint : <ConstraintChecking>
          <Consistency type="SubClass Restriction" name = $S:Class1>
               Specialization
          </Consistency>
          <MMConstraint>    $E:MetamodelConstraint </MMConstraint>
          <ReqElement>      $E:Generalization   </ReqElement>
          <ConstraintStatus> inconsistent          </ConstraintStatus>
      </ConstraintChecking>
```



```
//     Extract the constraint
//  ($E:MetamodelConstraint) of
//  a particular class which is
//  specified within the
//  metamodel and is restricted
//  that $S:Class1 cannot have a
//  subclass.
```

```
//     Extract the generalization
//  element which has a
//  particular class as a parent
//  class ($S:Class1ID) and
//  contains a child class
//  ($S:NewClassID) and is
//  specified within the user
//  requirement model.
```

Fig. 6. An example of a constraint checking rule.

```
C_generation : <SpecificApplicationModel>
                  $E:ModelElement
               </SpecificApplicationModel>

                          ←

<ModelConstruction type= $S:type>          //     A variable $S:type can be instantated into the
    $E:ModelElement                        //  three types of UML model elements: UML:DataType,
</ModelConstruction>,                      //  UML:Association and UML:Class.
[ $S:type MemberOf
    {UML:DataType, UML:Association, UML:Class} ].
```

Fig. 7. A rule for generating a specific application model.

subclass), then a ConstraintChecking-element described by $C_{\text{constraint}}$'s head will be generated with the inconsistency status value equal to "inconsistent".

3.2.2. *Model generation knowledge*

Referring to UML/XMI serialization, a UML class diagram consists of three model elements, namely (a) UML:DataType-element, (b) UML:Class-element and

(c) UML:Association-element. TMG generates a specific application model with respect to these three elements.

Clause $C_{\text{generation}}$ in Fig. 7 specifies that a specific application model is constructed by combining the model construction of the three types of UML elements.

Definition 3 below describes the formulation of a model construction rule.

Definition 3. A rule for constructing a model element of type t is formulated as an XML clause of the form:

```
<ModelConstruction type = t >
            e
</ModelConstuction>
                              ← b1, b2 , ..., bn .
```

where

t	$\in$ {UML:DataType, UML:Assoication, UML:Class},
e	is an XML expression describing the model element to be constructed and conforming to UML/XMI syntax,
b_i	are XML expressions or XML constraints which extract corresponding model elements from the metamodel and the requirement model, and also define how to construct the resulting model element e.

4. An Application of the TMG Framework: Bank Account System

A bank account system is employed to demonstrate the TMG approach, as it is a well known example often referred to in software engineering literature. It shows the main idea of the application concepts, the user requirements of which can be clearly differentiated.

4.1. *Generic application model*

Figure 8 illustrates a metamodel of a general bank account system, which is a representation of the class diagram of Fig. 1(a). It comprises five main classes: Account, Customer, ATMCard, SavingAccount and CurrentAccount, while an OCL constraint associated with a particular class is specified within a dash-rectangle with a dash line linked to that class. In the example, the constraint self.isLeft() restricts the ATMCard class from having a subclass. Moreover, note that in order to specify that a particular metamodel element is mandatory, one can formulate the following OCL constraint and associate it with that metamodel element: object.allInstance() $\rightarrow$ notEmpty(). Assume that all metamodel's elements are mandatory objects in this example; hence for ease of presentation, the OCL constraint restricting this is omitted from the figure.

Corresponding to Fig. 8, Fig. 9 shows a fragment of the metamodel elements represented in UML/XMI and the OCL constraint represented in OCL/XML, respectively. Note that the element types Metamodel and MetamodelConstraint are

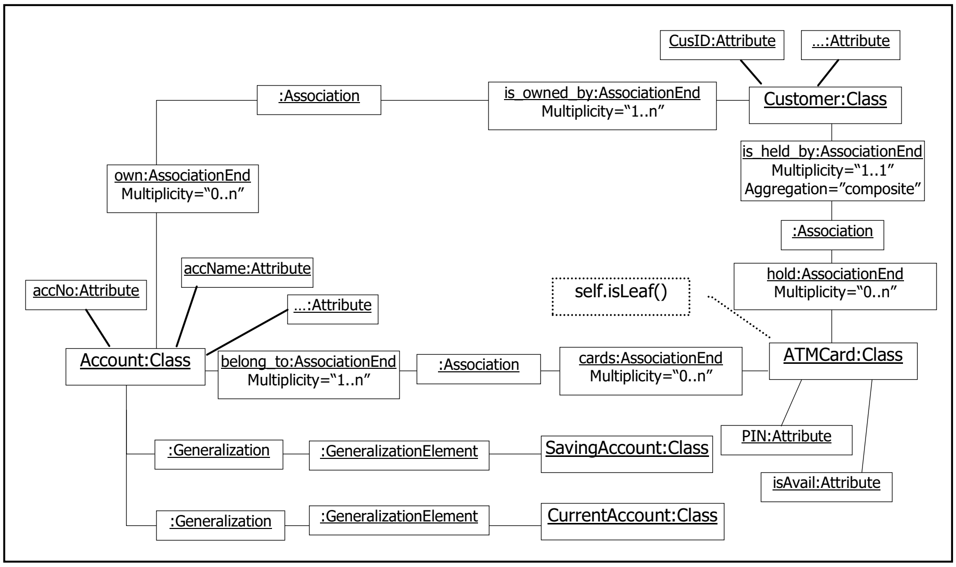


Fig. 8. Bank account metamodel.

used to wrap and denote each model element of a metamodel and a constraint, respectively.

4.2. *User requirement model*

In this example, assume that the user requirements of a new bank account system are concerned with two types of account: Saving-Account and Islamic-Account, while other types are not considered. Based on such requirements, SavingAccount and IslamicAccount classes are modeled as subclasses of Account class. Another requirement deals with a new type of ATMCard called DebitCard. Moreover, the requirement also restricts the number of ATM cards per account to not more than two. This can be simply expressed by specifying the multiplicity of the association-end, which connects ATMCard and Account classes, to be "**1..2**". Based on these requirements, Fig. 10 models them in terms of a UML diagram. Such a diagram, when encoded in XML syntax using UML/XMI, will be wrapped in RequirementModel-element. Figure 11 shows a fragment of such a representation, assuming that the identifiers of the classes Account, ATMCard, IslamicAccount, SavingAccount, and DebitCard are S.1, S.2, S.3, S.4, and S.5, respectively.

4.3. *TMG generation*

This section describes the generation process of TMG. A specific bank account system is produced from the generic application model (Fig. 8) merged with user requirements (Fig. 10) by applying GTKL. First, an example of

```
<Metamodel>
    <UML:DataType xmi.id = 'G.10' name = 'String'/>                                    //      A data type element
</Metamodel>

<Metamodel>
    <UML:Class xmi.id = 'S.1' name = 'Account'  specialization = 'G.13 G.14' >         //      An Account class
         <UML:Classifier.feature>                                                      //    element with an
                         <UML:Attribute xmi.id = 'S.2' name = 'accNo'  type = 'G.10' />  //    attribute accNo
               …
         </UML:Classifier.feature>
         …
    </UML:Class>
</Metamodel>

<Metamodel>
    <UML:Class xmi.id = 'S.7' name = 'SavingAccount' generalization = 'G.13' >         //      A SavingAccount
         <UML:Namespace.ownedElement>                                                  //    subclass element
               <UML:Generalization xmi.id ='G.13' name='' child='S.7' parent ='S.1'/>
         </UML:Namespace.ownedElement>
    </UML:Class>
</Metamodel>

<Metamodel>
    <UML:Association xmi.id = 'G.1' name = '' >                                        //      An Association
         <UML:Association.connection>                                                  //    element
         …
          <UML:AssociationEnd xmi.id='G.2' name='cards'
               isNavigable='false' ordering='unordered' aggregation='none' type='S.4'>
               <UML:AssociationEnd.multiplicity>
                     <UML:Multiplicity >
                           <UML:MultiplicityRange xmi.id = 'id.1' lower = '0' upper = '-1' />
                     </UML:Multiplicity>
               </UML:AssociationEnd.multiplicity>
          </UML:AssociationEnd>
         </UML:Association.connection>
    </UML:Association>
</Metamodel>

 <MetamodelConstraint>
    <ocl:inv class="ATMCards">                                                         //      A constraint:
            <ocl:term>                                                                 //    context ATMCards
                 <ocl:id_term idtype="id">self</ocl:id_term>                           //    inv: self.isleaf()
                 <ocl:term>
                         <ocl:id_term idtype="function"> isLeaf  </ocl:id_term>
                 </ocl:term>
            </ocl:term>
    </ocl:inv>
</MetamodelConstraint>
```

Fig. 9. A fragment of the bank account metamodel and constraint representation.

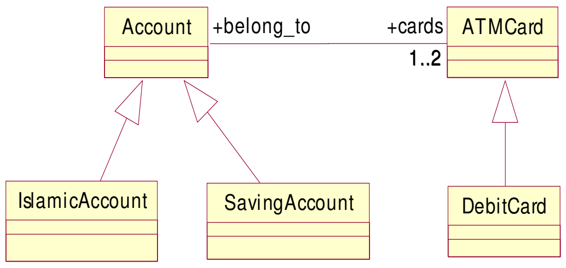


Fig. 10. A UML class diagram representing user requirements.

```
<RequirementModel>
  <UML:Class xmi.id = 'S.5'  name = 'DebitCard' generalization = 'G.6' >          //      DebitCard class is a
    <UML:Namespace.ownedElement>                                                  //   subclass of ATMCard
      <UML:Generalization xmi.id = 'G.6' child = 'S.5' parent = ‘S.2’ />          //   class
   </UML:Namespace.ownedElement>
  </UML:Class>
</RequirementModel>

<RequirementModel>
  <UML:Association xmi.id = 'G.1' name = '' visibility = 'public' >               //      An association
   <UML:Association.connection>                                                   //   between ATMCard and
      <UML:AssociationEnd xmi.id = 'G.2' name = 'bloeng_to' isNavigable = 'false' //   Account classes
          ordering = 'unordered' aggregation = 'none' type = ‘S.1’ >
       ....
      </UML:AssociationEnd>
      <UML:AssociationEnd xmi.id = 'G.3' name = 'cards' isNavigable = 'false'
         ordering = 'unordered' aggregation = 'none' type = ‘S.2’ >
        <UML:AssociationEnd.multiplicity>
          <UML:Multiplicity >
            <UML:Multiplicity.range>
              <UML:MultiplicityRange xmi.id = 'id.2' lower = '1' upper = '2' />   //      A lower and upper
            </UML:Multiplicity.range>                                             //   multiplicity value of
          </UML:Multiplicity>                                                     //   the association end role
        </UML:AssociationEnd.multiplicity>                                        //   named “cards”.
      </UML:AssociationEnd>
   </UML:Association.connection>
  </UML:Association>
</RequirementModel>
```

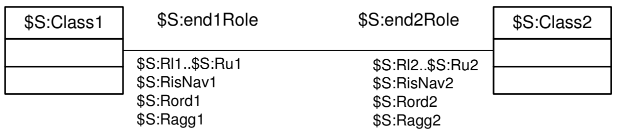
Fig. 11. A fragment of an XMI representation of the user requirements.

generating UML:Assoication-element is demonstrated, followed by another which generates UML:Class-elements. Finally, the resulting application model — a specific bank account system — is obtained by integrating the generated UML:DataType, UML:Assoication and UML:Class elements. Note that since generating UML:DataType-elements can be simply accomplished by a union of the UML:DataType-elements defined in the metamodel and in the requirement model, an example XML clause demonstrating this task is omitted.

Example 3. *Generation of an association element*

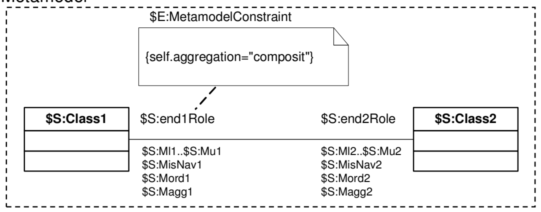
Clause $C_{\text{AssocConstruction}}$ in Fig. 12 depicts an example of the construction of a particular UML:Association element between classes \$S:Class1 and \$S:Class2 by considering the corresponding association defined by the metamodel and the requirement model. In brief, it specifies that an association, defined by the requirement model, between \$S:Class1 and \$S:Class2 with the association end roles \$S:end1Role and \$S:end2Role will be included in the target model if both the diagrammatic checking and constraint checking against the metamodel do not yield the “inconsistent” status. More specifically, the clause performs diagrammatic checking on MultiplicityRange.lower, MultiplicityRange.upper and constraint checking on Aggregation-value of an association role named \$S:end1Role.

```
C_AssoConstruction : <ModelConstruction type="UML:Assoication">
```



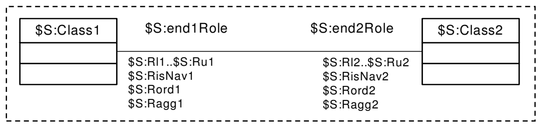
```
</ModelConstruction>
```

Metamodel



// Extracting the property values of
// two end roles of a particular
// association which relates class
// **$S:Class1** to the class **$S:Class2** and
// is specified within the metamodel.

RequirementModel



// Extracting the property values of
// two end roles of a particular
// association which relates class
// **$S:Class1** to the class **$S:Class2** and
// is specified within the requirement.

```
<DiagrammaticChecking>
   <Consistency type="AssociationEnd" name= $S:end1Role>
      MultiplicityRange.lower    </Consistency>
   <MMValue>  $S:Ml1  </MMalue>
   <ReqValue>  $S:Rl1   </ReqValue>
   <DiagramStatus> $S:lowerstatus </DiagramStatus>
</DiagrammaticChecking>,
[ $:S:lowerstatus != "inconsistent" ] ,
```

// Diagrammatic checking of
// attribute values of an association end
// named **$S:end1Role**.

// **$S:Ml1** and **$Rl1** store the lower
// multiplicity values come from the
// metamodel and the requirement of
// role **$S:end1Role**, respectively.

```
<DiagrammaticChecking>
   <Consistency type="AssociationEnd" name= $S:end1Role>
      MultiplicityRange.upper    </Consistency>
   <MMValue>  $S:Mu1 </MMValue>
   <ReqValue>  $S:Ru1   </ReqValue>
   <DiagramStatus> $S:upperstatus </DiagramStatus>
</DiagrammaticChecking>,
[ $S:upperstatus != "inconsistent" ] ,
…
```

// **$S:Mu1** and **$Ru1** store the upper
// multiplicity values come from the
// metamodel and the requirement of
// role **$S:end1Role**, respectively.

// Diagrammatic checking of the
// remaining attribute values
// (isNavigable, ordering and aggregation)
// is performed in the same manner.

```
<ConstraintChecking>
   <Consistency type="AttributeValue Restriction" name=$S:end1Role>
        Aggregation Value    </Consistency>
   <MMConstraint> $E:MetamodelConstraint </MMConstraint>
   <ReqElement> $S:Ragg1 </ReqElement>
   <ConstraintStatus> $S:c_status </ConstraintStatus>
</ConstraintChecking>,
[ $:Sc_status != "inconsistent" ].
```

// Constraint checking on the
// aggregation value of the requirement
// model (**$S:Ragg1**) against the
// metamodel constraint
// (**$E:MetamodelConstraint**) of role
// **$S:end1Role**.

Fig. 12. Model construction of UML:Association element.

Example 4. *Generation of a class element*

A UML:Class element can be constructed in two ways: (a) as a class defined by the metamodel that may or may not be refined by the requirement model; or

(b) as a new class defined by the requirement model that does not exist in the metamodel. The first case is simple; it could be accomplished by simply copying the class definition from the metamodel into the resulting model. However, the consistency checking on attributes and methods should be considered. In some cases, it may include new attributes and methods from the user requirements. This new class may have a relation with other classes by means of an UML:Association element as well as a UML:Generalization element.

The clause $C_{\text{ClassConstruction}}$ (Fig. 13) demonstrates an example of the generation of three class elements, namely, \$S:Class1, \$S:SubClass and \$S:NewClass, given that \$S:SubClass is \$S:Class1's subclass in the metamodel and does not exist in the requirement model, while \$S:NewClass is \$S:Class1's subclass in the requirement model and does not exist in the metamodel.

The head of $C_{\text{ClassConstruction}}$ describes the resulting definition of these three classes. The first two elements of $C_{\text{ClassConstruction}}$'s body extract the class definition of \$S:Class1 from the metamodel and the requirement model, respectively. The ConstraintChecking-element verifies that there is no metamodel constraint violated by having \$S:NewClass as \$S:Class1's subclass. The last two XML constraints in the clause's body define how to obtain the sets of attributes (\$E:AttributeSet) and methods (\$E:MethodSet) of \$S:Class1. In particular, the attributes of \$S:Class1 (\$E:AttributeSet) are the union of the set of \$S:Class1's attributes defined by the metamodel (\$E:MMAttr) and the set of those defined in the requirement model (\$E:ReqAttr). The methods of \$S:Class1 are generated in a similar manner.

Referring to the bank account example, if this rule is applied to class Account;

- The bank metamodel: It contains two subclasses named SavingAccount and CurrentAccount.
- The requirement model: It consists of two subclasses named SavingAccount and IslamicAccount.

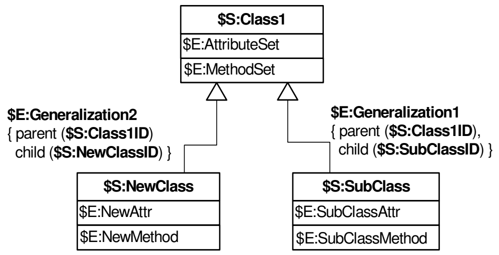
The result of the application of this rule yields three subclasses named SavingAccount, CurrentAccount and IslamicAccout.

Example 5. *Generation of a specific bank account model*

The generation of a specific application model by applying user requirements (Fig. 10) produces a bank account system containing IslamicAccount which is displayed in Fig. 1(c). The generation process can be shown in Fig. 14.

Figure 14 depicts a specific bank account model which can be generated from the inputs. Model elements of a target model are mostly copied from the metamodel. The round dotted blocks indicate certain significant elements that are not defined by the requirement model but are mandatory elements of the metamodel. On the other hand, the elements enclosed by a dashed block are those defined by the requirement and not contained in or different from the metamodel elements, which include: (i) the classes IslamicAccount and DebitCard; (ii) the multiplicity values of a cards role. The bank metamodel includes a constraint that does not allow the ATMCard to have any subclass, hence the class DebitCard is not part of the result.

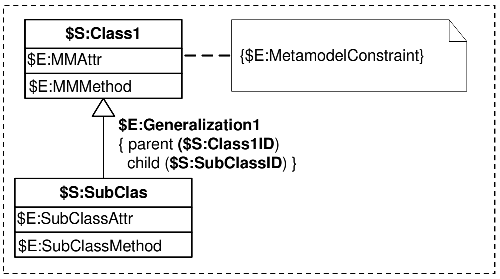
$C_{ClassConstrruction}$: <ModelConstruction type="UML:Class">



</ModelConstruction>

←

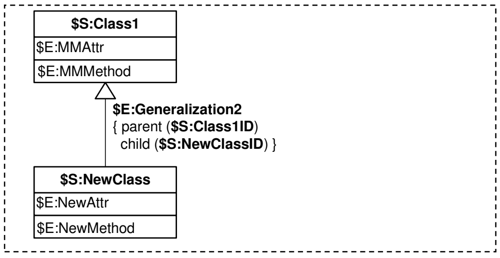
Metamodel



// A generalization element (**$E:Generalization1**) contains an ID of **$S:Class1** as a parent ID and an ID of **$S:SubClass** as a child ID.

[**$S:SubClass** NotExistIn **RequirementModel**]

RequirementModel



// A generalization element (**$E:Generalization2**) contains an ID of **$S:Class1** as a parent ID and an ID of **$S:NewClass** ID as a child ID.

[**$S:NewClass** NotExistIn **Metamodel**]

```
[ $S:NewClass != $S:SubClass ]
<ConstraintChecking>
  <Consistency type="SubClass Restriction" name= $S:Class1>
     Specialization     </Consistency>
   <MMConstraint>    $E:MetamodelConstraint </MMConstraint>
   <ReqElement>      $E:Generalization2 </ReqElement>
   <ConstraintStatus> $S:c_status          </ConstraintStatus>
</ConstraintChecking> ,
[ $S:c_status != "inconsistent" ] ,

[ $E:AttributeSet = $E:MMAttr union $E:ReqAttr ] ,
[ $E:MethodSet = $E:MMMethod union $E:ReqMethod ] .
```

Fig. 13. Model construction of UML:Class elements (a new subclass generating).

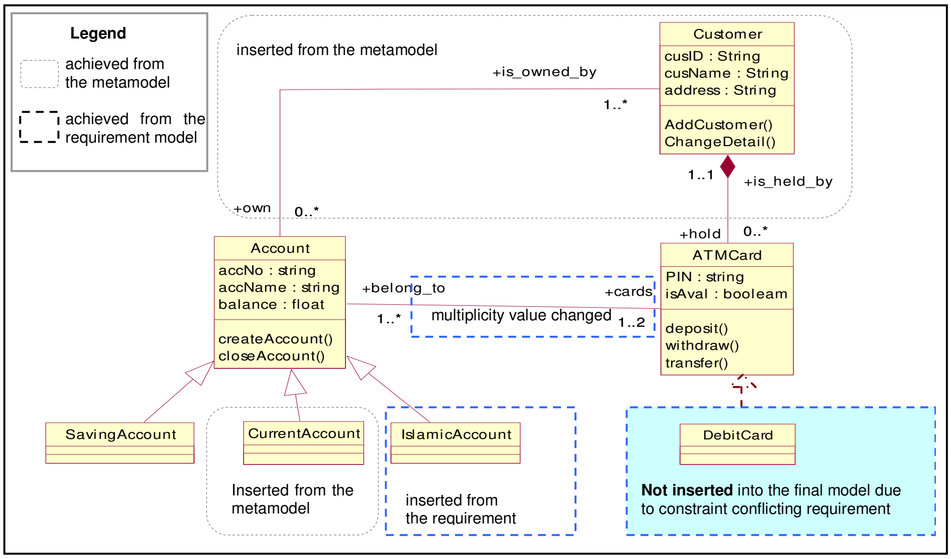


Fig. 14. A generation of a specific bank account model.

With reference to the generic application model, the IslamicAccount class will be added into the model as another subclass of Account class. The DebitCard class is removed since it is not consistent with respect to the application constraints. The multiplicity of the association-end named '**cards**' is set to "**1..2**".

5. Related Works

The separation of PIM abstractions into different levels has been proposed by Exertier *et al.* [10]. PIM abstractions are classified from a higher level to a more detailed level as *context PIM*, *requirements PIM*, *analysis PIM* and *component design PIM*. *Context PIM* describes business principles and high-level services or events offered by the system. *Requirements PIM* defines both functional and non-functional requirements and their relationships. *Analysis PIM* holds the functional specification of the system, focusing on domain and application areas, and complies with the principle of the separation of concerns. *Component design PIM* represents a platform-independent solution expressed in terms of software components and can be used to produce several PSMs with respect to the component platforms. The development process begins with the construction of the *context PIM*, which is then further refined into *component design PIM*. This research work has put forward the concept of the separation of PIM abstractions but mentioned nothing about PIM-PIM transformations.

He *et al.* [15] presented an approach to modelling platform-independent web applications based on template role models and MDA. A template role model is a role collaboration created from UML profiles for patterns. It is designed from common models which are applicable to multiple functions for web applications. A *basic PIM* is constructed from a role model and role constraints. Role constraints involve inclusion of additional requirements. A *basic PIM* is polished into a *refined PIM* from which role composition, encapsulation of common logic, and other design patterns are extracted. The approach advocates the benefit of reusability of levels of abstractions: a template role model can be reused for several functions; a *basic PIM* can be reused to create a number of *refined PIMs*; and a *refined PIM* can be reused to generate systems for different platforms.

The concept of PIM abstractions in the above research work underpins the idea of the separation of PIM levels in TMG. However, the research focuses on the development of models without describing how new or specific user requirements are integrated or employed in the development.

Garcia *et al.* [13], using the concepts of MDA and product line approach, suggested an approach of automating the generation of a PIM of a system from user requirements. In the product line context, a domain PIM normally consists of common and variable parts of business logic. The common part forms the basic architecture of the generic PIM of the domain; whereas the variable one depicts points of variation within that generic PIM and contains all possible user requirements, a set of rules and constrains associated with them. These variation points will be instantiated to specific values by decisions made by a particular customer, yielding a specific PIM for that customer. To aid customers, all the decisions to be made by them are grouped into a Decision Model, the structure of which is specified by a common Meta-Model mechanism. XML and its related technologies such as XML Schema, XSL and XMI are claimed to be the basic tools for the representation and implementation of the PIM, Decision Model and Meta-Model. The idea of generating a specific PIM from combined information from a generic PIM and a specific user's requirements is similar to that of TMG. However, this approach represents specific users' requirements by a Decision Model consisting of a set of decisions; whereas TMG expresses them in terms of UML diagrams. UML diagrams are more expressive, standardized and widely accepted than ad hoc Decision Models, though they are represented in XML Schema, a standard schema language. Moreover, a Decision Model that covers all possible user requirements, rules and constraints is difficult to construct. It is not explicitly given how such a model is generated from a common Meta-Model.

Model and Function Driven Development (MFDD) [25] is an Information System (IS) development technique which focuses on reusing Conceptual Schemas (CSs). A CS comprises two kinds of knowledge: domain model and functions. The domain model defines the general knowledge on the domain and consists of the entity and relationship types, the integrity constraints, the derivation rules and the domain event types with their effects. The functions define the query types that the IS must respond to. Two levels of conceptual schemas were proposed, namely generic CS and specific CS. A generic CS contains a domain's generic model which consists of parts common to all or many CSs of that domain. A specific CS is the CS of a particular IS obtained by refining a generic CS. The concept of refining generic CSs to create specific CSs is similar to the approach of TMG, but refining details are not given.

The main originality of TMG is the proposal and development of a concrete framework for the generation of specific PIMs from a generic PIM and new specific user requirements by means of transformations of XMI/XML documents. These documents are hidden from the user who only sees their corresponding UML diagrams or OCL constraints.

6. Conclusions

This paper proposes a new approach, TMG, to the automatic generation of a specific application model from its generic application model. A generic application model consists of the general features of the corresponding application domain, while a specific one additionally contains supplementary requirements unique to a particular design case. Both of them are conceptual models since no implementation platform details are included; and hence can be considered as a generic PIM and a specific PIM, respectively. A generic application model is created only once but can be reused to generate several specific application models of individual design cases. As a result, the development time is significantly reduced. In other words, this approach enables developers to primarily focus on modeling a system's specific requirements, while letting other mandatory parts be reused and generated automatically. In addition, by using GTKL this approach can detect and handle inconsistencies between a generic application model and its specific user requirements. It produces valid and consistent application models acceptable to users. Moreover, it provides direct transformation of OCL constraints, an augmentation to UML models, and unifies the constraint and the models by means of UML/XMI serialization. Consequently, the transformation approach is more powerful than others.

The main contribution of this paper is the proposal and development of a novel approach to software development by means of model transformation at the MDA's PIM abstract level. It introduces the concept of design-case and platform independent, generic application models which can be reused to generate various design-case specific application models. In addition to model integration and generation, the approach can also manage model consistency. Moreover, by employment of XML Declarative Description, all software artifacts expressed in XMI can directly be handled, processed and computed. None of the existing related approaches can deal with such artifacts in a similar manner.

TMG can be readily applied to generate other types of UML diagrams by the addition of necessary knowledge of the diagrams to GTKL. Moreover, it can be extended to provide PIM-PSM transformations by enhancing GTKL with the knowledge of platform architectures such as CORBA or EJB and their mapping from PIM. The implementation of the TMG approach is being developed by the employment of a programming language called XML Equivalent Transformation (XET)

[3]. XML clauses are coded as XET rules which will operate directly on a UML diagram expressed in terms of an XMI/XML document. In other words, XET rules transform a UML diagram via the transformation of its corresponding XMI document until a satisfactory diagram is obtained.

References

1. K. Akama, T. Shimitsu, and E. Miyamoto, Solving problems by equivalent transformation of declarative programs, *J. Japanese Society of Artificial Intelligence* **13**(6) (1998) 944–952, in Japanese.
2. K. Akama and E. Nantajeewarawat, Formalization of the equivalent transformation computation model, *J. Advanced Computational Intelligence and Intelligent Informatics* **10**(3) (2006) 245–259.
3. C. Anutariya *et al.*, An equivalent-transformation-based XML rule language, in *Int. Workshop Rule Markup Languages for Business Rules in the Semantic Web*, Sardinia, Italy, 2002.
4. J. Bězivin and N. Ploquin, Tooling the MDA framework: A new software maintenance and evolution scheme proposal, *J. Object-Oriented Programming* (*JOOP*), December 2001.
5. G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide* (Addison Wesley, 1998).
6. E. S. Borch, W. J. Jespersen, B. Kinvald, and K. Osterbye, A model driven architecture for REA based systems, in *Proc. Workshop on Model Driven Architecture: Foundations and Application*, TR-CTIT-03-27, University of Twente, 2003, pp. 103–107.
7. K. Czarnecki and S. Helsen, Classification of model transformation approaches, in *Proc. 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, USA, 2003.
8. P. Desfray, MDA — When a major software industry trend meets our toolset, implemented since 1994, *SOFTEAM* (2001).
9. A. S. Evans, Reasoning with UML class diagrams, *Second IEEE Workshop on Industrial Strength Formal Specification Techniques* (*WIFT'98*) (IEEE Computer Society Press, 1998), pp. 102–113.
10. D. Exertier, B. Langlois, and X. L. Roux, PIM Definition and Description, in *First European Workshop Model-Driven Architecture with Emphasis on Industrial Applications*, Netherlands, 2004.
11. L. Favre, Foundations for MDA-based forward engineering, *J. Object Technology* **4** (2005) 129–153.
12. Y. Fujikawa and T. Matsutsuka, New web application development tool and its MDA-based support methodology, *FUJITSU Sci. Tech. J.* **40** (2004) 94–101.
13. B. A. Garcia, J. Mansell, and D. Sellier, From Customer Requirements to PIM: Necessity and Reality, European Software Institute, Bizkaia, Spain, November, 2004, http://www.metamodel.com/wisme-2002/papers/belenGarcia.pdf.
14. N. Guelfi and G. Perrouin, Using model transformation and architectural frameworks to support the software development process: The FIDJI approach, in *Midwest Software Engineering Conference*, Chicago, IL, 2004.
15. C. He, F. He, K.-Q. He, and W. Tu, Constructing platform independent models of web application, in *Proc. 2005 IEEE Int. Workshop on Service-Oriented System Engineering* (*SOSE'05*), Beijing, October, 2005.
16. F. Marschall and P. Braun, *Model Transformations for the MDA with BOTL*, in

A. Rensink (ed.), CTIT Technical Report TR-CTIT-03-27, University of Twente, The Netherlands, 2003, pp. 25–36.

17. J. Mazon, J. Trujilo, M. Serrano, and M. Piattini, Applying MDA to the development of data warehouse, in *8th Int. Workshop on Data Warehousing and OLAP*, Bremen, Germany, November, 2005.
18. S. Meliá, C. Cachero, and J. Gômez, Using MDA in web software architectures, in *2nd Int. Workshop on Generative Techniques in the Context of MDA*, California, USA, 2003.
19. A. Naco and V. Wuwongse, Representing of OCL in XML syntax, *Research Proposal Entitle Reasoning with UML Diagrams*, Technical report, Asian Institute of Technology, Thailand, 2002.
20. OMG, Common Warehouse Metadata (CWM) 1.1 Specification, Object Management Group, Document formal/2002-03-03 (2002).
21. OMG, Meta Object Facility (MOF) 1.4 Specification, Object Management Group, Document formal/2002-04-03 (2002).
22. OMG, Model Driven Architecture Guide Version 1.0.1, Object Management Group, Document omg/03-06-01 (2001).
23. OMG, Unified Modeling Language (UML) 1.5 Specification, Object Management Group, Document formal/2003-03-01 (2003).
24. OMG, XML Metadata Interchange (XMI) 1.2 Specification, Object Management Group, Document formal/2002-01-01 (2002).
25. R. Raventos, Model and function driven development, in *Doctoral Symposium, 7th Int. Conf. Unified Modeling Language*, Portugal, October, 2004.
26. S. Sendall and W. Kozaczynski, Model transformation — the heart and soul of model-driven software development, *IEEE Software* **20**(5) (2003) 42–45.
27. The QVT-Merge Group, MOF 2.0 Query/Views/Transformations, revised submission, OMG document ad/2004-04-01 (2004).
28. J. Warmer and A. Kleeppe, *The Object Constraint Language: Getting your Models Ready for MDA*, 2nd edn. (Addison Wesley, 2003).
29. E. Willink, *The UMLX Language Definition*, http://www.eclipse.org/gmt-home/doc/umlx/umlx.pdf.
30. V. Wuwongse, C. Anutariya, K. Akama, and E. Nantajeewarawat, XML Declarative Description (XDD): A language for the semantic web, *IEEE Intelligent Systems* **16**(3) (2001) 54–65.
31. W3C, XSL Transformations (XSLT) Version 1.0. W3C Recommendation, November 1999.